Precise Inference of Expressive Units of Measurement Types

ANONYMOUS AUTHOR(S)

Ensuring computations are unit-wise consistent is an important task in software development. Numeric computations are usually performed with primitive types instead of abstract data types, which results in very weak static guarantees about correct usage and conversion of units. This paper presents *PUnits*, a pluggable type system for expressive units of measurement types and a precise, whole-program inference approach for these types. *PUnits* can be used in three modes: (1) modularly check the correctness of a program, (2) ensure a possible unit typing exists, (3) annotate a program with units. Annotation mode allows human inspection and is essential since having a valid typing does not guarantee that the inferred specification expresses design intent. *PUnits* is the first units type system with this capability. Compared to prior work, *PUnits* strikes a novel balance between expressiveness, inference complexity, and annotation effort. We implement *PUnits* for Java and evaluate it by specifying the correct usage of frequently used JDK methods. We analyzed 234k lines of code from eight open-source scientific computing projects with *PUnits*, which inferred 87 scientific units and generated well-specified applications. The experiments show that *PUnits* provides an effective, sound, and scalable alternative to using encapsulation-based units APIs (like `javax.measure`), enabling Java developers to reap the performance benefits of using primitive types instead of abstract data types for unit-wise consistent scientific computations.

Additional Key Words and Phrases: Type System, Type Inference, Units of Measurements, Dimensional Analysis, Scientific Computing

1 INTRODUCTION

Ensuring computations are unit-wise consistent is an important task in software development for scientific, engineering, and business domains. Performing calculations with mismatched dimensions (e.g., length, time, speed, etc.) or units (e.g., meters, millimeters, seconds, meter per second, etc.) can result in catastrophic failures. For efficiency reasons, such computations are usually performed with primitive types instead of abstract data types, which results in very weak static guarantees about correct usage and conversion of units.

The Mars Climate Orbiter disintegrated in 1998 on its approach to Mars due to one software component communicating values in Imperial units, while the receiving component expected values in International System (SI) units [Board 1999]. This mismatch cost US taxpayers $327.6 million USD.

In 2012, Mozilla added a swipe-to-close feature to its Firefox browser for Android. An error in the implementation prevented users from activating this feature. The relevant code is presented in Fig. 1. The initial minimum swipe speed was set to 1000 pixels-per-second and defined as an integer constant `MIN_SPEED` in the program (line 2). If a user swipes a browser tab with a speed (`speedX`) exceeding this minimum, the code would animate the closing of the tab (line 8). Subsequently, the feature was updated to adapt to different devices by performing the computations with respect to the screen size in inches (line 7). A bug report [Johnston 2012] indicates that the swipe speed was not modified during the feature update. As a result, the integer was interpreted as 1000 inches per second, effectively disabling the feature. As another example, two errors in Daikon[Ernst et al. 2007], a dynamic likely invariant detection tool, were caused by mismatches between seconds and milliseconds [Daikon 2003, 2004]. The standard Java type system does not prevent such errors, as the computations are performed on primitive values.

2020. 2475-1421/2020/5-ART1 $15.00
https://doi.org/

```
1  class TabSwipeGestureListener {
2    @pxPERs  int MIN_SPEED  = 1000;
3    @pxPERin int DPI        = 72;
4    @px      int VIEW_WIDTH = 1920;
5
6    void onFling(View view, @pxPERs float speedX) {
7      if (speedX / DPI > MIN_SPEED) // Error
8        animate(view, VIEW_WIDTH, (int)(VIEW_WIDTH / speedX));
9    }
10   void animate(View view, @px int final_pos, @s int duration) { ... }
11 }
```

Fig. 1. Minimized and annotated example of the Firefox Android swipe-to-close error. The type qualifiers `@px`, `@s`, `@pxPERs`, `@inPERs`, and `@pxPERin` represent pixels, seconds, pixels-per-second, inches-per-second, and pixels-per-inch, respectively. Type qualifiers are implemented as Java 8 type annotations [Ernst et al. 2012]. These qualifiers are not present in the original code [Johnston 2012]. For the annotated code, *PUnits* produces an error message for the comparison on line 7.

