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
{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 check for 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 Osprey, a *sound* type system to *automatically* check for all potential errors involving units of measurements. Our type system 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 a prototype of Osprey for C programs and evaluated it on various test programs, including computational physics or 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.

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 [17]. 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.

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 arithmetic operations, 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 a concrete example, shown 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 sample code computes an electron's final energy using the following 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 present *Osprey*, a type system for automatic dimensional analysis of C programs. 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 *abelian group*.[2] We thus need more powerful algorithms to perform type checking for unit correctness. Our

ICSE 2006 Shanghai, China, May 20–28, 2006

[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 $unitless$ [24].

[2] An abelian group is a finite or infinite set of elements together with

```
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.

type checking algorithm combines both standard type checking and Gaussian Elimination methods to validate units. As a novel contribution, Osprey 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 designed and implemented Osprey meeting this goal. Osprey is sound: if it does not find any 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 or mechanical engineering applications. Osprey discovered unknown errors in mature code. It is also precise with few false positives. Osprey is efficient and scales to large programs with hundreds of thousands of lines of code. Osprey 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 Osprey (Section 2). We then present components of Osprey (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).

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.

2.1 Users' View of Osprey

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

The system 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 of Osprey

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* [16]. 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 constraint variables and constraints. The result is subsequently fed to a *Gaussian Elimination* (GE) engine (Section 3.4). During this solving phase, whenever a unit error is discovered, an error report will be issued to inform the user of the error.

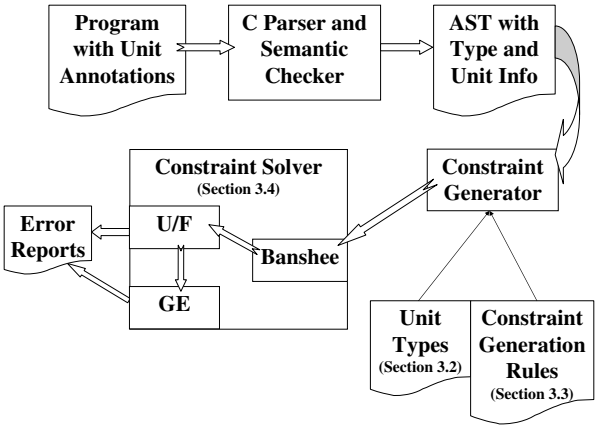

Figure 3: Tool developers' view of Osprey.

3. TYPE SYSTEM FOR UNIT CHECKING

In this section, we present details of the components in Osprey and illustrate with the example in Figure 1.

3.1 Dimensions and Units

Before introducing our type system, we first discuss properties of dimensions and units. Every dimension can be derived from the seven base dimensions in the International System of Units (SI) [11]. 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*. Unit prefixes in SI, such as $kilo$ and $milli$, are used to derive units and they are viewed as unit factors with respect to base units.

3.2 Unit Types

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

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

The abelian group for units is defined by the grammar for ut. The first seven elements are the seven base units, identity is $unity$, multiplication is denoted by the symbol $*$, and $\mathtt{ut}^{-1}$ gives the inverse element of ut. The symbol f denotes a unit factor for converting one unit to another and validating the correctness of user-provided factors (*cf.* Section 5.3). 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 in C 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, we illustrate constraint generation with the sample code in Figure 1. Interested readers can find a formal description in the full paper [12].

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 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

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.

that we also need to instantiate the set of constraints for the function body, usually referred to as a *function summary* and generated according to the code in the body. The combination of function summaries and syntactical instantiation enables performing interprocedural 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*25.4;
```

Such a program may produce physically meaningful results if `25.4` is used as a unit factor for converting $inch$ to $millimeter$. For the sake of automatic unit checking, Osprey needs to know whether `25.4` is such a unit factor or just an arbitrary constant. Therefore, the user needs to annotate `25.4` as a unit factor. In Osprey, we use `$f` to indicate that the subsequent constant is a conversion factor, as shown in `mm = inch*($f)25.4`. 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 [16] 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 the constraints 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 to resolve them. Wand and O'Keefe [24] 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 implemented the algorithm for two reasons: the general GE algorithm is cubic time and incapable of reporting why a linear system is unsolvable.

