Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs

Lingxiao Jiang Zhendong Su
Department of Computer Science
University of California, Davis, CA 95616, U.S.A.
{jiangl, su}@cs.ucdavis.edu

ABSTRACT

Misuse of measurement units is a common source of errors in scientific applications, but standard type systems do not prevent such errors. Dimensional analysis in physics can be used to manually detect such errors in physical equations. It is, however, not feasible to perform such manual analysis for programs computing physical equations because of code complexity. In this paper, we present a type system to *automatically* detect potential errors involving measurement units. It is constraint-based: we model units as types and flow of units as constraints. However, standard type checking algorithms are not powerful enough to handle units because of their abelian group nature (*e.g.*, being commutative, multiplicative, and associative). Our system combines techniques such as type inference and Gaussian Elimination to overcome this problem. We have implemented Osprey, a prototype of the system for C programs, and evaluated it on various test programs, including computational physics and mechanical engineering applications. Osprey discovered unknown errors in mature code; it is precise with few false positives; it is also efficient and scales to large programs—we have successfully used it to analyze programs with hundreds of thousands of lines of code.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*reliability, validation*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms: Languages, Reliability, Verification

Keywords: Gaussian Elimination, constraint-based analysis, dimensional analysis, measurement units, type systems

1. INTRODUCTION

Scientific applications use measurement units such as meters, seconds, or kilograms. Misuse of measurement units in these applications can be disastrous: it is believed that the Mars Climate Orbiter is lost because data denominated in the English system was fed into the navigation system which expected metric units [18]. In order to have correct computational results, it is important to validate dimensional unit correctness of a program. However, standard type systems do not enforce the correct use of units.

ICSE'06, May 20–28, 2006, Shanghai, China.

Physicists routinely use *dimensional analysis* to check the dimensional unit correctness of quantities in equations. Dimensional analysis assumes that each physical quantity has a meaningful, fixed unit of measure and the units of both sides of an equation are the same. Although useful, such analysis can be difficult to carry out, especially for non-physicists. Many physical equations involve complicated computation, and it is difficult to track the flow of units in those equations. Manually applying dimensional analysis to programs that calculate such equations is even more complicated.

We use the concrete example in Figure 1 to illustrate unit errors and explain our analysis. For now, please ignore those shaded tokens starting with a `$`, such as `$unity`: they are unit annotations for our type system. The code computes an electron's final energy using the formula (adapted from Brown's work on SIUNITS [4]):

$$\frac{1}{X_0} = 4\alpha r_e^2 \frac{N_A}{A}\{Z^2[L_{rad} - f(Z)] + ZL_{rad}^{'}\}$$

We briefly explain the physical meaning of the variables: α is the fine structure constant; r_e is the radius of an electron; N_A is the Avogadro's number; L_{rad} and $L_{rad}^{'}$ are Tsai's constants; X_0 is *radiation length*; and $f(Z)$ is treated as zero in the code.

The code has two unit errors, neither of which can be detected by the standard C type checker: (1) Although the name of the function `radiationLength` implies that the unit of its return value is that of $length$, it is actually a unit of $area\ density$, for example, $kilogram{*}meter^{-2}$. The return statement in `radiationLength` should return the reciprocal of the original expression; (2) The argument of `exp` in the return statement of `finalEnergy` should be $unitless$ according to the Π theorem in physics.[1] Considering this together with the first error, the argument should be "`thick * density / X0`." Unit errors occur for many reasons, such as misunderstanding physical meaning of equations or simply programming errors. It is difficult for programmers to apply manual dimensional analysis to discover these errors because of function calls, structures, pointers, and other language constructs. It is thus desirable to mechanize dimensional analysis. Although many approaches exist to support automatic dimensional analysis, there is not yet a practical method for verifying unit correctness of large C programs. We defer a detailed survey of related work to Section 7.

In this paper, we design a type system for automatic dimensional analysis. In our system, we model units as types in programming languages and reduce dimensional analysis to type checking. However, the semantics of units is more complicated than that of standard types. Units can be operated on with arithmetic operations, such as multiplication, division, and square root, and they form an

[1] According to the Buckingham's Π theorem [9] in physics, parameters and return values of a transcendental function (*e.g.*, exponential, logarithmic, or trigonometric functions) should be $unitless$ [26].

```
1  double pow(double, $unity double);
2  $unity double log( $unity double);
3  $unity double exp( $unity double);

4  extern $unity double alpha, NA;
5  extern $meter double re;

6  typedef struct {
7    $kilogram double atomicWeight;
8    $unity double atomicNumber;
9  } Element;

10 double radiationLength(Element * material) {
11   double A = material->atomicWeight;
12   double Z = material->atomicNumber;
13   double L = log( 184.15 / pow(Z, 1.0/3) );
14   double Lp = log( 1194.0 / pow(Z, 2.0/3) );
15   return ( 4.0*alpha*re*re ) * ( NA/A )
16        * ( Z*Z*L + Z*Lp );
17 }

18 double finalEnergy(Element * material,
19     $kilogram*meter^-3 double density,
20     $meter             double thick,
21     $kilogram*meter^2*second^-2 double initEnergy)
22 {  double X0 = radiationLength(material);
23    return initEnergy / exp( thick / X0 );
24 }
```

Figure 1: Sample code with unit errors.

abelian group.[2] We thus need more powerful algorithms to perform type checking for unit correctness. Our type checking algorithm combines both standard type checking and Gaussian Elimination methods to validate units. As a novel contribution, our system can also validate the factors used for converting one unit to another of the same dimension. Our goal is to have a system that is *sound* (does not miss any errors), *scalable* (can analyze large programs), *precise* (does not report many spurious errors), and *usable* (is easy for programmers to use).

We have implemented Osprey, a prototype of the system for C programs meeting this goal. Ignoring certain unsafe features of C, Osprey is sound: if it does not find unit errors in a program, then the program is guaranteed to be free of unit errors. To validate the other claims (*i.e.*, being precise, scalable, and usable), we have extensively evaluated Osprey on various test programs, including computational physics and mechanical engineering applications. Osprey discovered unknown errors in mature code. It is also precise with few false positives in our experiments. It is efficient and scales to large programs with hundreds of thousands of lines of code. It is also easy to use because it requires only lightweight annotations (in the form of simple type qualifiers) and is fully automatic.

The rest of the paper is structured as follows. We first give an overview of our system (Section 2). We then present details of the components in the system (Section 3), followed by a discussion of its implementation (Section 4). Next, we show experimental results and evaluation of Osprey (Section 5) and discuss its current limitations and possible ways to enhance it (Section 6). Finally, we discuss related work (Section 7) and conclude (Section 8).

[2] An abelian group is a finite or infinite set of elements together with a binary operation (with multiplication as the operation on units) satisfying a few properties: closure, associativity, commutativity, and existence of identity and inverses.

```
ERROR: The constraint:
    u_20_thick = u_23_thick_DIV_X0 * u_22_X0
is reduced to:
    meter^1 = meter^2kilogram^-1.
```

Figure 2: Sample error report for code in Figure 1.

