ASE'03, IEEE, pp. 304 - 309. 2003.

Certifying Measurement Unit Safety Policy

Grigore Roşu
Department of Computer Science,
University of Illinois at Urbana-Champaign
grosu@uiuc.edu

Feng Chen
Department of Computer Science,
University of Illinois at Urbana-Champaign
fengchen@uiuc.edu

Abstract

Measurement unit safety policy checking is a topic in software analysis concerned with ensuring that programs do not violate basic principles of units of measurement. Such violations can hide significant domain-specific errors which are hard or impossible to find otherwise. Measurement unit analysis by means of automatic deduction is addressed in this paper. We draw general design principles for measurement unit certification tools and discuss our prototype for the C language, which includes both dynamic and static checkers. Our approach is based on assume/assert annotations of code, which are properly interpreted by our deduction-based tools and ignored by standard compilers. We do **not** *modify the language in order to support units. The approach can be extended to incorporate other safety policies without great efforts.*

1 Introduction and Motivation

Correctness of a software application often involves aspects beyond the syntax and semantics of the programming language(s) in which it is implemented. Detecting violations of safety policies specific to a domain of interest can reveal deep errors, which are very hard, if not impossible, to find by just analyzing programs within their language semantics. Checking software for measurement unit consistency, e.g., that one does not add or compare meters and feet, is the topic of this paper. We describe an integrated tool supporting a segment of C, which contains both dynamic and static checkers, and explain their trade-offs. The user interacts with our prototype via code annotations, which are special comments, and via safety policy violation warning reports. Although we focus on measurement unit safety here, this work falls under what call *domain-specific safety policy certification* [18], whose main idea is to axiomatize a specific domain of interest, and then to define an appropriate abstraction from programming language constructs into that abstract domain.

There has been much work on incorporating measurement units in programming languages. The earliest seems to be [3]. An intuitive approach is to enrich programming languages with measurement units. Mechanisms that allowed units to meaningfully occur in programs were suggested in [14] and [13], and support for measurement unit within existing languages, like Pascal [7, 8] and Ada [9], were also proposed. Based on the belief in [21] that type checking can and should be supported by semantic proof and theory, [22] associated numeric types with polymorphic dimension parameters, hereby avoiding measurement unit errors, and a formally verified method to add, infer and check dimension types in ML-style languages was proposed in [16, 15]. Unfortunately, these approaches have not been accepted by mainstream programmers. One reason may be the reluctance of software developers to use/learn new programming languages just to ensure unit safety. Rewriting existing programs in a new language is even more inconvenient, not to mention having to give up a favorite trusted compiler. Another important reason may be the limitation of using type checking front end interfaces to languages: programs which do not type check are rejected.

A more practical solution is to develop packages or libraries for measurement unit analysis and integrity. Reusable Ada packages are discussed in [12, 19]. A similar approach was taken by the Mission Data System (MDS) team at NASA JPL, who developed a large C++ library incorporating a few hundred classes representing typical units, like `MeterSecond`, together with appropriate methods to replace the arithmetic operators when measurement unit objects are involved. A package called "Measurement unit Analysis" [17] introduces measurement unit variables in the Computer Algebra system of *Mathematica*. These approaches avoid changing the underlying programming languages. However, they add unnecessary runtime overhead due to additional method calls, and cannot handle simple situations like the product of elements in a vector. Another problem here is how to support the legacy systems and system migration, which involves rewriting all the related programs. That is unacceptable in practice.

Most of these related work is based on type checking, defining a type system with physical units on top of a programming paradigm, and then using the compiler to catch

inconsistencies. Using code annotations instead of types, our approach does not modify the underlying languages, thus facilitating system migration and evolution, and supports highly abstract unit invariants (see Subsection 4.3), which helps one in developing portable library functions. Besides, unlike in type-based techniques, our approach is based on warnings that users can choose to ignore or not. Past efforts focus on *dimensional analysis*, whose purpose is to catch inconsistencies among dimensions, i.e., different units belonging to the same dimension are viewed as compatible, such as `meter` and `inch`, and conversions are applied automatically between different units in the same dimension. Our approach is to consider all units unrelated and to do no unit conversion automatically. Unit conversion computations can be done in different ways and involve roundoff errors, so we believe that the users should be entirely responsible for unit conversions. However, our warnings can have different levels of importance; an inconsistency involving units of different dimensions is more serious than one between units of same dimension.