In order to have a more usable system, we believe the general technique for solving linear equations is necessary. We adapt Antoniu and Steckler's technique, exploit a union/find algorithm to reduce the number of unit variables and constraints, re-program the linear system solver in the linear algebra package *CLAPACK* [2], and utilize 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 corresponding 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 $kg * m^{-1} = u_2$. As an another example, $u_4 * u_4 = u_4$ 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 standard 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, `u_mm = u_inch*25.4` can be transformed to the following linear equations:

$$\begin{aligned}
\texttt{u_mm}_{meter} - \texttt{u_inch}_{meter} &= 0 && (1)\\
\texttt{u_mm}_{kilogram} - \texttt{u_inch}_{kilogram} &= 0 && (2)\\
\texttt{u_mm}_{second} - \texttt{u_inch}_{second} &= 0 && (3)\\
\texttt{u_mm}_{ampere} - \texttt{u_inch}_{ampere} &= 0 && (4)\\
\texttt{u_mm}_{kelvin} - \texttt{u_inch}_{kelvin} &= 0 && (5)\\
\texttt{u_mm}_{mole} - \texttt{u_inch}_{mole} &= 0 && (6)\\
\texttt{u_mm}_{candela} - \texttt{u_inch}_{candela} &= 0 && (7)\\
\texttt{u_mm}_{factor} - \texttt{u_inch}_{factor} &= \log_{10} 25.4 && (8)
\end{aligned}$$

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* [19]. 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* [19] 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

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' then
        C ← C \ {c}
      else if c matches '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
```

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
```

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 Osprey runs in linear time in the size of the input abstract syntax tree. Banshee solves equality constraints in linear time. Each iteration of the while loop in Figure 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, Osprey requires worst-case cubic time and quadratic space. Notice that the GE step is the bottleneck of our system and thus the U/F step is important to reduce the order of the input linear systems to GE to make Osprey more scalable.

Osprey is sound: it does not miss any unit errors. Although unit constraints are of the abelian group nature, the proof of soundness for Osprey still follows that for CQual [8] and is omitted here.

4. IMPLEMENTATION

4.1 Unit Representation

The 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 $[2, 1, -2, 0, 0, 0, 0] * 1$. With such a representation, arithmetic operations on units can be reduced to vector additions, subtractions, or comparisons. On the other hand, such a representation may be space-consuming. Cunis [6] uses a prime number-based representation: distinct small prime numbers are used to represent different base units, and each rational represents one unit. For example, the above unit can be represented as the rational $12/25 = 2^2 * 3^1 * 5^{-2}$. Such a representation is more time and space efficient than exponent vector-based approach, but it cannot represent units with non-integer exponents, *e.g.*, the unit of the square root of *energy*. We use the exponent vector-based representation in our implementation.

4.2 Unit Environment

Users provide unit annotations for physical quantities in the form of type qualifiers; the number of annotations required is usually small compared to the number of tokens in a program (*cf.* Table 1).

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 lexical constants are mapped to $unity$.

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 monotonic dummy unit variable as a ground unit to terminate the recursion ($struct(ref(\delta_{dummy}))$). This decreases the precision of our analysis and may cause false alarms.

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 the system may require side annotations to improve precision of the analysis. 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 every call site to relate the return value and the argument.

To make it easy to derive new units, Osprey also provides a concise syntax for defining new units, unit aliases, unit prefixes and abbreviations based on the seven base units. Not all units specified in SI are defined in the system yet, but we expect no technical difficulty in doing so.

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 constraint instantiation. There are also techniques based on the so-called context-free language reachability problem [21]. However, these techniques usually handle simpler problems, namely atomic label flow problems [13, 20], and are not directly applicable for unit types. We thus adopt the approach of function summaries and constraint 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 instantiated constraints are merged and become part of the summary of the function containing calls to `square`.

Although it can be done, instantiated variables in our implementation would not be instantiated again to preserve scalability of Osprey. 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 simple and preserves the soundness of Osprey. It also offers good precision in practice as we will show later in the paper.

4.4 Constraint Resolution

Osprey 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, most of these cases correspond to situations where the generated linear system has infinite number of solutions; therefore, Osprey issues an annotation warning to tell users that how many additional annotations are needed to make the solution unique when infinite number of solutions are detected during GE. Osprey reports the number as the difference between the numbers of unit variables and unit constraints.