2. OVERVIEW OF OUR APPROACH

Our analysis is cast as a constraint-based type inference system, consisting of a definition of types, a set of type checking rules, a constraint generation phase, and a constraint solving phase. Given a program, constraints are generated based on the definition of types and type checking rules. The constraints are then solved, and errors will be reported if the constraints are unsolvable. In the following, we present the type system along with its prototype implementation Osprey, to make it more concrete.

2.1 Users' View

To users, our system works like a standard type system. Users assign types (units) to program variables and other objects, and the system checks type correctness of the program and may issue error reports for users to fix these errors.

In practice, Osprey should be familiar to users because the unit annotations are analogous to types. Consider again the sample code in Figure 1. The tokens starting with a `$` are unit annotations. The units represented by these annotations should be self-explanatory; $kilogram^1$*$meter^2$*$second^{-2}$ is actually a unit of $energy$. Osprey provides aliases and abbreviations for commonly used units. For example, the aliases and abbreviations $unity$, m, kg, s, and E are used to represent $unitless$, $meter$, $kilogram$, $second$, and the aforementioned unit of $energy$, respectively. Our later discussions will use some of these abbreviations.

Osprey issues the error report shown in Figure 2 for the sample code. In the error report, `u_20_thick` represents the unit of `thick` declared on line 20; `u_23_thick_DIV_X0` represents the unit of the expression "`thick / X0`" on line 23; `u_22_X0` represents the unit of "`X0`" declared on line 22. The division in the original program is rephrased as multiplication in the error report.

Such a report means that the code corresponding to these unit variables contains a unit error. By examining the code in Figure 1, we see that on line 23, the unit of the argument for `exp` must be $unitless$ (according to the Π theorem, *cf.* Footnote 1), and thus `u_23_thick_DIV_X0` is $unity$ and `X0` should have the same unit as `thick`, *i.e.*, $meter$, but in fact it is $meter^2$*$kilogram^{-1}$ according to the error report. After checking the origin of the value of `X0`, we know that either the return value of `radiationLength` or the way we use the function is problematic. Thus, such error reports may help users to fix the errors mentioned in Section 1.

2.2 Internal View

Figure 3 depicts the internals of Osprey. We use a specialized type definition for units (Section 3.2) and a set of unit constraint generation rules (Section 3.3) for the constraint generation phase. Because of the abelian group nature of units, the generated constraints may involve equalities, multiplications, or inverses. The constraints that involve only equalities are resolved by the constraint resolution engine—*Banshee* [17]. We then use the (partial) solution from this phase and simplify all constraints using a tailored *union/find* (U/F) engine to reduce the number of unit variables and constraints. The result is subsequently fed to a *Gaussian*

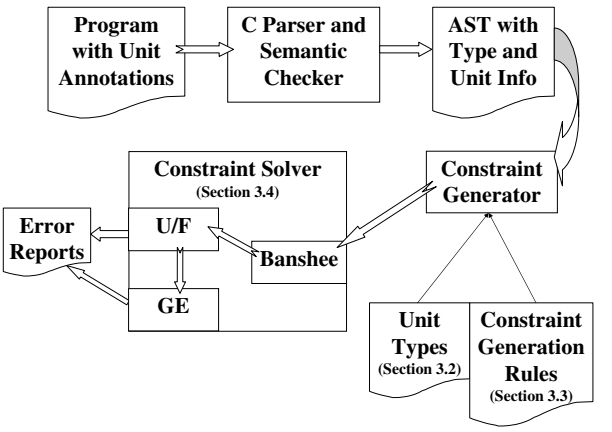

Figure 3: Internal structure of Osprey.

Elimination (GE) engine (Section 3.4). During this solving phase, whenever a unit error is discovered, an error report will be issued to inform users of the error.

3. TYPE SYSTEM FOR UNIT CHECKING

3.1 Dimensions and Units

We first introduce properties of dimensions and units. Every dimension can be derived from the seven base dimensions in the International System of Units (SI) [12]. Each base dimension has a corresponding base unit, but may have more than one unit. For example, $meter$ is the base unit of $length$, while $centimeter$ and $foot$ are also units of $length$. Each unit of a dimension can usually be converted to other units of the same dimension by multiplying a *unit factor*. For example, 0.01 is the unit factor converting $centimeter$ to $meter$ because $1meter = 100centimeter$. Unit prefixes in SI, such as $kilo$ and $milli$, are used to derive units and can be viewed as unit factors.

3.2 Unit Types

We model units as types and define a *unit type language*:

$$\begin{aligned} \mathtt{ut} &::= meter \mid kilogram \mid second \mid ampere \mid kelvin \\ &\mid mole \mid candela \mid unity \mid \mathtt{ut}_1 * \mathtt{ut}_2 \mid \mathtt{ut}^{-1} \mid f \mid \delta \\ \mathtt{cut} &::= \mathtt{ut} \mid ref(\mathtt{cut}) \mid struct(\mathtt{cut}_1, \ldots, \mathtt{cut}_n) \\ &\mid lam(\mathtt{cut}_0, \mathtt{cut}_1, \ldots, \mathtt{cut}_n) \end{aligned}$$

The abelian group for units is defined by the grammar for ut: The first seven elements are the seven base units; $unity$ denotes the identity; multiplication is denoted by the symbol $*$; $\mathtt{ut}^{-1}$ denotes the inverse element of ut; and the symbol f denotes a unit factor. We also introduce unit variables, δ, to represent unknown units. Unit types without variables, such as $meter$ and "$kilogram * meter^{-2} * 2.2$," are called *unit constants*.

To express programming language constructs, we also introduce *composite unit types* (cut). The last three production rules for cut define unit types for pointers, structures, and functions respectively. In lam, $\mathtt{cut}_0$ denotes the unit type of a return value; $\mathtt{cut}_1, \ldots, \mathtt{cut}_n$ denote the unit types of fields (of a structure) or parameters (of a function). These three kinds of unit types have no real physical meaning, but they are helpful for tracking flow of units over these language constructs. For example, in the code in Figure 1, the argument `material` to the function `radiationLength` is of the type $ref(struct(kilogram, unity))$.

3.3 Unit Constraints

We now introduce *unit constraints* to model the flow of units in a program. Unit constraints are mainly of two forms: $u_a = u_b$ or $u_a = u_b * u_c$, where u_a, u_b, and u_c are either unit variables or constants. Due to space constraints, instead of giving the formal constraint generation rules in our system, we illustrate constraint generation in Osprey with the sample code in Figure 1. Interested readers can find a formal description in the full paper [13].

We follow the standard technique of constraint generation in constraint-based program analysis. The idea is natural: we essentially perform a recursive traversal of the abstract syntax tree (AST) of a program and generate constraints for each node based on the node's corresponding generation rule. Constraint generation rules can be roughly classified into two categories: declarations and expressions. The former changes the unit environment (which maps program variables to unit types) and may indirectly generate new constraints, while the latter generates new constraints directly and may affect the unit environment.

Figure 4 shows constraints generated for some representative fragments of the code in Figure 1. As for notation, mappings enclosed in [] are to be added into the current environment, and constraints enclosed in {} are to be generated when the corresponding code is being analyzed. We explain some of the rows in the figure:

Row 1 The unit variables `u_2_log@return` (for the return value) and `u_2_log@_1` (for the parameter) are both $unitless$.