2 Measurement Units in Maude

Maude [4] is a high performance specification and verification system in the OBJ family [11] that supports both equational and rewriting logics. It provides a good framework to create executable environments for different logics, theorem provers, programming languages and models of computation. We use Maude to specify the C programming language and also its extension with units of measurement. The following is a Maude specification of units of measurement, which is crucial to our application:

```
fmod BASICUNITS is
  sorts BUnit .
  ops mile kg meter second Celsius : -> BUnit .
endfm
fmod UNITS is protecting RAT . extending BASICUNITS .
  sorts SpecialUnit Unit UnitList .
  subsorts BUnit SpecialUnit < Unit < UnitList .
  ops noUnit any fail : -> SpecialUnit .
  op _^_ : Unit Rat -> Unit [prec 10] .
  op __ : Unit Unit -> Unit [assoc comm prec 15] .
  op nil : -> UnitList .
  op _,_ : UnitList UnitList -> UnitList [assoc id: nil] .
  vars U U' : Unit .  vars N M : Rat .
  eq U noUnit = U .     eq U any = U .
  eq U fail = fail .    eq fail ^ N = fail .
  eq any ^ N = any .    eq noUnit ^ N = noUnit .
  eq U ^ 0 = noUnit .   eq U ^ 1 = U .
  eq U U = U ^ 2 .      eq U (U ^ N) = U ^ (N + 1) .
  eq (U ^ N) (U ^ M) = U ^ (N + M) .
  eq (U U') ^ N = (U ^ N) (U' ^ N) .
  eq (U ^ N) ^ M = U ^ (N * M) .
endfm
```

Keywords `sort` and `op` refer to the types of data and the operations on these data. In the above modules, we have different sorts of data: `BUnit` for basic units, `SpecialUnit`, `Unit` and `UnitList`. Units like `mile` have been declared as constants of sorts `BUnit`. The unit `any` can be dynamically converted to any other unit, depending on the context; e.g., in `x+=1`, the increment 1 is interpreted to have the unit `any` and dynamically converted to the unit of `x`. The special unit `noUnit` is used to distinguish a cancelled unit (for example after calculating `meter meter^(-1)`) from `any`, in order to report appropriate warnings. The special unit `fail` is attached to a variable in case its unit cannot be computed due to safety violations. The result sort of an operation is listed after `->` and the argument sorts between `:` and `->`. e.g., the power operator `op _^_ : Unit Rat -> Unit` takes a unit and a rational number and returns another unit.

Maude allows attributes like associativity (A), commutativity (C), precedence and identity to be associated with binary operators. For example, `op _ _ : Unit Unit -> Unit` is declared AC, So Maude finds `second meter second` and `meter second^2` equivalent. In fact the above module terminates and is Church-Rosser modulo AC, thus enabling an automated deduction approach to measurement unit analysis. The equations in the above module, introduced via the keyword `eq`, define the power operator. In Maude, one module can extend other modules and inherit their sorts, operations and equations. The above unit specification is split into the `BASICUNITS` module, which defines a series of basic units, and the `UNITS` module, which defines the main operations on units together with their calculus. This separation is based on the observation that the set of basic units varies from application to application, while the calculus of units is always the same. Our prototype allows users to define their own basic unit set through a configuration file, and then generates the `BASICUNITS` module automatically. Nonstandard units, such as currency, can also be defined this way.

3 Executable Semantics of C

Equational logic is an important paradigm in computer science. It admits complete deduction and is efficiently mechanizable by rewriting: CafeOBJ [6], Maude [4] and Elan [1] are equational specification systems that can perform millions and tens of millions of rewrites per second on standard PC platforms. Goguen and Malcolm [10], Wand [23], Broy, Wirsing and Pepper [2], and many others, showed that equational logic is essentially strong enough to easily describe virtually all programming language features.

We have defined the semantics of a segment of C as an algebraic specification in Maude of about 1,000 equations. This specification supports a significant subset of the C language by carefully simulating the running environment. The `union` data type and the arithmetic computation on pointers are the only features left out. Since Maude specifications can be executed by rewriting, we were able to run dozens of non-trivial, often recursive, C programs directly within the mathematical definition of C. Equational specifications of programming languages, as well as extensions of them, usually can be developed rapidly because they just reflect a rigorous, formal definition of the language. Besides obvious advantages in programming language design,

ASE'03, IEEE, pp. 304 - 309. 2003.

a major benefit of having a language defined formally is that one can also reason formally about programs, using deductive techniques. This is what we do in this paper: we extend C's specification with support for units of measurement, and then analyze C programs both dynamically (by "executing" them with the semantics of C) and statically, by equational reasoning as implemented in Maude via term rewriting.