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 the performance of the tool. The test programs are shown in Table 1; they all involve units. For each program, we show lines of code (for both source and preprocessed), annotation burden (measured as number of annotations over number of tokens in the original program), time and space complexity of running our analysis, and the number of unit variables and constraints generated. Columns labeled "U/F" and "GE" show the appropriate costs for the union/find step and the Gaussian Elimination step during constraint solving respectively. Our analysis is modular, and 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 a test 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 more details on the programs: (1) `ex18.c` and `big*.c` are test cases used in C-UNITS [22]; [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 [23], 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` to apply Osprey 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 the scalability of Osprey. We use `ghostscript` and `gnuplot` for the same purpose. Because they have only a few computations involving units, 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

[3]The `big*.c` programs are slightly modified from test programs contained in a distribution of C-UNITS. There are also some smaller examples besides `ex18.c` that we analyzed. Our tool verified these as unit correct, so we do not include them here.

Name	Lines Of Code		Anno-	Time Cost (s)			Peak Memory		Peak Number of			
		Prepro-	tation	Gene-	Solving		Usage (MB)		Unit Vars		Constraints	
	Source	cessed	Burden	ration	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	874	107991	1291

Table 1: Experimental results.

analysis is modular, the single `big*.c` files are artificial worst-case scenarios for Osprey in terms of numbers of unit variables and constraints. The GE phase is currently the bottleneck of the whole system because of its quardratic space complexity. If a program contains many constraints of the form $u_1 = u_2 * u_3$ and they cannot be reduced in the U/F phase, our system may not be able to solve them. For example, `big4.c` constains hundreds of thousands of expressions of the form `x=a*b`, and our system fails during the GE phase when checking this program. First, we believe such situations rarely happen in practice. Second, as furture work, we plan to incoporate sparse linear solvers to further improve our system's scalability.

The data for `ghostscript` and `gnuplot` should be the norm. We see that our U/F technique is very effective, giving orders of magnitude reduction in terms of number of unit variables and constraints: it is a key technique to make Osprey scale.

We also see that the number of annotations makes significant difference: the more annotations, the more unit variables and constraints that may be reduced by U/F. This implies that adding more unit annotations into programs is good for not only debugging but also analysis performance.

5.3 Precision and Errors Found

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

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 of $unity$. It is interesting that the code passed developers' tests because the computed value of `theta` is close to the actual value and gives meaningful results. Developers of the `Ch` mechanism toolkit have confirmed that it is a real error, in particular, a misuse of a mechanical formula.

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

Error 2 The second error is caused by a misunderstanding of 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 of $radian$, `omega2` and `omega3` are of $radian*second^{-1}$, and `alpha2` is of $radian*second^{-2}$. 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. This error is missed by C-UNITS [22]. The ability to discover this kind of errors 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. Combined with its soundness, the results imply that Osprey is effective both for bug detection and verification. Section 5.4.2 discusses more details about the errors.

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.

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. Furthermore, users can use Osprey even if providing no annotations at all, just as what we have done for `big{0,2}.c`, `ghostscript`, and `gnuplot`. The column "Annotation Burden" in Table 1 shows the ratios of number of annotations over number of tokens in test programs, ranging from 2% to 11% with larger programs having lower ratios.

Although Osprey requires few annotations, more unit 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 as many program objects as possible when using Osprey, just as one declares types for program variables. In addition, annotations are relatively stable to program changes: changes to a fragment of code will not require changes to annotations of other parts of the code. The burden of retrofitting legacy code using Osprey can be light because of the light and stable annotations.

5.4.2 Error Reporting

Whenever an error report is issued during the constraint solving phase, unit variables and constraints involved in this error are also reported. Following the naming convention of variables used in our implementation, users can easily locate the positions of the variables in the original program. Osprey can discover the positions where errors occur, but usually cannot pinpoint the origins of the errors because it does not backtrack the flows of units and row operations in Gaussian Elimination. We heuristically pick several possible variables for the error reports: (1) During the union/find phase, variables whose units are inconsistent with the units of their representatives are picked; (2) During the backward substitution stage in GE, variables in a row directly causing the linear system to be unsolvable are picked. These are hints for users to find the real origins of the 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 dimensional analysis. 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. Osprey 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, Osprey cannot determine the exact unit and may issue a false alarm. For example, the unit of `x` in the following code changes during the computation and cannot be known statically and thus may cause false positives:

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

When the loop bounds can be statically determined, such false positives can be eliminated.

Erroneous Annotation Osprey assumes users' annotations are correct, but sometimes programmers may accidentally write wrong annotations, illustrated by the following code:

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

As long as there are sufficient annotations, Osprey is able to detect such errors.