Rows 3 and 4 These two rows illustrate how structures are modeled in our system. When a field is defined within a structure, a new mapping for the corresponding unit variable is added, for example, the unit variable `u_6_unamed@atomicWeight` for the field `atomicWeight` in the anonymous structure is mapped to $kilogram$; when a field is accessed, the corresponding unit variable is used to generate constraints, such as the constraint in row 4. A field of a structure corresponds to a fixed unit variable, and thus different instances of the structure always have the same unit. This kind of modeling of fields within a structure is called *field-level field-sensitivity*.

Row 5 This row shows how constants are modeled. A fresh variable `u_13_const#1_DIV_const#2` (for the division) is created and a new constraint among the variables is generated. The variables `u_13_const#1` and `u_13_const#2` are for the second and third constants on line 13 and both $unity$.

Rows 6 and 7 These rows show the effects of the calls to `pow`. According to the function declaration on line 1 in Figure 1, there is no unit annotations for the first parameter and the return value, and Osprey considers them to be *polymorphic* (*i.e.*, different calls to the same function are treated independently and thus the units of the polymorphic elements can be different at the different call sites), while the second parameter is $unitless$. Constraints relating parameters and actual arguments are generated at the call sites. To distinguish the two call sites, different instances of the polymorphic variables are needed. We can see in Figure 4 that the unit variable for the first parameter `u_1_pow@_1` and that for the return value `u_1_pow@return` are instantiated using the position information of the call sites, while the unit variable for the second parameter `u_1_pow@_2` is kept the same. Such a technique is called *syntactical instantiation* and is commonly used to implement context-sensitive analysis. More details on polymorphism and context-sensitivity are given in Section 4.3.

Row 11 This code involves a function call and an assignment. The function call is treated the same as the ones to `pow`, except that we also need to instantiate the set of constraints for the function body, usually referred to as a *function summary* and

Row	Line # and Source Code		Modification to Unit Environment	Generated Constraints
1	2	`$unity...log($unity...);`	[u_2_log@return : $unity$, u_2_log@_1 : $unity$]	$\emptyset$
2	4	`$unity double alpha;`	[u_4_alpha : $unity$]	$\emptyset$
3	7	`$kilogram...atomicWeight;`	[u_6_unamed@atomicWeight : $kilogram$]	$\emptyset$
4	11	`A =...->atomicWeight`	$\emptyset$	{u_11_A = u_5_unamed@atomicWeight}
5	13	`1.0/3`	[u_13_const#1_DIV_const#2 : δ]	{u_13_const#1_DIV_const#2 * u_13_const#2 = u_13_const#1}
6	13	`pow(Z,...)`	[u_1_pow@return_13 : δ]	{u_1_pow@_1_13 = u_12_Z, u_1_pow@_2 = u_13_const#1_DIV_const#2}
7	14	`pow(Z,...)`	[u_1_pow@return_14 : δ]	{u_1_pow@_1_14 = u_12_Z, u_1_pow@_2 = u_14_const#4_DIV_const#5}
8	16	`Z*Lp`	[u_16_Z_MUL_Lp : δ]	{u_16_Z_MUL_Lp = u_12_Z * u_14_Lp}
9	16	`( Z...+...)`	$\emptyset$	{u_16_Z_MUL_Z_MUL_L = u_16_Z_MUL_Lp}
10	22	`double X0 = ...;`	[u_22_X0 : δ]	$\emptyset$
11	22	`X0=radiationLength...`	$\emptyset$	{u_22_X0 = u_10_radiationLength@return_22}

Figure 4: Sample generated constraints.

generated according to the body. The combination of function summaries and syntactical instantiation enables performing inter-procedural analysis efficiently. Due to space limitations, we do not show the complete set of constraints.

Another common situation involves user-defined unit conversions. For example, consider the following code:

```
$millimeter double mm;
$inch double inch;
mm = inch*($f)25.4;
```

Such a program may produce physically meaningful results if `25.4` is used as a unit factor for converting $inch$ to $millimeter$. In order to validate units, Osprey needs to know whether `25.4` is such a unit factor or just an arbitrary constant. Therefore, a user needs to tell Osprey that `25.4` is a unit factor using `$f`. Based on such annotations, Osprey generates a constraint `u_mm = u_inch*25.4` and verifies the correctness of this unit conversion during the subsequent constraint solving phase.

3.4 Constraint Resolution

We now discuss how to solve unit constraints. The general form of a unit constraint is:

$$u_1 * \ldots * u_n = v_1 * \ldots * v_m$$

where u_i's and v_i's are either unit variables or constants. In our analysis, $n + m$ is usually 2 or 3 due to the structure of C abstract syntax trees and the constraint generation rules.

Constraints of the form $u = v$, where u and v are both variables, are standard equality constraints. Given a set of such constraints, Banshee [17] can efficiently compute an *equivalence class representative* (ECR) for each unit variable u, and the unit of u is the same as that of its ECR. If all constraints are in such a form, we can completely rely on Banshee to solve them in linear time.

Constraints that involve multiplications and unit constants, such as $u_1 = u_2 * u_3$ and $u = a$ (a represents a unit constant), require different techniques. Wand and O'Keefe [26] use *Gaussian Elimination* (GE) and a specialized unification algorithm to solve equations. The algorithm in their paper handles fewer units, and their system is presented for the simply-typed lambda calculus. Antoniu *et al.* [3] also suggest solving unit constraints via GE, but they have not fully deployed the algorithm for two reasons: GE is cubic time and incapable of reporting why a linear system is unsolvable.

In order to have a more usable system, especially validating unit conversion factors, we believe GE for solving linear equations is necessary. We adapt Antoniu and Steckler's technique, exploit a union/find algorithm to reduce numbers of unit variables and constraints, re-program the linear system solver in the linear algebra package *CLAPACK* [2], and utilize the line numbers in the naming convention illustrated in Figure 4 to locate sources of unit errors.

Our algorithms are shown as Algorithms 1 and 2. The function REPLACE in Algorithm 1 replaces all variables in a constraint with their ECRs. The constraint is then simplified with REDUCE such that it contains at most one unit constant and no repetitive variables. The simplified constraint is subsequently processed according to its form. For example, if the current ECRs of u_1 and u_3 are m and $m^2{*}kg^{-1}$, respectively, then $u_1 = u_2 * u_3$ can be reduced to $kg{*}m^{-1} = u_2$; $u_4 * u_4 = u_4$ can be reduced to $u_4 = unity$. The ECRs are updated accordingly. Errors may be issued if the units of the two sides of a constraint are not the same.