4 Design Conventions and Annotations

Our approach is based on annotations, which are ignored by compilers as comments, but regarded as special statements by our tool. The design of our application has been mainly influenced by three factors: correctness, unchanged programming language, and low amount of annotations.

4.1 Correctness

By "correctness" we mean that there are no possible violations of safety policy that our tool does not report. We consider correctness a crucial aspect because, unlike other tools like ESC [5] being mainly intended to *help* users find some bugs in their programs, our application is intended to be used in the context of safety critical software, such as air/space craft and navigation, where software developers want to be aware of any inconsistency in their code. Our tool is composed of a dynamic checker and a static checker; the former generates accurate warnings while the latter warns users of all possible conflicts.

4.2 Unmodified Programming Language

A major influencing factor in the design of our prototype was the decision to *not* modify the underlying programming language at all, for example by adding new types. Our reason for this decision is multiple. First, we do not want to worry about providing domain specific compilers; one can just use the state of the art optimized compilers for the language under consideration. Second, by enforcing an auxiliary typing policy on top of a programming language in order to detect unit inconsistencies via type checking, one must pay the price of some runtime overhead due to method calls that would replace all the normal arithmetic operators; our static prototype does not add any runtime overhead. Third and perhaps most importantly, we do not put the user in the unfortunate situation of having a correct program rejected because it cannot be type checked, which is in our view the major drawback of typed approaches to unit safety; instead, our users has the option to either add more auxiliary unit specific information to help the checker or to ignore some of the warning messages.

4.3 Annotation Schemas

Code annotations, which are special comments, have been proved to be very useful and necessary in the practical software development, especially in large-scale applications. Today, it is not surprising to see even more comments than instructions in a commercial program. There are many practices to develop reliable software based on assertions, such as Design By Contract (DBC) [20].

The unit-related annotations are introduced with the syntax `/*U _ U*/` and are of two kinds: assumptions and assertions. Our annotation schemas are general and can be applied to any domain-specific safety policy checker, but in this paper we focus on unit safety policy. The next is an example showing some of the complex unit expressions that can be manipulated by our tool; it also emphasizes the importance of annotations. This program has functions to calculate distances, convert energy and calculate the angle under which a projectile of a given weight and acting energy should be launched in order to travel a given distance:

```
float lb2kg(float w)
/*U assert unit(w) = lb U*/ /*U assume returns kg U*/
{ return 10 * w / 22; }
float distance(float x1, float y1, float x2, float y2)
{ return sqrt((x2-x1)^2 + (y2-y1)^2); }
float energy2speed(float energy, float weight)
{ return sqrt(2 * energy / weight); }
float projectiletan(float dist,float speed,float g)
 /*U assert unit(speed)^2 = unit(dist) unit(g) U*/
{ float dx, dy;
  dx = speed * speed + sqrt(speed^4 - (dist * g)^2);
  dy = dist * g;  return dx/dy;}
main() {
float projectilex, projectiley, targetx, targety,
      dist, projectilewight, energy, speed, g;
projectilex = 0;
/*U assume unit(projectilex) = meter U*/
projectiley = 0;
/*U assume unit(projectiley) = unit(projectilex) U*/
targetx = 17;
/*U assume unit(targetx) = unit(projectilex) U*/
targety = 21;
/*U assume unit(targety) = unit(projectiley) U*/
dist=distance(projectilex,projectiley,targetx,targety);
projectileweight = 5;
/*U assume unit(projectileweight) = lb U*/
energy = 2560;
/*U assume unit(energy) = kg meter^2 second^-2 U*/
speed = energy2speed(energy, projectileweight);
g = 10; /*U assume unit(g) = meter second^-2 U*/
printf("%f\n", projectiletan(dist, speed, g));}
```

The first function converts `lb` to `Kg`. The next one computes the distance between two points. No annotations are given, but a warning will be generated anyway if the arguments do not have the same unit. The third function computes the speed of an object, given the energy acting on it. The last function computes the tangent of the angle of a projectile, given a certain distance it wants to reach, an initial speed and a gravitational acceleration. This function is annotated with an assertion describing a unit invariant among its arguments. This allows one to use such functions in various contexts, such as under metric or English system conventions, as well as for other possible combinations of units.

The above code contains a unit safety violation, when the function `projectiletan` is called, because the unit of `speed` is `Kg^(1/2) meter second^-1 lb^(-1/2)` so the assertion is violated. To correct this problem, the user should first properly convert the projectile weight to `Kg` using the function `lb2kg`.