Imprecise Model To reduce confusion and improve Osprey's compatibility with different editions of C, 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 we did not encounter this kind of error 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 $meter$.

Osprey does not support this style of programming and would issue false errors. A path-sensitive analysis, such as the one

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.

by Das *et al.* [7], may be incorporated to improve the precision of Osprey. It remains to be seen whether such enhancements are worthwhile w.r.t. the added complexity. In addition, such false alarms can also be classified as bad programming style because the unit of a program variable should not change.

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 do not encounter any false negatives or positives caused by the computational imprecision during our experiments, but it may be a hole making our implementation unsound. However, the underlying unit type system certainly remains sound.

6. POSSIBLE SYSTEM ENHANCEMENTS

In this section, we discuss limitations of our current implementation and possible directions to enhance Osprey.

6.1 Other Dimensions and Units

Most units can be represented using exponents and unit factors, but some units cannot. $Fahrenheit$ and $Celsius$ for degrees are such examples. $Fahrenheit$ can be converted to $Kelvin$ using:

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

but not with only a single unit factor. 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 Osprey does not model currently. We believe it is straightforward to integrate these dimensions and units into Osprey 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* or the *dimension level*. At the unit level, two quantities are considered unit consistent if and only if their units are exactly the same (including factors). At the dimension level, two quantities are considered 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 be of appropriate units when necessary. They do not need to supply unit conversion factors or perform this conversion manually. Under such situations, a dimension-level analysis is more useful.

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

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

It 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 concerns usability of such a system. When we see an assignment such as `X=0.3048*Y`, should the assignment be transformed or not? This depends on the meaning of 0.3048. Perhaps the user intends to convert `Y` from $foot$ to $meter$ via the assignment, or perhaps 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, it is reasonable to require that no factors should be used by users and transformation is always performed when inconsistent units are encountered in an equation.

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`)1*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.

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 [24] 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 [15] 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.

Osprey follows the same approach and leverages ideas developed in these studies. There are, however, some key differences. First, Osprey is more general. While they only consider dimensions, we consider both dimensions and units and deal with unit factors and interactions among different units. Second, Osprey is for C, a popular language for programming scientific applications, and theirs are for functional languages. We also use novel techniques to make it scalable to large programs.

Osprey 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. Although Xelda supports unit coercions if users provide unit conversion factors, it does not validate the correctness of the factors, while Osprey 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 and used to represent 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 systems are usually provided as additional libraries for the original languages. Examples include SIUNITS [4] for `C++` based on `STL`, MetaGen for Java based on MixGen [1]—a Java extension, Keller's Eiffel Library [14] (for Eiffel), Hilfinger's Ada package [10] (for Ada), and Novak's system [18] for GLisp—an extension of Lisp, among others. This general approach 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 [22] 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 by the system. 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 Osprey, a sound type system for validating unit correctness of C programs. We used a constraint-based formulation and presented novel techniques to make the system scalable and easy to use. We have implemented our system and extensively evaluated the tool. Osprey has discovered unknown errors in mature code. It is precise with few false positives, and all false positives can be easily recognized. Osprey is efficient and scales to large programs with hundreds of thousands of lines of code. It is also easy to use, requiring only simple 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 apply our tool on additional production code. Both the NASA Jet Propulsion Laboratory (JPL) and the Lawrence Livermore National Laboratory (LLNL) have expressed interest in using Osprey to check code at the labs. We also plan to publicly release Osprey in the fall of 2005.

Acknowledgments

We would like to thank Harry Cheng and Yu-Cheng Chou for providing us with the source code of the Ch mechanism toolkit and helping us evaluate our tool. 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 on this work and useful feedback on drafts of this paper.

9. REFERENCES

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele. Object-oriented units of measurement. In *OOPSLA*, 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 metaprogramming in SIUNITS: the library of unit-based computation, 2001.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual*. `http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf`, 2004.

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

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

[12] 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.

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

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

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

[16] J. Kodumal. *Banshee–A toolkit for building constraint-based analysis*. `http://banshee.sourceforge.net/`, 2004.

[17] MCO Failure Board, `http://mars.jpl.nasa.gov/msp98/orbiter`. *Mars Climate Orbiter*.