Algorithm 2 reduces a set of unit constraints to linear systems. Each unit constraint can be transformed to eight linear equations corresponding to the seven base dimensions and one unit factor by taking logarithm. For example, the aforementioned `u_mm = u_inch*25.4` can be transformed to the following linear equations:

$$\begin{aligned}
\mathtt{u_mm}_{meter} - \mathtt{u_inch}_{meter} &= 0 \\
\mathtt{u_mm}_{kilogram} - \mathtt{u_inch}_{kilogram} &= 0 \\
\mathtt{u_mm}_{second} - \mathtt{u_inch}_{second} &= 0 \\
\mathtt{u_mm}_{ampere} - \mathtt{u_inch}_{ampere} &= 0 \\
\mathtt{u_mm}_{kelvin} - \mathtt{u_inch}_{kelvin} &= 0 \\
\mathtt{u_mm}_{mole} - \mathtt{u_inch}_{mole} &= 0 \\
\mathtt{u_mm}_{candela} - \mathtt{u_inch}_{candela} &= 0 \\
\log_{10} \mathtt{u_mm}_{factor} - \log_{10} \mathtt{u_inch}_{factor} &= \log_{10} 25.4
\end{aligned}$$

Such a transformation is performed by TOLINEAREQUATION in Algorithm 2. The resulting linear systems have solutions if and only if there are no unit errors in the original program. We solve the linear equations via *LU Factorization* [20]. The function LUFACTORIZATION decomposes a linear system into a unit lower-triangular matrix L and non-unit upper-triangular matrix U. *Forward substitution* and *backward substitution* [20] then transform L and U to diagonal matrices in turn, via row operations in linear algebra, to obtain a solution. The original solver in CLAPACK has applied these techniques, but we have modified it to handle non-square matrices and singular U whose diagonal elements contain zeros. Also, during backward substitutions, whenever an unsolvable equation (*i.e.*, the left-side coefficients of the equation are all zeros, while its right-side is non-zero) is encountered, the names of the unit variables involved in the equation are reported to help users to locate the source of errors.

3.5 Complexity and Soundness

The constraint generator in our system takes linear time in the size of the input abstract syntax tree. Banshee solves equality con-

Algorithm 1 Union/Find Algorithm for Simplifying Constraints

```
function UF(C : ConstrSet, R : ECRMap)
    repeat
        for all c ∈ C do
            c ← REPLACE(c, R)
            c ← REDUCE(c)
            if c matches 'a = a' or 'u = u' then
                C ← C \ {c}
            else if c matches 'u = a' then
                R ← R[ECR(u) ↦ a];
                C ← C \ {c}
            else if c is of the form 'u1 = u2' then
                R ← R[ECR(u1) ↦ ECR(u2)]
                C ← C \ {c}
            end if
        end for
    until R does not change
    return (C, R)
end function
```

straints in linear time. Each iteration of the **repeat/until** loop in Algorithm 1 takes linear time. Because the number of variables in a unit constraint is usually no more than three, the complete U/F algorithm takes linear time and is capable of reducing many variables and constraints (*cf.* Table 1). The time and space complexity of GE are cubic and quadratic respectively, in the size of the linear system, which is bounded by the size of the program.

Putting everything together, our system requires worst-case cubic time and quadratic space. Notice that the GE step is the bottleneck and thus the U/F step is important to reduce the order of the generated linear systems to improve scalability.

Ignoring certain unsafe features in C, such as type casts, unions, and pointer arithmetic, our type system underlying Osprey is sound: it does not miss any unit errors. Although unit constraints are of the abelian group nature and they are solved using Gaussian Elimination, the proof of soundness for our system still follows that for CQual [8] and is omitted here.

4. IMPLEMENTATION

4.1 Unit Representation

A common way to represent units is based on exponent vectors over base units and unit factors. For example, $m^2 * kg * s^{-2}$, a unit of $energy$, can be represented as $[2, 1, -2, 0, 0, 0, 0] * 1$. Thus, arithmetic operations on units can be reduced to vector additions, subtractions, or comparisons. Compared with this representation, Cunis's [6] prime number-based representation may be more time and space efficient: distinct small prime numbers are used to denote different base units, and each rational is used to represent a unique unit. For example, the above unit can be represented as the rational $12/25 = 2^2 * 3^1 * 5^{-2}$. However, the prime number-based representation can not represent units with non-integer exponents, *e.g.*, the unit of the square root of $energy$. We use the exponent vector-based representation in Osprey.

4.2 Unit Environment

Osprey takes as an additional input a configuration file that allows new definitions for unit prefixes, unit aliases, and unit factors that can be used in unit annotations. For example, "`#define millimeter milli-meter`" defines $millimeter$; "`#define inch meter 39.370079`" defines $inch$ because $1\ meter = 39.370079\ inches$. The definitions are sufficient for unit validation on the code in Section 3.3 (*cf.* Section 3.4 for the resulting linear system).

Algorithm 2 Gaussian Elimination for Solving Unit Constraints

```
function GE(C : ConstrSet, R : ECRMap)
    D ≡ base dimensions ∪ {factor}
    for all d ∈ D do
        LS_d ← ∅
        for all c ∈ C do
            LS_d ← LS_d ∪ TOLINEAREQUATION(c, d)
            LS_d ← LUFACTORIZATION(LS_d)
            LS_d ← FORWARDSUBSTITUTION(LS_d)
            LS_d ← BACKWARDSUBSTITUTION(LS_d)
            R ← UPDATEECRMAP(LS_d, R)
        end for
        return R
    end for
end function
```

Users provide unit annotations for physical quantities in the form of type qualifiers [8]; the number of annotations required is usually small compared to the number of tokens in a program (*cf.* Table 1, column "Annotation Burden").

We adapt the parser of CQual [8] to generate abstract syntax trees and perform standard semantic checking for programs. The unit environment is constructed during constraint generation. We also use the following recursive function to construct the unit type for a variable x based on its C type τ when no appropriate annotations for x are provided:

$$\mathsf{enrich}(\tau, x) \triangleq \begin{cases} ref(\mathsf{enrich}(\tau_1, x)) & \text{if } \tau = ref(\tau_1) \\ struct(\mathsf{enrich}(\tau_1, f_1), \ldots, \mathsf{enrich}(\tau_n, f_n)) & \text{if } \tau = struct(f_1 : \tau_1, \ldots, f_n : \tau_n) \\ lam(\mathsf{enrich}(\tau_0, x_0), \ldots, \mathsf{enrich}(\tau_n, x_n)) & \text{if } \tau = lam(\tau_0, \ldots, \tau_n) \\ \delta_x & \text{otherwise} \end{cases}$$

where pointers, structs, and functions are transformed to reference, structural, and functional units respectively; other un-annotated variables are mapped to fresh unit variables δ_x; un-annotated numerical constants are mapped to $unity$ by default.

Special care is needed to avoid infinite recursions when dealing with recursive types using enrich. For example, consider the following structure declaration:

```
struct list { struct list *next; ... }
```

We can detect that the unit type for `struct list` is a recursive one ($struct(ref(struct(ref(\ldots)))))$ via tracking records of encountered types, and use a dummy unit variable as a ground unit ($struct(ref(\delta_{dummy}))$) to terminate the recursion. This decreases the precision of our analysis and may cause false alarms, but it is efficient and inessential to unit checking.

Many library functions should also be annotated with units. Fortunately, we believe most of them can be treated in the same way as transcendental functions or polymorphic functions. There are situations where users can use side annotations to improve Osprey's precision. For example, the library function "`double sqrt(double x)`" may need an annotation of the form "`u_sqrt * u_sqrt = u_x`" to relate the return value and the parameter; for the library function "`double pow(double base, double power)`," users may need to provide similar annotations at call sites to relate the return value and the first argument.