There are two types of assumptions supported by our application, namely `/*U assume unit(_) = _ U*/` and

`/*U assume returns _ U*/`. The first can appear anywhere in the program and takes as arguments a variable and a unit expression. For the static checker, if the variable is not a simple one then it will be automatically replaced by its simple root, e.g., `s[10][i]` will be replaced by just `s`. The unit expression can be any combination of basic units and `unit(Expr)`, the second being evaluated in the current execution environment(s). We can see some examples of assumptions in the `main` function of the above example.

The second kind of assumption annotation is used only for functions, to enforce returning a specific unit. It can be used within unit conversion functions, such as the function `lb2kg`, or simply to state the result unit of a function when it cannot or is not desired to be inferred (e.g., in the case of library functions). It is placed just before the body of the function and takes a unit expression as a sole argument, which will be evaluated before the body of the function but after the arguments of the function are instantiated. The body of the function will still be analyzed and warnings will be appropriately given, but the assumed unit will be returned and used in callee's context.

Assertions have the syntax `/*U assert _ U*/`, the argument being any boolean expression on units, using the boolean connectors `and`, `or`, `implies`, `not`, over equalities of unit expressions. Assertions can be highly unit invariant. For example, in the assertion for `projectiletan`, the variables can be represented either in the metric or in the English system. Assertions can be anywhere in a program, including just before the body of a function, as shown in the previous example. All assertions are treated the same way by the dynamic checker: the boolean expression is evaluated and a warning is reported when the result is `false`. The static checker, however, interprets the assertions in three different ways, depending on where they appear. Assertions which appear just before the body of a function, like in the above example, are used to check the consistency of the input arguments; together with return assumptions, these give the tool the ability to handle library functions. Assertions which appear within the body of a loop are treated as loop invariants (Subsection 5.2 gives more details on these). The remaining assertions are handled like the dynamic checker: they are evaluated in the corresponding environment and warnings are generated if false.

4.4 Reducing the Amount of Annotations.

Another major factor influencing our design was the overall observed and sometimes openly declared reluctance of modifying or inserting annotations in programs. Therefore, we paid special attention to reducing the amount of needed annotations. As a consequence, each variable is considered to have a default unique unit, which is different from any other existing unit. Thus, our tool will output a warning on the simple code segment `x = 10; y = 10; r=x+y` if `x` and `y` have not been assumed to have any units before.

This brings us to a major design convention, called the *locality principle*, which says that one is assumed to know and understand what one is doing locally, within a single instruction, with respect to constants. For example, if one writes `x++`, then one means to increase the value of `x` by 1, and this 1 has exactly the same unit as `x` at that particular moment during the execution of the program. There is no difference between the statements `x++` and `x = x + 1`, so we apply the same locality principle to numerical constants. Therefore, a constant assignment to a variable, such as `x = 5`, will not change the unit of `x`. Conservative users can avoid the locality principle by attaching a unit to numerical values via appropriate assumptions, e.g., `tmp = 5 ; /*U assume unit(tmp) = second U*/`, and then execute `x = x + tmp`; a warning will be reported in this case if the unit of `x` cannot be shown to be `second`.

Based on these conventions, the following sorting code needs only one assumption to satisfy the safety policy:

```
int n = 25, i, j, a[25] ; /*U assume unit(i) = any U*/
for (i = 1 ; i <= n ; i = i + 1) a[i] = n - i + 1 ;
for (i = 1 ; i < n ; i = i + 1)
  for (j = i + 1 ; j <= n ; j = j + 1)
    if (a[j] < a[i]) { temp=a[i]; a[i]=a[j]; a[j]=temp; }
```

The only assumption needed, assigning the universal unit `any` to the counter `i`, guarantees the compatibility of `i` and `n` when they are compared later, within the loop conditions. The first loop assigns the unit of `n` to each of the 25 elements of `a`. In the case of the static analyzer, the array `a` will be assigned the unit of `n` by executing the loop body symbolically only twice, regardless of the value of `n` (because the environment set stabilizes; see next section). Then the second loop is analyzed and no warning is reported because the nested loop assigns the unit of `i`, `any`, to `j`, so any subsequent comparisons of `j` are safe; the environment set also stabilizes in two iterations of the loop. Without the assumption, 5 warnings would be reported.