This paper presents *PUnits*, an expressive type system to enforce units of measurement and a precise whole-program type inference approach that helps developers annotate code with unit types. *PUnits* is implemented for Java as an optional type system [Bracha 2004]. It handles all of Java's language features and works on real-world Java applications. Nevertheless, the idea for *PUnits* is not limited to Java and can be widely adapted.

PUnits can detect the error in the Firefox example with the annotated version of the code shown in Fig. 1. The type qualifiers specify the units of the program elements, and *PUnits* reasons about the correctness of the code by checking their qualified types. Developers utilize *PUnits* by adding annotations, type checking their program, and then fixing errors or adding additional annotations. *PUnits* reports an error as it computes the resulting unit of the division on line 7 to be `@inPERs` (inches-per-second), which is incompatible to `@pxPERs` (pixels-per-second). For Daikon, *PUnits* detects reproductions of the units errors. We fully annotated the Daikon source code and *PUnits* guarantees that these kinds of errors will not be introduced again.

Developers can also use *PUnits* to perform whole-program type inference to check whether there exists any valid typing. With only `animate()` fully annotated, *PUnits* infers `speedX` to be `@pxPERs`. The choice of the unit for `DPI` determines the unit for `MIN_SPEED` and vice versa. *PUnits* determines that the program has at least one valid typing. It can infer one variable to be dimensionless and the other to `@pxPERs`. The programmer inspects the inference results for the two variables and realizes that the solution does not match their design intent, which indicates there is an error. The programmer then annotates `DPI` and `MIN_SPEED` with the intended units. *PUnits* then concludes that there is no valid typing and produces an error on line 7. Note how this error would have been missed without a human looking at the inference results and deciding whether the result matches their expectations. Therefore, *PUnits* also provides whole-program type inference with annotation mode, where the most precise units are inserted back into the source code to allow human inspection and to provide well-specified applications and libraries. *PUnits* is the first units type system with this capability.

PUnits makes contributions to the expressiveness, precision, usability, effectiveness, and soundness of units of measurement types.

- *PUnits* is *expressive*: it is parameterized by the set of base units (for example, the seven SI base units), and can represent all units defined as products and quotients of the base units. Unit calculations with arbitrary prefixes and exponents are supported. The design is not limited

to SI units; a developer can provide base units for any custom measurement system such as pixels, bytes, and currencies. The system is also parameterized by the set of mathematical operations, each supported by a function which gives the resulting unit of applying the mathematical operation to two units. *PUnits* can be instantiated for dimensional analysis by supplying a set of base dimensions instead of base units.

- *PUnits* is *precise*: our whole-program type inference approach infers the most precise typing possible when more than one type-safe solution is permitted. When inserted into source code, a precise solution prevents unintended new uses of well-annotated code. To the best of our knowledge, no prior work has attempted to infer units which are inserted into the source code; they are traditionally only used to determine whether the program type checks.
- *PUnits* is easy to use: we implemented defaulting rules, method-local flow-sensitive type refinement, parametric polymorphism over units, and receiver-dependent types to minimize the need for unit annotations in method bodies, removing unnecessary clutter. No other units type systems have receiver dependent types. By adding a few key unit specifications in application code or in commonly used libraries, inference can automatically discover and annotate every relevant program element with type-safe unit specifications.
- *PUnits* is effective: we present case studies of using *PUnits* for eight Java projects, including Daikon, totalling 234k lines of code. The experiments show *PUnits* can discover and prevent units-related errors in real-world projects. An encapsulation-based units APIs has failed to detect these errors. *PUnits* reaps the performance benefits of using primitive types instead of abstract data types to achieve unit-consistent programs. *PUnits* inferred 87 scientific units for five Java projects, and created well-specified applications and libraries with low annotation effort.
- *PUnits* is sound: we proved a minimal formalization of the system sound in Coq [Coq Development Team 2004]. The proof scripts are openly available [Anonymous-Authors 2020].