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

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

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

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

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

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

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

APPENDIX

A. CONSTRAINT GENERATION RULES

The rules in Figure 5 focus on main expressions and declarations in C, rules for statements and other expressions are omitted here. Rules for declarations are important for initializing unit environments and constraint sets. The application of the rules depends on the structure of abstract syntax trees, and essentially a syntax-directed procedure. The evaluation order of expressions is enforced by the priority and associativity of C operators, but different order should have no effect on generated constraints.

Here are some explanations for the rules.

The first three rules are for declarations; and $U; C \vdash e : u : U_1; C_1$ means a node e in an abstract syntax tree has the unit u under the current unit environment U and the current set of dimensional constraints C, and leads to a new unit environment U_1 and a new set of constraints C_1 ($U \subseteq U_1$ and $C \subseteq C_1$ are neither always true, especially when context-sensitive analysis is used); when it no meaning to assign a unit to a node or U is not changed by the node, u or U part will be omitted. Such situations can be easily recognized based on the context of the rules. For example, $U; C \vdash u \quad \tau \quad id : U[id \rightarrow u] \cup U_\tau; C \cup C_\tau$ means that a declaration '$u \quad \tau \quad id$' enlarges the current unit environment with '$\{id \rightarrow u\} \cup U_\tau$', and enlarges the current constraint set with C_τ, where U_τ and C_τ are the unit environment and the constraint set specific to the C type τ (useful when implementing instant-level field sensitivity or context-sensitivity).

(VARDECL) describes what happens to the current unit environment U and constraint set C when declaring a new variable x with C type τ and unit annotation u_x. u_x denotes the unit of x and may depend on τ (section 4.2). The new unit environment is the union of U_τ and U substituted by $\{x \rightarrow u_x\}$; the new constraint set is the union of C_τ and C. U_τ, C_τ are the unit environment and constraint set specific to τ. They are usually empty when τ is a primitive type or a pointer; but if τ is a composite type, such as a struct and a function, U_τ may include unit information for its components which should be instantiated when τ is used, and C_τ may include constraints among these units. For example, for the 'square' function in Section 4.3 , U_{square} should be $\{square_a \rightarrow \delta_1, square_{return} \rightarrow \delta_2\}$, and C_{square} should contain $\{\delta_2 = \delta_1 * \delta_1\}$ generated by the intra-procedural analysis for the body of 'square'. Unit variables in U_τ and C_τ may be replaced by fresh variables whenever instant-level field sensitivity or context sensitivity is used.

In (STRUCTDECL), each field of a struct of n fields has its unit information inserted into the unit environment; this, together with the rule (FIELDACCESS), makes our system field-sensitive. Also, the struct itself also has its unit information ($s \rightarrow S(u_1, \ldots, u_n)$) inserted into the unit environment. C_0 refers to constraints among the units of the n fields and the struct itself. It can be used to express the dimensional relations among the fields, but usually empty.

Similar meaning can be applied to the rule (FUNCDECL) which is used for declaring a function. x_is are parameters and e represents return expressions. For a function declaration, e may represent just a unit declaration. For a function definition, the function body should be analyzed to get the a correct result for e, and thus U_{return} and C_{return} may contain unit information and constraints for the body. C_0 here refers to constraints among the n parameters and the e, for example, C_0 for the 'square' function (Section 4.3) can be $\{\delta_2 = \delta_1 * \delta_1\}$, and $C_{return} \cup C_0$ should be equal to the C_{square} mentioned above. C_0 is particular useful when the 'square' is just a declaration and the constraints between parameters and return value must be given via unit annotation. Actually, it is a practical consideration for C library functions. Also, together with (FUNCAPP), this rule can be used to handle polymorphism.

The other rules are for expressions.

In the rule (CONST) and (VAREXPR), $U(n)$ or $U(x)$ may be empty, then default units for constants or variables are used (refer to 4.2).

In the rule (TIMES), we intent to separate constraints involving abelian group nature from standard constraints, then Banshee can directly handle the standard parts; the other parts are not complicate to resolve (refer to 3.4). So, we introduce a fresh name '$dummy\_name$' and a fresh unit variable u_{12} to represent the unit of the product, and the unit environment and constraint set are changed accordingly. In section B, a naming convention for fresh variables will be described.

The rule (ADD) here is natural: the units of the operands and the result should be all same. This is reasonable for our dimensional analysis. [5]

(PLUSPLUS) treats 1 as a polymorphic constant which is reasonable. The rule (CAST) has more discussion on unit annotations for constants.