4.3 Context Sensitivity

Consider the following example of a polymorphic function:

```
δ2 double square (δ1 double a) { return a*a; }
$m double m1;   $kg double k1;
m = square(m1); /* (1) */
k = square(k1); /* (2) */
```

The function `square` can take data in any unit as arguments. In a homomorphic setting, δ_1 is a fixed unit, although its exact unit is not explicitly known. In this case, both the units of `m1` and `k1` flow into δ_1, which causes a unit clash and a false error alarm would be issued at the call site marked (2). With polymorphism, δ_1 and δ_2 are viewed as generic-variables and would be instantiated as different unit variables at the two call sites. Now, the units of `m1` and `k1` flow into different instantiated variables and no error would be issued. In practice, users do not need to use δ explicitly; any return value and parameter without annotations are treated by Osprey to be polymorphic, just like those of `pow` in Figure 1.

In static analysis, a standard technique to implement context-sensitive analysis is through function summaries and syntactical instantiation. There are also techniques based on the so-called context-free language reachability problem [23]. However, these techniques usually handle simpler problems, namely atomic label flow problems [14, 22], and are not directly applicable for unit types. We thus adopt the approach of function summaries and instantiation. For example, the summary of `square` is $\{\delta_1 * \delta_1 = \delta_2\}$. It may be instantiated as $\{\delta_{1\_1} * \delta_{1\_1} = \delta_{2\_1}\}$ at call site (1), and $\{\delta_{1\_2} * \delta_{1\_2} = \delta_{2\_2}\}$ at call site (2). These two sets of instantiated constraints are merged and become part of the summary of the function containing the calls to `square`.

Although it can be done, instantiated variables would not be instantiated again in Osprey to preserve scalability. For example, consider the following simple functions:

```
double bar (u double a) { return square(a); }
double foo ($m double b) { return bar(b); }
double hoo ($s double c) { return bar(c); }
```

where `a` is annotated with a unit variable u, and `b` and `c` are respectively annotated with $meter$ (`$m`) and $second$ (`$s`). The summaries for the functions are given below:

$$\texttt{bar} = \{\delta_{1\_\texttt{bar}} * \delta_{1\_\texttt{bar}} = \delta_{2\_\texttt{bar}}, \delta_{1\_\texttt{bar}} = u\}$$
$$\texttt{foo} = \{\delta_{1\_\texttt{bar}} * \delta_{1\_\texttt{bar}} = \delta_{2\_\texttt{bar}}, \delta_{1\_\texttt{bar}} = u_{\texttt{foo}} = \texttt{\$m}\}$$
$$\texttt{hoo} = \{\delta_{1\_\texttt{bar}} * \delta_{1\_\texttt{bar}} = \delta_{2\_\texttt{bar}}, \delta_{1\_\texttt{bar}} = u_{\texttt{hoo}} = \texttt{\$s}\}$$

One can see that a false alarm will occur due to the unit flow from `b` ($meter$) to `c` ($second$) via δ_{1_bar}. Ideally, the $\delta_{1\_\texttt{bar}}$ and $\delta_{2\_\texttt{bar}}$ in the latter two summaries should be instantiated again to avoid such false positives, but such *multi-level instantiation* requires analysis based on call graphs and is computationally expensive. Thus, we restrict our implementation to one-level syntactical instantiation to support *leaf polymorphism* only. Such a restriction is a simple and sound approximation of full polymorphism. It also offers good precision in practice as we perform the experiments in the paper.

4.4 Constraint Resolution

Our system may not discover any unit errors when there are no sufficient unit annotations in programs. For example, if there were no annotations in Figure 1, Osprey can obviously find a solution for the unit constraints of the program, *e.g.*, by assigning $unity$ to all unit variables, and it would have missed the errors. We deem this a usability problem and do the following to mitigate the problem: although not always true, when there are no enough annotations, the generated linear system will have infinite number of solutions; in such cases, Osprey issues a warning to tell users that how many additional annotations are needed to make the solution unique, while the number is the difference between the numbers of unit variables and unit constraints during the GE phase.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate Osprey in terms of scalability (Section 5.2), precision (Section 5.3), and usability (Section 5.4).

5.1 Test Programs and Results

We have run Osprey on various test programs, including computational physics and mechanical engineering applications, open source projects, and some large artificial programs to stress test its performance. The test programs are shown in Table 1. For each program, we show lines of code (for both source and preprocessed), annotation burden (ratio of number of annotations (*i.e.*, number of '$') over number of tokens in the original program (counted using the `wc` program)), time and space cost by Osprey, and the number of unit variables and constraints generated. Columns labeled "U/F" or "GE" show the costs for the union/find step or the Gaussian Elimination step during constraint solving. Currently we perform the experiments on a file-by-file basis. When there are multiple files in a program, the memory consumption and the number of unit variables and constraints are taken as the maximum across all files in the program. For other data, we take the sum over all files of the program being analyzed. All our experiments were done on a machine with a 2GHz Intel Xeon and 1GB RAM (2GB virtual memory), running Linux kernel 2.6.12.

We give some details on the programs: (1) `ex18.c` and `big*.c` are test cases from C-UNITS [24]; [3] (2) `fe.c` comes from SIUNITS [4]; (3) `coil02`, `ghostscript`, and `gnuplot` are open source projects (`coil.c` is the main part of `coil02`, an electrical inductance calculator); and (4) The rest of the programs are part of the `Ch` mechanism toolkit [25], a set of linking libraries used for developing kinematic analysis or synthesis algorithms, written in `Ch`, a superset of `C` with classes in `C++`. We manually transform the `Ch` code to `C` because Osprey currently does not support `C++`. The `big*.c` programs are large artificial single-file programs involving many arithmetic operations on units. Although they do not produce physically meaningful results, they are useful in evaluating Osprey's scalability. We use `ghostscript` and `gnuplot` for the same purpose; they have only a few computations involving units, and we analyze them with no annotations and treat all functions as polymorphic.

5.2 Scalability

Table 1 shows that Osprey is efficient and scales to large programs with hundreds of thousands of lines of code. Because our current experiments are file-by-file, the single `big*.c` files are artificial worst-case scenarios in terms of numbers of unit variables and constraints. The GE phase is currently the bottleneck of Osprey because of its quadratic space complexity. If a program generates many constraints of the form $u_1 = u_2 * u_3$ and they cannot be reduced by U/F, Osprey may not be able to solve them. For example, `big4.c` contains hundreds of thousands of expressions of the form `x=a*b`, and Osprey fails during the GE phase. However, we believe such situations rarely happen in practice; also the data for `ghostscript` and `gnuplot` justify Osprey's scalability on real code. As future work, we plan to incorporate sparse linear solvers to further improve Osprey's scalability.

U/F is a key technique to make Osprey scale. It gives orders of magnitude reduction in numbers of unit variables and constraints.

[3] The `big*.c` programs are slightly modified from test programs in a distribution of C-UNITS. C-UNITS also has other examples besides `ex18.c`. Osprey validates their units, and we do not include them here.