Compared to prior work, *PUnits* strikes a novel balance between expressiveness, inference complexity, and annotation effort. *PUnits* is a practical tool for improving the quality of software and can help developers write unit-correct code.

This paper is organized as follows. Sec. 2 presents an overview of *PUnits* and its type checking and inference approach. Sec. 3 presents the formalization and summarizes the Coq proof. Sec. 4 presents the design and implementation details of *PUnits*. Sec. 5 presents the case studies. Sec. 6 compares *PUnits* to related work and, finally, Sec. 7 concludes and discusses possible future work.

2 OVERVIEW

A type system for units of measurements must be expressive enough to allow the use of different measurement systems. The system must be robust to changes in the supported units and enable reusable code with selective sensitivity to units. The system must also be able to efficiently reason about program correctness, independent of which particular measurement systems are used. Finally, the system must support developers in terms of their workflows and minimize the burden required to use the system.

A pluggable type system [Bracha 2004] enhances the type system of a language to enforce stronger semantics and provide additional static guarantees. Pluggable type systems provide a set of type qualifiers to imbue additional semantic meaning to the types of a language, and enforce custom semantics through a set of type rules expressed over qualified types. Developers can annotate their code with type qualifiers and optionally use a suite of pluggable type systems to check their programs for potential errors that the standard type system of a language does not prevent.

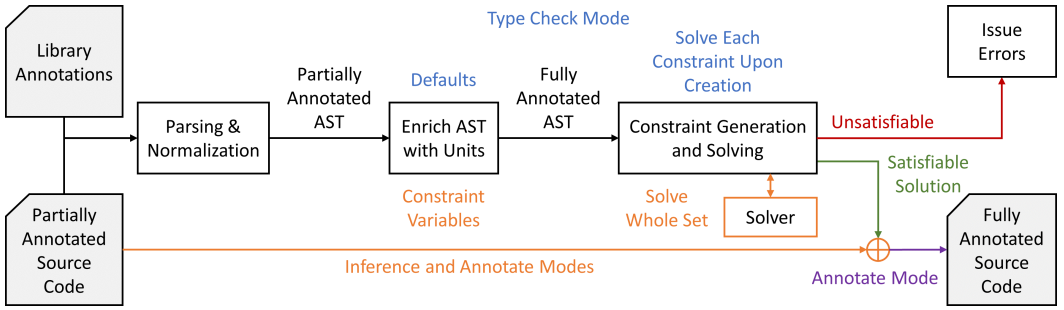

Fig. 2. Type checking and inference process with three different modes: 1) Modular type checking using defaults and eager constraint solving; 2) Whole-program type inference to ensure valid typing; and 3) Whole-program type inference with additional breakable constraints to annotate code with the most precise set of units.

PUnits is a pluggable type system that allows developers to specify units of measurement systems by defining a set of base units, and to specify the units of their program elements via type qualifiers. Each type qualifier specifies a single unit. For example, `@m int` is a qualified type specifying an integer with the unit of meters. The qualifier `@Dimensionless` specifies the unit of truly dimensionless quantities, such as number literals, e, and π, and usually does not need to be written. The qualifiers are organized as a flat lattice over the set of scientific units, with top $\top$ and bottom $\bot$ type qualifiers. We use the term *unit* to denote both the scientific unit of a program element as well as the type qualifier used to annotate the program element.

Developers provide partially-annotated source code and specifications for library code as input to *PUnits*, which first normalizes the units into an internal representation (Sec. 2.1). *PUnits* enforces unit-consistent computations by generating and solving typing constraints through syntax-guided constraint generation rules. A user of *PUnits* can pick between one of three modes: 1) modular type checking (Sec. 2.2), 2) whole-program type inference to ensure valid typing (Sec. 2.3), and 3) whole-program type inference to annotate the program with the most precise set of units (Sec. 2.3.1). The modes are respectively given the short names of type check mode, inference mode, and annotate mode. Fig. 2 illustrates the overall type checking and type inference process of *PUnits*.