In (ASSIGN), based on Steensgaard's style analysis, an equality constraint $u_1 = u_2$ is generated.

The rule (ARRAYACCESS) assumes that all elements in an array should have a same unit. Also, the rule implies that pointers can not be used as indices. Alternatively, we may treat array accesses as dereferences: $e_1[e_2] \equiv *(e_1 + e_2)$. However, such a treatment may allow arbitrary pointer arithmetic and is inconsistent with (ADD) here and causes the flows of units over pointers broken. Thus, we choose the more restricted option in our dimensional analysis.

(FIELDACCESS) is a field-sensitive rule for constraint generation. Another form of field accesses in C: $e \rightarrow id$ is treated as $(*e).id$ and the rule for it is omitted here. Also, checking whether "id is the ith field of u" relies on the C type system in implementation.

In (FUNCAPP), U_0 and C_0 refer to the environment where f is defined, so they are based on C's semantics, and are usually different for global functions and local functions. U_{fI}, C_{fI} and u_{iI} represent instantiations of the environment affected by a polymorphic function f. For a non-polymorphic function, $U_{fI} = U_f$, $C_{fI} = C_f$ and $u_{iI} = u_i$.

Other rules are straightforward and not explained here.

B. VARIABLE NAMING CONVENTIONS

Basically, the name of a unit variable is comprised of the line number of its corresponding program object, the position index number for uniqueness and the name of the program object. Thus, if our solver says some unit variable has no solution later on, programmers can easily locate the corresponding position in source code. We hope the burden of manual locating can be automated later. The following is the summarized naming convention, each kind of names contains several components in order and separated by an underscore:

- The name of a unit variable for a lexical constant: 'u', line number, index, 'const', the serial number of the lexical constant, the serial number of the current scope;
- The name of a unit variable for a struct field: 'u', line number, index, struct name, '@', field name, the serial number of the current scope;
- The name of a unit variable for a variable: 'u', line number, index, variable name, the serial number of the current scope;
- The name of a unit variable for a polymorphic function parameter: 'u', line number, index, function name, '@', parameter name, the serial number of the current scope, the number of function calls;
- The name of a unit variable for a polymorphic function return value: 'u', line number, index, function name, '@return', the serial number of the current scope, the number of function calls;
- The name of a unit variable for an expression, such as multiplications: 'u', line number, index, the name of the left operand, 'MUL', the name of the right operand, the serial number of the current scope;
- The name of an instantiated unit variable: the name of the polymorphic unit variable, line number and index of the position of a function call site.

[5] In other domains, we may want to add quantities of different units together, such as adding 'apples' and 'oranges' [24]. A possible approach to handle such arithmetic operations is to introduce a new dimension whenever necessary and extend base dimensions with it. The constant 0 should always be polymorphic. To do so, Wand and O'Keefe use a dimension constructor *newdim* [24], and Kennedy uses a *dimension* declaration [15]. Both studies are for ML-style languages. For our system, we have not encountered situations requiring such a feature.

$$\frac{}{U; C \vdash u_x\ \tau\ x : U[x \to u_x] \cup U_\tau; C \cup C_\tau} \quad \text{(VarDecl)}$$

$$\frac{\begin{array}{c} U; C \vdash u_1\ \tau_1\ x_1 : U[x_1 \to u_1] \cup U_{\tau_1}; C \cup C_{\tau_1} \quad \ldots \\ U[_{i=1}^{n-1} x_i \to u_i] \cup_{i=1}^{n-1} U_{\tau_i}; C \cup_{i=1}^{n-1} C_{\tau_i} \vdash u_n\ \tau_n\ x_n : U[_{i=1}^{n} x_i \to u_i] \cup_{i=1}^{n} U_{\tau_i}; C \cup_{i=1}^{n} C_{\tau_i} \end{array}}{U; C \vdash s(x_1, \ldots, x_n) : U[_{i=1}^{n} x_i \to u_i] \cup_{i=1}^{n} U_{\tau_i} \cup \{s \to S(u_1, \ldots, u_n)\}; C \cup_{i=1}^{n} C_{\tau_i} \cup C_0} \quad \text{(StructDecl)}$$