Name	Lines Of Code		Anno-	Time Cost (s)			Peak Memory		Peak Number of			
							Usage (MB)		Unit Vars		Constraints	
	Source	Prepro- cessed	tation Burden	Gene- ration	Solving U/F	GE	Gen	Solving	U/F	GE	U/F	GE
ex18.c	18	17	6/62	0.001	0.001	0.00	36.7	77.8	29	0	50	0
fe.c	29	23	12/107	0.005	0.003	0.00	36.8	77.8	67	0	156	0
coil.c	482	398	12/1492	0.025	0.019	0.002	38.7	78.0	512	24	859	15
gearedfivebar	667	1120	62/2234	0.100	0.044	0.003	43.7	78.7	1594	23	2720	26
crankslider	829	1071	105/3299	0.093	0.041	0.001	42.9	78.5	1419	4	2424	2
fourbar	3107	3166	264/10021	0.300	0.225	0.011	53.0	82.0	5637	39	10741	63
sixbars	4240	6564	331/13762	0.627	0.527	0.055	69.0	86.1	11150	139	21772	168
big0.c	2995	2705	0	0.190	0.254	0.00	49.8	81.8	4207	0	10510	0
big1.c	13017	11705	2/63716	0.936	1.60	0.00	88.3	93.3	18207	0	39009	0
big2.c	106985	96611	0	15.4	32.3	0.00	460.4	206.3	150283	0	322027	0
big3.c	499999	449384	1/2446636	235.2	733.0	0.00	1990.3	653.0	699041	0	1497939	0
big4.c	122886	122890	0	23.5	207.3	failed	752.3	failed	294921	135172	614411	135169
gnuplot	73366	348978	0	13.677	5.199	1.884	82.5	83.8	10149	494	15993	471
ghostscript	404669	2368515	0	165.8	24.3	8.1	154.8	116.0	53357	1291	107991	874

Table 1: Experimental Results.

We also observe that the number of annotations has significant impact on analysis performance: the more annotations, the more unit variables and constraints that may be reduced by U/F. This suggests that adding more unit annotations is better not only for debugging a program but also for improving scalability of the tool.

5.3 Precision and Errors Found

Osprey discovered two unknown errors in real applications and three in test code from other tools (one is an unknown error missed by other tools). We explain the three unknown errors:

Error 1 Here is the code fragment in `gearedfivebar` to calculate the coupler curve of a geared five bar, a term used in mechanical engineering:

```
theta = linkLength / ( 1+lamda );
...
couplerPointPos(couplerLink, theta, ...);
```

where `theta` is of $radian$,[4] `linkLength` is of $meter$, and `lamda` is unitless. It is interesting that the code passed developers' tests because the computed value of `theta` is close to the actual value and gives almost meaningful results. Developers of the `Ch` mechanism toolkit have confirmed that it is a real error, in particular, a misuse of a mechanical formula.

Error 2 This error is caused by misusing programming interface. The following code computes forces and torques of a crank slider, another mechanical engineering term:

```
double angularAccel(double theta2,
     double omega2, double theta3,
     double omega3, double alpha2);
...
int forceTorques(...) {
   ...
   angularAccel(theta2, theta3,
                omega2, omega3, alpha2);
   ...
}
```

where `theta2` and `theta3` (both parameters and arguments) are annotated as $radian$, `omega2` and `omega3` as $radian*second^{-1}$, and `alpha2` as $radian*second^{-2}$. At the function call site, `omega2` and `theta3` are passed in the wrong order. This error has also been confirmed by the developers. It is due to their misunderstanding of the programming interface. The developers fed random values to these parameters during testing and missed the bug.

Error 3 This error is caused by using the wrong unit factor. Here is a fragment of the (annotated) code in `ex18.c`:

```
$meter double mile2meter($mile double x) {
   return ( x*($f)1682 );
}
```

Osprey issues an error that the unit of the return value ($meter$) does not match the unit of `x*1682` ($meter$*1.045). Indeed, the unit factor for converting mile to meter is around 1609.344, but the code above uses 1682 instead. The ability to discover incorrect unit conversion factors is a distinctive feature of Osprey. To the best of our knowledge, no other tool has this capability.

Table 2 summarizes the numbers of errors reported by Osprey. The redundant reports are chain reactions to other kinds of reports and can be eliminated if others are eliminated. We see that Osprey is precise with low false positives. Section 5.4.2 discusses more details about the errors.

5.4 Usability

We now discuss how easy it is to use Osprey in terms of annotation burden and effectiveness of error reporting.

5.4.1 Annotation Burden

Osprey requires simple unit annotations in the form of type qualifiers, and does not require annotations for all variables in a program because of the flexibility offered by our unit type inference algorithm. Of course, users can use Osprey with no annotations at all, just as what we have done for `big{0,2}.c`, `ghostscript`, and `gnuplot`. The annotation burdens for the test programs range from 2% to 11% with larger programs having lower ratios.

To further reduce annotation burden, Osprey can suggest "critical" variables for users to annotate. This is based on the U/F step that can group variables with the same unit together, thus only selected representatives from each group need to be annotated.

Although Osprey requires few annotations, more annotations are always desirable. The more annotations, the more potential unit errors Osprey can discover. More annotations are also helpful to discover errors in the annotations themselves because of added redundancy. Thus, although not necessary, we advocate annotating

[4] In fact, $radian$ is equivalent to $unity$, and $1\ degree = \frac{\pi}{180}\ radian$.

Program Name	Number of Error Reports	Real Errors	Redundant Errors	False Errors	Imprecise Model
ex18.c	1	1	0	0	0
fe.c	2	2	0	0	0
coil.c	11	0	8	3	1
crankslider	5	1	0	4	1
fourbar	10	0	7	3	1
gearedfivebar	6	2	4	0	1
sixbars	16	0	12	4	1

Table 2: Error reports for test programs.

Kind	Sample Code (Number of Errors)
Unit Mismatch	fe.c (2), crankslider (1), gearedfivebar (2)
Factor Mismatch	ex18.c (1)
Programming Style	coil.c (3), crankslider (4), fourbar (3), sixbars (4)
Inherent Error	pow (1)
Erroneous Annotation	N/A
Imprecise Model	coil.c (1), crankslider (1), fourbar (1), gearedfivebar (1), sixbars (1)
Warning	N/A

Table 3: Classification of Error Reports.

as many program objects as possible when using Osprey. In addition, unit annotations, similar to data types of program variables, are relatively stable to program re-organization: structural changes will not require annotation changes as long as data in the program have the same semantics. Thus, annotations for legacy code can be reused because retrofitting legacy code usually changes program organization, not the algorithmic/data aspects of the program.

5.4.2 Error Reporting

Whenever a unit error is found during the constraint solving phase, unit variables and constraints involved in the error are reported. Following the naming convention of variables in our implementation, users can easily locate the positions of the variables in the original program. Osprey can discover the positions where errors emerge, but generally cannot pinpoint the origins of the errors because it currently does not trace the flows of units or row operations in GE. Thus, we heuristically pick several possible variables for the error reports: (1) During the U/F phase, variables whose units are inconsistent with the units of their representatives; (2) During the backward substitution stage in GE, variables in a row directly causing the linear system to be unsolvable. These are hints for users to find the real origins of a particular unit error.