2.1 Base Units and the Normalized Representation of Units

PUnits internally represents all units through a normalized representation to efficiently reason about program correctness. A *base unit* is the unit of measurement from which all other units from the same dimension can be derived. Each base unit is represented by a symbol. For SI, the base units are {`m`, `s`, `g`[1], `A`, `cd`, `K`, `mol`}.

PUnits represents scientific units as a single prefix and a product of one or more base units, each raised to an integer power: $p\ \overline{u^z}$ (where $\overline{X}$ denotes a sequence of elements $X_1, \ldots, X_k$). Each base unit appears exactly once.

The representation can support any unit system. In the Java implementation, p is encoded as a base-10 prefix with an integer exponent to support SI-like units: $p = 10^z$. Thus, we need to declare additional base units for units that are not a base 10 multiple (e.g., inch vs cm). Fig. 3 shows some examples of how units are represented in *PUnits*. The prefix p could be encoded as floating-point value. However, the precision of analysis will be subjected to floating-point rounding errors, and safe floating-point comparisons must be used. The set of base units is customizable, allowing for

[1]Although kg is defined as the SI base unit for mass, *PUnits* uses g since the metric prefix is captured and encoded in the prefix component of the normalized representation.

Unit	10^z	g^z	m^z	s^z	...
m^2	0	0	2	0	0
kg	3	1	0	0	0
N	3	1	1	-2	0
kN	6	1	1	-2	0
Dimensionless	0	0	0	0	0

Fig. 3. Example unit representations. Squared-meter is represented as m^2 and kilogram is represented as 10^3 g with g as the base unit. All SI prefixes of a unit are multiplied together into one prefix. Newton is represented by 10^3 g m s^{-2} and a kilo-newton is normalized and represented by 10^6 g m s^{-2}. `@Dimensionless` is represented by 10^0.

```
1  void main() {
2    @1 long start = System.currentTimeMillis();    // @ms <: @1
3    /*... perform operations ...*/
4    @2 long end = System.currentTimeMillis();      // @ms <: @2
5    @3 long limit = 1000@4;                        // @4 <: @3
6    checkTimeLimit((end - start)@5, limit);        // @5 <: @6, @3 <: @7
7  }
8
9  void checkTimeLimit(@6 long duration, @7 long timeout) {
10   if (duration > timeout) {...}                 // @6 <: @7 or @7 <: @6
11 }
```

Constraint Variable	@1	@2	@3	@4	@5	@6	@7
Type Check Mode	@ms	@ms	D	D	@ms	D	D
Inference Mode	⊤	⊤	⊤	⊤	⊤	⊤	⊤
Annotate Mode	@ms	@ms	@ms	@ms	@ms	@ms	@ms

Fig. 4. A program which performs a number of operations and then checks to see if the operations were performed within a time limit. Placeholders `@1` to `@7` are used to mark the locations for which types need to be determined. `System.currentTimeMillis()` is annotated to return values with the unit of `@ms` (milliseconds). The table below the code presents the units assigned to the placeholders `@1` through `@7` in each of the three solving modes, where D stands for `@Dimensionless`.

base units such as currencies or abstract quantities and lengths such as pixels. The representation is compact, and allows for efficient calculation of the resulting units of various arithmetic operations. Two units are equal if and only if every component in their normalized representations are pairwise equal.

PUnits can also be instantiated for dimensional analysis by using a set of base dimensions instead of base units, together with the fixed prefix 1. Unit-wise analysis offers more precision than dimensional analysis and is able to detect more errors.

2.2 Modular Type Checking

Modular type check mode allows methods and classes to be type-checked independently. It give developers quick feedback on potential units-related errors, with a method-local view of potential problems.