5 A Measurement Unit Prototype Certifier

Our prototype certifier includes both dynamic and static checkers, built on the Maude executable semantics of C, together with a formatting tool which adjusts the format of the input C programs into well-formed Maude terms, and a console program that puts all the components together, hiding the details of Maude and providing the users a friendly interface. A preprocessing tool is under development to support `include` and `define` statements in C. The architecture of our certifier is shown in Figure 1.

The interested reader is encouraged to check the URL `http://fsl.cs.uiuc.edu`, where links to a latest version of the certifier with documents.

5.1 Dynamic Checker.

Our dynamic checker essentially interprets the annotated C program within its enriched executable semantics. This is done by properly extending the executable semantics of C

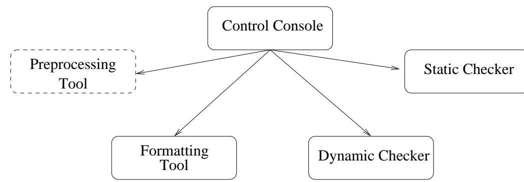

Figure 1. Architecture of a C Unit Checker

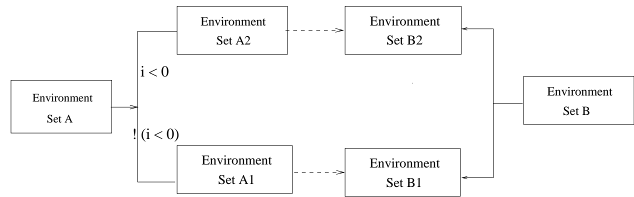

Figure 2. Calculating Environment Sets

discussed in Section 3. A major extension concerns execution environments. The value data of every numeric variable stored in the environment is now a pair, consisting of a numeric value and a unit of measurement. When expressions are evaluated, their variables' units are used to check the safety policy. For example, if `x + y` is encountered at line 15 and the corresponding value data are `[7, second]` and `[3, second]`, 10 is correctly assigned to the sum but a warning will be issued of the form `15 : x + y`.

Another major extension of C's semantics, needed also by the static checker, is w.r.t. annotations: they act like new, domain-specific instructions. An assumption `/*U assume unit(Var) = UnitExp U*/` is interpreted as follows: 1) evaluate `UnitExp` in the current environment; 2) modify the environment by associating the calculated unit to the variable `Var`, without changing its current numeric value; if `UnitExp` fails to evaluate to a correct unit due to violations of the safety policy, then the unit `fail` will be assigned to `Var`. Due to its precision in analysis (because of the exact execution path and environment), the dynamic checker can allow the user to assign a unit to any numerical, abstract memory location. Assumptions `/*U assume returns UnitExp U*/` are interpreted as follows: `UnitExp` is evaluated when the function is called and returned as unit associated to the returned number; the function is also executed to report all additional warnings. Assertions of boolean unit expressions are simply evaluated to boolean values and warnings are returned if they evaluate to false.

5.2 Static Checker.

The main idea behind our static checker is to cover all the execution paths rather than just one. A simplifying abstraction is to ignore all numerical values and only consider the domain-specific, abstract values (units of measurement) of variables. Due to the loss of precision, at each point one has to consider a *set* of environments in parallel, namely all those in which a potential execution of the program can be. Each statement is abstractly evaluated in all the environments. If the unit safety policy is violated in any of the environments then a warning is output. A new set of environments will be computed after each statement. Figure 2 shows how this is done for a conditional statement `if (i < 0) then Stmt else Stmt'`.

What makes this approach tractable in practice is the fact that most conditional statements change the units of variables the same way, so we are rarely faced with the problem of analyzing an exponential number of possibilities.

Reusing existing libraries is a dominant tendency in current software development. However, the code of reused libraries is often unaccessible except for the interfaces to the library functions, resulting in the inapplicability of the dynamic checker. But for the static checker it is *not* necessary to have the code of every function used to perform the analysis. All what is needed is to know how the units are changed after the function call, which can be associated to the function declaration via the return assumption and assertions. The static checker will verify the assertions of the arguments, apply the return assumption and then continue to check the rest of the program. If the source code is available, the static checker can also check the function's body.

The most complicated part is the treatment of loops. The general solution involves loop invariants, but, due to the lack of their understanding by ordinary programmers, we would like to avoid them as much as possible. Our solution is based on *loop patterns* that one can easily and efficiently analyze statically. One such pattern, e.g. is one in which the body of a loop does not change the set of environments; in this case the loop can be safely ignored. Another pattern is when the set of environments stabilizes after several iterations; this pattern, for example, is triggered to analyze the sorting algorithm in Subsection 4.4.