$$\frac{\begin{array}{c} U; C \vdash u_1\ \tau_1\ x_1 : U[x_1 \to u_1] \cup U_{\tau_1}; C \cup C_{\tau_1} \quad \ldots \\ U[_{i=1}^{n-1} x_i \to u_i] \cup_{i=1}^{n-1} U_{\tau_i}; C \cup_{i=1}^{n-1} C_{\tau_i} \vdash u_n\ \tau_n\ x_n : U[_{i=1}^{n} x_i \to u_i] \cup_{i=1}^{n} U_{\tau_i}; \cup_{i=1}^{n} C_{\tau_i} \\ U[_{i=1}^{n} x_i \to u_i] \cup_{i=1}^{n} U_{\tau_i}; C \cup_{i=1}^{n} C_{\tau_i} \vdash e : u_{return} : U_{return}; C_{return} \end{array}}{U; C \vdash f(x_1, \ldots, x_n) \to e : U_{return} \cup \{f \to \text{lam}(u_{return}, u_1, \ldots, u_n)\}; C_{return} \cup C_0} \quad \text{(FuncDecl)}$$

$$\frac{U(n) = u_n}{U; C \vdash n : u_n; C} \quad \text{(Const)}$$

$$\frac{U(x) = u_x}{U; C \vdash x : u_x; C} \quad \text{(VarExpr)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2}{U; C \vdash e_1 * e_2 : u_{12} : U_2[dummy\_name \to u_{12}]; C_2 \cup \{u_{12} = u_1 * u_2\}} \quad \text{(Times)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2}{U; C \vdash e_1 + e_2 : u_2 : U_2; C_2 \cup \{u_1 = u_2\}} \quad \text{(Add)}$$

$$\frac{U; C \vdash e : u : U_1; C_1}{U; C \vdash e\texttt{++} : u : U_1; C_1} \quad \text{(PlusPlus)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2}{U; C \vdash e_1 < e_2 : unity : U_2; C_2 \cup \{u_1 = u_2\}} \quad \text{(LessThan)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2}{U; C \vdash e_1, e_2 : u_2 : U_2; C_2} \quad \text{(Comma)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2}{U; C \vdash e_1 = e_2 : u_2 : U_2; C_2 \cup \{u_1 = u_2\}} \quad \text{(Assign)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2 \qquad U_2; C_1 \vdash e_3 : u_3 : U_3; C_3}{U; C \vdash (e_1 ? e_2 : e_3) : u_3 : U_3; C_2 \cup C_3 \cup \{u_1 = unity, u_2 = u_3\}} \quad \text{(IfExpr)}$$

$$\frac{U; C \vdash e : u : U_1; C_1}{U; C \vdash \&e : ref(u) : U_1; C_1} \quad \text{(Addr)}$$

$$\frac{U; C \vdash e : u : U_1; C_1}{U; C \vdash *e : ref^{-1}(u) : U_1; C_1} \quad \text{(Deref)}$$

$$\frac{U; C \vdash e_1 : u_1 : U_1; C_1 \qquad U_1; C_1 \vdash e_2 : u_2 : U_2; C_2}{U; C \vdash e_1[e_2] : ref^{-1}(u_1) : U_2; C_2 \cup \{u_2 = unity\}} \quad \text{(ArrayAccess)}$$

$$\frac{U; C \vdash e : u : U_1; C_1 \qquad u = S(u_1, \ldots, u_n) \qquad id \text{ is the } i\text{th field of } u}{U; C \vdash e.id : u_i : U_1; C_1} \quad \text{(FieldAccess)}$$

$$\frac{\begin{array}{c} U; C \vdash e_1 : v_1 : U_1; C_1 \qquad \ldots \qquad U_n; C_{n-1} \vdash e_n : v_n : U_n; C_n \\ U_0; C_0 \vdash f(x_1, \ldots, x_n) \to e : u : U_f; C_f \qquad u = \text{lam}(u_0, u_1, \ldots, u_n) \end{array}}{U; C \vdash f(e_1, \ldots, e_n) : u_{0I} : U_n \cup U_{fI}; C_n \cup C_{fI} \cup \{v_i = u_{iI}, \forall i. 1 \le i \le n\}} \quad \text{(FuncApp)}$$

$$\frac{U; C \vdash e : u_1 : U_1; C_1 \qquad e \text{ is a lexical number}}{U; C \vdash (u_2)e : u_2 : U_1; C_1 \cup \{u_1 = unity\}} \quad \text{(Cast)}$$

Figure 5: Constraint generation rules.