PUnits uses defaults for missing units (Sec. 4.1.1). The defaults are chosen to give errors across method and class boundaries, allowing developers to quickly catch potential problems and focus on fixing one method or class at a time. A set of mandatory constraints (Sec. 3.2) is generated by

syntax-guided constraint generation rules (Sec. 3.3). In type check mode, each constraint is solved upon creation. Unsatisfied constraints cause compilation errors.

The modes are illustrated through the example presented in Fig. 4, together with the constraints generated for each line. In type check mode, after applying defaults and method-local type refinement (Sec. 4.1.2), without adding any additional annotations, initial type checking will fail. An error is issued on line 6 for passing a `@ms` value to a parameter which expects a `@Dimensionless` value. Type check mode gives method-local views of potential errors, some of which may be due to a lack of annotations in the code. By iteratively applying the type check mode, all units mismatches in the code will be revealed. The method call and comparison are type-safe if `@3`, `@4`, `@6`, and `@7` are annotated with `@ms`.

2.3 Whole-program Type Inference

Whole-program type inference mode ensures a valid typing exists for a program and, optionally, infers the most precise units for a program (see Sec. 2.3.1 below). Inference mode pinpoints the set of code locations across the program that together could cause a units-related error.

In whole-program type inference, constraint variables are created, which are placeholders for concrete units. The same mandatory constraints (Sec. 3.2) are generated by syntax-guided constraint generation rules (Sec. 3.3), as in type check mode. The constraints for the whole program are collected and solved together by an MaxSMT solver (Sec. 4.3). Multiple type-safe solutions are possible in inference mode and, as long as a solution exists, inference succeeds.

In Fig. 4, the constraints are solved together in inference mode and determined to be satisfiable. One possible solution assigns $\top$ as the unit for all constraint variables. The program is therefore type safe. However, this solution does not use the most precise units that could be assigned to the constraint variables.

As a simple unsatisfiable example, we can annotate parameter `duration` on line 9 with `@s`. From lines 2 and 4, the constraints require `@1` and `@2` to be `@ms` or $\top$. This propagates to the subtraction on line 6, and `@5` must be `@ms` or $\top$. Since neither unit is a subtype of `@s`, the constraints introduced in lines 2, 4, and 6 are together unsatisfiable.

The errors given in type check mode are not a superset of the errors given in inference mode. Errors given in inference mode provide a whole-program view of the reasons why no type-safe solutions can be inferred, and indicate potential errors.

2.3.1 *Annotation Mode.*

Annotation mode is whole-program type inference with additional, breakable, preference constraints (Sec. 4.3). It annotate the program with the most precise units possible. This is intended to help developers create well-specified applications and libraries.

For example, if a parameter can be annotated with `@m` or $\top$ as its unit, then `@m` is the more precise typing and is preferred over the more general typing $\top$.

PUnits chooses to infer the most precise typing to prevent accidental misuses of annotated code by newly added code. *PUnits* also prefers inferring `@Dimensionless` as a unit for under-constrained constraint variables. Annotate mode inserts all non-default units from its solution into the corresponding types in source code. Annotations on local variables are also not inserted, because flow-sensitive type refinement can infer such annotations (Sec. 4.1.2). This decreases noise in the fully-annotated program. Preferring units to be `@Dimensionless` will raise errors when developers introduce additional units into a program. Like in inference mode, if any of the mandatory constraints cannot be satisfied, errors are raised for the code locations which generated the unsatisfiable constraints.

When using annotation mode for the example in Fig. 4, the solution assigns `@ms` to each constraint variable as it is the most precise unit for each of the types. The solution is inserted into the source

$$
\begin{array}{rrl}
\text{Program} & P & ::= \overline{vd}\ \overline{s} \\
\text{Variable Declaration} & vd & ::= T\ v = z \\
\text{Assignment Statement} & s & ::= v = e \\
\text{Expressions} & e & ::= l \mid v \mid e\ op\ e \\
\text{Labeled Literal} & l & ::= T\ z \\
\text{Types} & T & ::= \top \mid \bot \mid p\ \overline{u^z} \mid \alpha \\
\text{Arithmetic Operations} & op & ::= + \mid - \mid * \mid \div \mid \ldots \\
\text{Base Units} & u & ::= \ldots
\end{array}
$$

v variable identifier
p prefix number literal
z signed integer literal
α constraint variable identifier