For those loops that do not fall under any of the provided patterns, loop invariants are needed. They are inserted into the body of the loop by the programmers as assertions. When the static checker encounters such assertions, it proves them, collects them and then uses them in certifying other parts of the programs. Let us explain how it works on a simple example:

```
for (i = 1 ; i < n ; i = i + 1) {
x = x * s[i];/*U assert unit (x) = meter ^ i U*/}
x = x ^ (1 / n);
```

The assertion will be attempted to be symbolically proved by induction on the loop counter. After the loop, there will be two pieces of information collected, namely `!(i < n)` and `unit(x) = meter ^ i`. Furthermore, this loop falls under the pattern stating that the control variable's increment is 1. The condition can thus be modified to `i == n`. Later, when `x = x ^ (1/n)` is evaluated, the above is used and `unit(x) = meter` is inferred.

5.3 Comparing the Two Checkers

The main advantage of the dynamic checker is the correctness of its reported warnings: any reported warning represents a violation of the unit safety policy. The user should therefore consider these reports very seriously. The main drawback of the dynamic checker is its coverage: it only covers the path that was generated by the particular test case. Therefore, other errors might exist in the analyzed program which were not revealed and which can appear when the program is executed with different numerical values as input. Another drawback of the dynamic unit safety checker is that its execution time consists of the analyzed program's execution time plus the runtime overhead. Therefore, if a program calculates a computationally complex function or does not terminate in a reasonable time, then so does the unit safety prototype, which can be a serious drawback in some applications. Another problem with the dynamic checker is that it does not support library functions whose source code is not accessible.

The main advantage of the static checker is that it covers all the reachable code, so it does not miss any unsafe expression: a careful analysis of the reported warnings can reveal all the unit safety leaks. Another advantage is its relative efficiency, because it does not execute the programs, so non-termination of the program does not imply non-termination of the checker. However, depending on the amount of automated deduction that one wants to put in such a static certifier, it can actually become rather inefficient. A major drawback of the static certifier is the potentially long list of false alarms that it reports. The user can reduce their number using assume annotations, but one should be careful when using assumptions because they can be wrong and thus present a safety threat.

6 Conclusion and Future Work

An automated deduction approach to measurement unit safety was presented, in the form of dynamic and static unit safety checkers. Future work includes incorporating other domain-specific policies and supporting other major languages.

References

[1] P. Borovanský, H. Cîrstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN. User manual – `http://www.loria.fr`.

[2] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Trans. on Prog. Lang. and Systems*, 9(1), 1987.

[3] T. Cheatham. Handling fractions and n-tuples in algebraic languages. Presented at the 15th ACM Annual Meeting, Aug. 1960.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285, 2002.

[5] Compaq. ESC for Java, 2000. URL: `www.research.compaq.com/SRC/esc`.

[6] R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. World Scientific, 1998. AMAST Series in Computing, volume 6.

[7] A. Dreiheller, M. Moerschbacher, and B. Mohr. Physcal - programming Pascal with physical units. *ACM SIGPLAN Notices*, 21(12), 1986.

[8] N. Gehani. Units of measure as a data attribute. *Comp. Lang.*, 2, 1977.

[9] N. H. Gehani. Ada's derived types and units of measure. *Software: Practice and Experience*, 15(6), 1985.

[10] J. Goguen and G. Malcolm. *Alg. Semantics of Imperative Programs*. MIT, 1996.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ*. Kluwer, 2000.

[12] P. N. Hilfinger. An Ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[13] R. T. House. A proposal for the extended form of type checking of expressions. *The Computer Journal*, 26(4), 1983.

[14] M. Karr and D. B. L. III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, 1978.

[15] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of POPL'97*. ACM, 1997.

[16] A. J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, St. Catherine's College, University of Cambridge, November 1995.

[17] R. Khanin. Dimensional analysis in Computer Algebra. In *Proceedings of ISSAC'01*. ACM, 2001.

[18] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *Proceedings of ASE'01*. IEEE, 2001.

[19] G. W. Macpherson. A reusable Ada package for scientific dimensional integrity. *ACM Ada Letters*, XVI(3), 1996.

ASE'03, IEEE, pp. 304 - 309. 2003.

[20] B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 2000.

[21] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17, 1978.

[22] M. Rittri. Dimensional inference under polymorphic recursion. In *Proceedings of Functional Programming Languages and Computer Architecture*. ACM, 1995.

[23] M. Wand. First-order identities as a defining language. *Acta Informatica*, 14, 1980.