Table 3 classifies different kinds of error reports:

Unit Mismatch This category covers all unit errors that can be discovered by manually checking whether the units on the two sides of an equation are the same or not. This kind of unit errors may be caused by erroneous formulae in programs, passing erroneous data into programs, among others. These are real errors.

Factor Mismatch Erroneous unit factors cause this kind of errors. Such errors may be caused by careless computation or programming. It is an advantage of Osprey that it detects this kind of errors.

Programming Style This kind of errors is caused by violations of the basic assumptions of standard type systems. Some intermediary variables may be used several times, taking on different units at each different use. Such errors do not affect computational results, but are considered bad programming style and error-prone. This is analogous to using an integer as a character, pointer, and etc. at the same time in C programs. Test programs `coil.c` and `sixbars` contain such errors, and we classify such error reports as false positives. Another bad programming style is to store values of different units in the same array. Our system reports errors for such cases because all elements in an array are considered having the same unit, similar to standard type systems. Two programs, `crankslider` and `fourbar`, contain such code, and we also classify these error reports as false positives.

Inherent Error Due to the abelian group nature of dimensions and the undecidability of general properties, whenever a unit multiplication occurs in a potentially unbound loop, our system cannot determine the exact unit and may issue a false alarm. For example, the unit of `x` in the following code is difficult to determine statically and will lead to a false alarm:

```
$m double x = input;
for (i=0; i<unknownBound; i++)
   x *= x;
```

When the loop bound can be statically determined, such false positives can be prevented with unrolling the loop and using different instances for `x`.

Erroneous Annotation Similar to any other forms of program annotations, user-provided unit annotations can be inconsistent. Our system is able to discover such inconsistencies. For example,

```
$radian double x;
...
x = ($degree)180;
```

Osprey reports that `x` is assigned a unit ($degree$) different from its previously assigned unit ($radian$).[5]

Imprecise Model To reduce confusion and improve Osprey's compatibility with different C dialects, we adopt a more strict semantic model of C in our implementation. For example, Osprey does not allow a structure, a variable, or a function to share the same name. Such a strategy may cause additional false positives, but did not in our experiments.

Some code may require more precise analysis techniques, such as *path-sensitive analysis* (the ability to distinguish different program paths) or *instant-level field-sensitive analysis* (the ability to distinguish different instances of the same structure). For example, `coil02` and the `Ch` toolkit use particular flags to determine the units of variables:

```
if ( flag )
   x = a*F; /* foot to meter */
else  x = a;
```

where `F` is the unit conversion factor from $foot$ to $meter$.

Our system does not support this style and would issue false errors. A path-sensitive analysis, such as the one by Das *et al.* [7], may be incorporated to improve our system. But it remains to be seen whether such enhancements are worthwhile with respect to the added complexity. In addition, such false alarms can also be classified as a bad programming style because unit types of program variables should not change.

[5]Because Osprey uses $unity$ for all un-annotated numerical constants by default, it may miss the error if `x` or `180` is not annotated. We believe this is a usability problem, and the users benefit more from such a system by providing more annotations.

Warning Osprey needs to handle floating point numbers, *e.g.*, to compute unit factors. Thus, computational imprecision is a potential problem. Nuances among unit exponent vectors and unit factors may be discarded and cause two different units to be considered equal, or vice versa. We are careful about the number of significant digits during computations and always apply traditional safe comparisons between floating point numbers, trying to have accurate results within the limitation imposed on the internal representations of floating point numbers. We did not observe any issues due to this kind of computational imprecision in our experiments. Such an implementation issue may make Osprey miss certain errors, but this has no effect on the soundness of the underlying type system.

6. POSSIBLE SYSTEM ENHANCEMENTS

6.1 Other Dimensions and Units

Most units can be represented using exponents and one factor, but some units cannot, *e.g.*, $Fahrenheit$ and $Celsius$ for degrees. To convert $Fahrenheit$ to $Kelvin$, we need more than one factor:

$$Kelvin = (Fahrenheit - 32) \times \frac{5}{9} + 273.16,$$

A possible approach to address such units is to use pre-defined unit conversion functions. For example, we may define the following function to convert $Fahrenheit$ to $Kelvin$:

```
$kelvin double f2k($fahrenheit double f) { ... }
```

The type system checks that these functions are called correctly with arguments of expected units. In addition, the correctness of such functions needs to be verified manually. This may not be an issue because these functions are generally quite simple and can be verified once and provided as library functions.

There are other models of dimensions and units, such as the *relativistic* model, the *high-energy* model, the *quantum* model, or the *natural* model [4]. There are also other base dimensions and units outside of physics, such as bit in electronics and $dollar$ in economics that our system does not model currently. We believe it is straightforward to integrate these dimensions and units into the current system and make it more widely applicable.

6.2 Dimension- vs. Unit-Level Analysis

Dimensional analysis may be carried out at two levels: the *unit level* and the *dimension level*. At the unit level, two quantities are considered to be unit consistent if and only if their units are exactly the same (including factors). At the dimension level, two quantities are unit consistent if and only if their dimensions are the same. Unit-level analysis is useful for detecting unit errors, including errors caused by wrong unit factors. However, dimension-level analysis may be more convenient to use. Programmers may prefer mixing data in different units and having the system automatically convert data to appropriate units when necessary. Then they do not need to manually supply unit conversion factors.

To support dimension level analysis, extra mechanism is required to enforce the unit correctness of programs. For example,

```
$meter float X;   $foot float Y;   X = Y;
```

The code is incorrect at the unit level, while the dimension level may consider it correct and must guarantee the correctness of the computation. One natural approach is to support automatic unit conversion through program transformation. For example, `X=Y` should be automatically transformed to `X=0.3048*Y` because $1\ foot = 0.3048\ meter$. Two issues arise.

One is about usability of such a system. When we see an assignment such as `X=0.3048*Y`, should the assignment be transformed or not? It depends on the meaning of `0.3048`. Perhaps the user intends to convert `Y` from $foot$ to $meter$ via the assignment, or `0.3048` is just an arbitrary constant. This confusion can be avoided by enforcing necessary programming conventions to decide when automatic transformation is expected. For example, one may require that no unit factors should be used by users and transformation is always performed when inconsistent units are encountered.

The other is how to determine the unit factors for program transformation. We cannot specify all the infinite number of unit factors statically. Here is one flexible, but perhaps not efficient approach: (1) Attach a unit factor variable to each expression in programs, such as `X=($f)`f`*Y`; (2) Perform the unit-level analysis and compute solutions for f; (3) Use a solution for f as the unit factor to transform programs.

Our current implementation is at the unit level only. It would be interesting to also incorporate dimension level analysis into Osprey.

6.3 Implementation Enhancements

Our current implementation of Osprey works on C code. Since many scientific applications are written in C++, it will be interesting to extend Osprey to support this language. We are collaborating with scientists at the Lawrence Livermore National Laboratory (LLNL) to implement a C++ version of Osprey based on their *ROSE* compiler framework [21]. We plan to apply our system to the large code base of scientific software at LLNL. Because the linear systems generated in Osprey are usually very sparse, we also plan to leverage LLNL researchers' expertise in sparse linear solvers to address the performance bottleneck in Osprey.