Fig. 5. Syntax and naming conventions. Constraint variables α are only used in inference and annotate modes, as placeholders for concrete units. The set of base units u and the set of arithmetic operators op can be customized.

code with the method signature on line 9 transformed into `void checkTimeLimit(@ms long duration, @ms long timeout)`. The developer can continue to type check and infer units for a larger program, ensuring all calls to `checkTimeLimit()` must pass `@ms` arguments.

3 *PUNITS* TYPE SYSTEM AND FORMALIZATION

This section formalizes the *PUnits* type system and inference approach and presents a core calculus, *πUnits*, as a minimal imperative language with integers, arithmetics, and units. We proved soundness in Coq [Coq Development Team 2004] and the proof scripts are openly available [Anonymous-Authors 2020].

3.1 Syntax of *πUnits*

The syntax of *πUnits* and the naming conventions are presented in Fig. 5. A program P consists of a sequence of variable declarations $\overline{vd}$, followed by a sequence of assignment statements $\overline{s}$. A variable declaration vd pairs a type T with an identifier v, and assigns an initial integer value z to the variable. An expression e can be a labeled literal l, a read of a variable v, or an arithmetic operation $e\ op\ e$ between two sub-expressions for some operation op. A labeled literal is an integer literal z labeled with a type T. The set of types T consists of top $\top$, bottom $\bot$, and normalized unit representations $p\ \overline{u^z}$ (see Sec. 2.1). In inference and annotate modes, T also includes constraint variables α.

3.2 Constraint Variables and Typing Constraints

A constraint variable α is a placeholder for a concrete unit. A fresh constraint variable is introduced for each location that is missing a unit and for the resulting unit of each arithmetic operation. Inference assigns one concrete unit to each constraint variable.

πUnits generates three kinds of constraints σ over types:

- Well-formedness constraint $wf(\alpha)$: enforces that any satisfying solution for constraint variable α is uniquely interpretable as a single unit.
- Subtype constraint $T_a <: T_b$: enforces T_a to be the same type or a subtype of T_b.
- Arithmetic constraint $\alpha = T_a\ op\ T_b$: enforces that α will be equal to the result of the arithmetic computation of $T_a\ op\ T_b$, for some defined arithmetic operation op. *πUnits* abstracts over a concrete set of arithmetic operations. op constraints build upon the idea of Viewpoint Adaptation [Dietl et al. 2007, 2011b] and can be thought of as computing the result type of the arithmetic operator op.

$$\text{Program:}\quad \boxed{\vdash P : \Gamma, \Sigma} \qquad \frac{\emptyset \vdash \overline{vd} : \Gamma, \Sigma_{\overline{vd}} \quad \Gamma \vdash \overline{s} : \Sigma_{\overline{s}}}{\vdash \overline{vd}\ \overline{s} : \Gamma, \Sigma}$$

$$\text{Variable Declaration:}\quad \boxed{\Gamma \vdash vd : \Gamma', \Sigma} \qquad \frac{\vdash T \text{ OK} : \Sigma_{vd}}{\Gamma \vdash T\ v = z : \Gamma\{v \rightarrow T\}, \Sigma_{vd}}$$

$$\text{Statement:}\quad \boxed{\Gamma \vdash s : \Sigma} \qquad \frac{\Gamma \vdash v : T_v, \emptyset \quad \Gamma \vdash e : T_e, \Sigma_e \quad \vdash T_e <: T_v : \Sigma_{<:}}{\Gamma \vdash v = e : \Sigma_e \cup \Sigma_{<:}}$$

$$\text{Expression:}\quad \boxed{\Gamma \vdash e : T, \Sigma} \qquad \frac{}{\Gamma \vdash T\ z : T, \emptyset} \qquad \frac{T = \Gamma(v)}{\Gamma \vdash v : T, \emptyset}$$

$$\frac{\alpha \text{ fresh} \quad \Gamma \vdash e_1 : T_1, \Sigma_1 \quad \Gamma \vdash e_2 : T_2, \Sigma_2}{\Gamma \vdash e_1\ op\ e_2 : \alpha, \Sigma_1 \cup \Sigma_2 \cup \{wf(\alpha), \alpha = T_1\ op\ T_2\}}$$