To improve the usability of the system, it will be useful to show users of Osprey not only *where* unit inconsistencies happen, but also *how* they have happened. One possibility is to record how units flow during constraint solving and display this information visually to the user together with the source code, similar to CQual's PAM mode [10]–a generic interface for marking up programs in emacs.

Finally, adding whole program analysis support may be useful, especially for programs with many cooperating modules. A straightforward approach is to merge unit constraints for each individual module and solve the complete constraint system altogether. This naïve approach is unlikely to scale. However, because many of the constraints are unrelated, one possible solution is to separate them into independent groups and solve each group individually.

7. RELATED WORK

In this section, we survey related work. Many approaches have been developed to perform automatic dimensional analysis. One common approach is via type system enhancements. Wand and O'Keefe [26] add dimensions and dimension variables to the simply-typed lambda calculus, and employ a unification-based algorithm to find the most general dimensions for every typable dimension-preserving term. Kennedy's dimension types [16] are designed for ML-style languages. He extends the standard ML type system with polymorphic dimension types and presents a unification-based algorithm to infer dimension types.

Our system follows the same approach and leverages ideas developed in these studies. There are, however, some key differences. First, our system considers both dimensions and units and deals with unit factors and interactions among different units, while they only consider dimensions. Second, we have a prototype implementation for C, a popular language for programming scientific applications, and theirs are for functional languages. Also, we use novel techniques to make our system scalable to large programs.

Our system is also related to CQual [8], a general framework for adding type qualifiers to C. The framework models the flow of qualifiers through a program using subtyping and type inference. However, standard type qualifiers are not expressive enough to model the abelian group nature of dimensions and units.

There are also unit inference and checking systems for other languages, such as Xelda [3] for Excel. Xelda uses unit transformers and constraint generators for Excel functions to infer units of formulae in a bottom-up fashion, propagating units from value cells (cells containing a number) to formula cells using cell references. The transformers and generators are analogous to our unit constraint generation rules, but they have not addressed user-defined data structures and functions in general purpose languages, and substantive effort may be required to design transformers and generators for all functions in Excel. Also, Xelda does not validate the correctness of unit conversion factors, although it supports unit coercions if users provide the factors; while our system does.

Besides of type system enhancements, other forms of language extensions have also been considered. The idea of *meta-classes* is one of these. Specific types are defined in the original programming languages to denote dimensions, and specific operations are defined to represent the arithmetic nature of dimensions. Unit inference or checking is done by the original type systems of the underlying programming languages. Such extensions are usually provided as additional libraries for the original languages, such as SIUNITS [4] for `C++` based on `STL`, MetaGen for Java based on MixGen [1]—a Java extension, Keller's Library [15] for Eiffel, Hilfinger's package [11] for Ada, and Novak's system [19] for GLisp—an extension of Lisp. This idea is feasible as long as the original programming language supports user-defined types. Although it may provide a tighter integration of dimensions and units into the original language, but it is not as flexible and requires significant changes to programming style and re-design of legacy code.

Another common approach is to validate unit correctness at runtime. Cunis [6] incorporates unit information into data objects at runtime for unit checking. C-UNITS [24] is based on a framework for program specification and verification—*Maude* [5]. The algebraic semantics of C is partially implemented in the framework, and unit information is provided as annotations by users. Unit correctness is checked when programs are simulated in the framework. The assume/assert-based specification approach requires heavier annotations and is difficult to scale to large programs. We believe a type system-based approach is more appropriate for unit checking because type systems are easier to use and more scalable.

8. CONCLUSIONS

We have presented a type system and implemented a prototype tool Osprey for validating unit correctness of C programs. The system is constraint-based and incorporates novel techniques to be scalable, precise and usable. We have extensively evaluated Osprey. It has discovered unknown errors in mature code. It is precise with few false positives, and all of which can be easily classified. It is efficient and scales to large programs with hundreds of thousands of lines of code. It is also easy to use, requiring only lightweight unit annotations, and is fully automatic. We believe that Osprey is a practical tool for improving quality of scientific software, and we are actively pursuing opportunities to improve the tool and apply it on additional production code.

Acknowledgments

We would like to thank Harry Cheng and Yu-Cheng Chou for providing the source code of the Ch mechanism toolkit and helping us evaluate Osprey. We are also grateful to Alex Aiken, Earl Barr, Prem Devanbu, Tom Epperly, Jeff Foster, Shriram Krishnamurthi, Ben Liblit, Ghassan Misherghi, Dan Quinlan, Matt Roper, Paul Steckler, and Bronis Supinski for interesting discussions and useful feedback on drafts of this paper. Finally, we thank the ICSE anonymous reviewers for their thoughtful comments on this paper.

9. REFERENCES

[1] E. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele Jr. Object-oriented units of measurement. In *OOPSLA*, pages 384–403, 2004.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, third edition, 1999.

[3] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *ICSE*, pages 439–448, 2004.

[4] W. E. Brown. Applied template meta-programming in SIUNITS: the library of unit-based computation, 2001.

[5] UIUC. *Maude*. `http://maude.cs.uiuc.edu/`.

[6] R. Cunis. A package for handling units of measure in lisp. *Lisp Symb. Comput.*, V(2):27–34, 1992.

[7] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[8] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, 1999.

[9] H. Hanche-Olsen. *Buckingham's* Π *Theorem*. `http://www.math.ntnu.no/~hanche/notes/buckingham/buckingham-a5.pdf`, 2004.

[10] C. Harrelson. *Program Analysis Mode (PAM) for Emacs*. `http://www.cs.berkeley.edu/~chrishtr/pam/`.

[11] P. N. Hilfinger. An Ada package for dimensional analysis. *ACM Trans. Program. Lang. Syst.*, 10(2):189–203, 1988.

[12] *International Systems of Units (SI)*. `http://physics.nist.gov/cuu/Units/`.

[13] L. Jiang and Z. Su. Osprey: A practical type system for validating unit correctness of C programs. `http://www.cs.ucdavis.edu/~su/unitfull.pdf`, 2005.

[14] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

[15] M. Keller. *Eiffel Library for Units of Measurement*. `http://se.inf.ethz.ch/projects/markus_keller/EiffelUnits.html`.

[16] A. Kennedy. Dimension types. In *ESOP*, pages 348–362, 1994.

[17] J. Kodumal. *Banshee–A toolkit for building constraint-based analysis*. `http://banshee.sourceforge.net/`.

[18] *Mars Climate Orbiter Mishap Investigation*. `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`.

[19] G. S. Novak. Conversion of units of measurement. *IEEE Trans. Softw. Eng.*, 21(8):651–661, 1995.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes*. Cambridge University Press, `http://www.nr.com/`, 2002.

[21] D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi. *ROSE: A Compiler Framework*. `http://www.llnl.gov/CASC/rose/`, release soon.

[22] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[23] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[24] G. Rosu and F. Chen. Certifying measurement unit safety policy. *ASE*, 2003.

[25] SoftIntegration©. *Ch User's Guide and Ch Mechanism Toolkit User's Guide*. `http://www.softintegration.com/`.

[26] M. Wand and P. O'Keefe. Automatic dimensional inference. In *Computational Logic–Essays in Honor of Alan Robinson*, pages 479–483, 1991.