Fig. 6. Constraint generation rules. The sequence forms of the judgments for $\Gamma \vdash \overline{vd} : \Gamma', \Sigma$ and $\Gamma \vdash \overline{s} : \Sigma$ are standard and are presented in Fig. 17 in Appx. A. Helper judgments are defined in Fig. 18 in Appx. A. The generated constraint set Σ encodes all constraints that need to be satisfied to give a type-correct program.

3.3 Syntax-directed Constraint Generation Rules

Fig. 6 shows the syntax-directed constraint generation rules. An environment Γ maps variables to their types. Notation $\Gamma(v)$ looks up the type of a variable v from Γ. A constraint set Σ contains the constraints σ needed to satisfy the particular rule. A variable declaration requires that its declared type is well-formed, given by the well-formed judgment ($\vdash T$ OK) defined in Fig. 18 in Appx. A. This judgment generates a well-formedness constraint for type variables. All other types are always well-formed. A variable declaration updates the environment Γ by mapping variable v to its declared type T, and produces constraint Σ_{vd}. An assignment statement requires that the type of the expression be a subtype of the variable type. Helper subtype judgment ($\vdash T_e <: T_v : \Sigma$) defined in Fig. 18 in Appx. A produces a constraint if either the type of the variable or expression is a constraint variable. The constraints for an assignment includes this subtype constraint and any constraints from the expression.

The rules for expressions are as follows. For a labeled literal expression $T\ z$, the type of the value is returned. For a variable read expression v, the type of the variable is retrieved by look-up in Γ. The judgments for labeled literals and variable reads do not generate any constraints. For arithmetic expressions, *πUnits* generates a fresh constraint variable α to hold the result type of the operation, a well-formedness constraint for α, an arithmetic constraint ($\alpha = T_1\ op\ T_2$), and the constraints generated for the sub-expressions.

3.4 Small-Step Operational Semantics

πUnits models a stack frame F as a map of variables to their static types T_v and their labeled values $T_l\ z$. The label type of a value T_l is its declared type or a type computed at runtime.

In *πUnits*, a program can evaluate to *STUCK* if a variable with type T_v is assigned a value with type T_l and T_l is a supertype of T_v. This subtype requirement in the operational semantics models misusing computed results with mismatching units. A program does not get *STUCK* during arithmetic computations in *πUnits*. This is required for our soundness proof to show that well-typed programs never get stuck.

Aside from the frame model, the operational semantics is standard. The details are presented in Fig. 19 in Appx. A.

$$\text{L-ST-Consistent:} \quad T_1' <: T_1 \implies (T_1' \; op \; T_2) <: (T_1 \; op \; T_2)$$
$$\text{R-ST-Consistent:} \quad T_2' <: T_2 \implies (T_1 \; op \; T_2') <: (T_1 \; op \; T_2)$$

Fig. 7. *op* and Subtype Consistent Axioms.

3.5 Coq Proof Summary

We proved the soundness of type checking in *πUnits* using the standard approach of proving progress and preservation theorems [Wright and Felleisen 1994]. The proof omits modeling constraint variables α, as they do not arise for type checking. The two theorems and additional supporting lemmas and summaries of their proofs are presented in Appx. B.

An interesting subcase arises for preservation during expression reductions. Lemma 3.1 defines the supporting lemma.

Lemma 3.1 (Preservation during expression reduction).
$$\Gamma \vdash e : T \wedge \vdash F \text{ OK} \wedge F, e \Rightarrow e' \implies \exists T' \cdot \vdash T' <: T \wedge \Gamma \vdash e' : T'.$$

The type of a value stored in a variable can be a subtype of the static type of the variable. For the subcase where $e = e_1 \; op \; e_2$ with static type $T_1 \; op \; T_2$ for some given operation *op*, we must show that the run-time type of the expression is always a subtype of its static type. This then ensures that the type of the final result of the computation is always a subtype of the static type of the variable to which it is being assigned, thus the program never gets *STUCK*.

If e_1 can reduce to e_1', by the inductive hypothesis $\Gamma \vdash e_1' : T_1'$ for some $T_1' <: T_1$ then $\Gamma \vdash e' : T_1' \; op \; T_2$. We are left with the obligation that $(T_1' \; op \; T_2) <: (T_1 \; op \; T_2)$, which is imposed upon the definition of *op* for any given *op*. Similarly, if e_2 can be reduced to e_2' and $\Gamma \vdash e_2' : T_2'$, the obligation $(T_1 \; op \; T_2') <: (T_1 \; op \; T_2)$ is also imposed on the definition of *op*. The two obligations are defined as the L-ST-Consistent and R-ST-Consistent axioms shown in Fig. 7. While the *op* between scientific units must obey the arithmetic laws, such as $a^n \times a^m = a^{n+m}$, the axioms inform how to define *op* between $\top$ or $\bot$ with a unit and with each other.

With operations defined in a way that satisfies these two axioms, the run-time type of the final result of reducing a well-typed expression will always be a subtype of the static type of the expression. Since the type rules require the type of an expression to be a subtype of a variable during an assignment, by subtype transitivity the run-time type of an expression is also a subtype of the variable. The proof concludes that the frame is always well-formed during execution and a well-typed *πUnits* program never gets *STUCK*.

3.6 Extending *πUnits* to the Full Java Language

Although *πUnits* is minimal, it sufficiently models the necessary core calculus to support all of *PUnits*'s features and extends to the full Java language.

πUnits models variable declarations, reads, and updates, which directly model Java fields and local variables. Wherever Java performs a subtype check, such as for method invocations, subclassing and method overloading, or generic type argument bounds, *PUnits* also performs a subtype check or encodes a subtype constraint.

πUnits also models arithmetic operations, which models the checks and constraints generated by *PUnits* for Java's arithmetic operations through the binary operators or through methods such as `Math.addExact(int x, int y)`. *PUnits* provides definitions of addition, subtraction, multiplication, division, and modulo which satisfy the op and subtype consistency axioms. The encoding of the operations for the solver are presented in Sec. 4.3. Supporting a range of mathematical operations is a feature of *πUnits*. However, it is not expected of the developer to provide these definitions and proofs as most mathematical operations can be composed from basic operations that satisfy the

axioms. We leave the definition and support of other arithmetic operators, such as exponentiation and roots, as future work.

All other operations and features supported by *PUnits* are built upon subtyping. For number comparison operations, *PUnits* requires that one of the two arguments be a subtype of the other. Similarly, it encodes a pair of subtype constraints for comparisons during inference. A comparison between a scientific unit and $\bot$ is permitted to support null reference comparisons, as $\bot$ is the the type of `null`. Comparisons between different scientific units is forbidden as they are never subtypes of each other.

The *πUnits* proofs revealed the insight about how to define *op* with $\top$ and $\bot$. We believe that extending the formalization by explicitly modeling additional features of the Java language will not provide additional insights about the units relations other than, for example, the standard rules for behavioral subtyping.

4 *PUNITS* DESIGN AND IMPLEMENTATION

PUnits is implemented for Java. It is built on top of the Checker Framework [Dietl et al. 2011a; Papi et al. 2008] and Checker Framework Inference [Dietl et al. 2011b], and is openly available. The Checker Framework is a pluggable type system development framework for Java, allowing type systems to plug into the Java compiler to enforce stronger semantics. Checker Framework Inference is a framework which extends type systems developed on top of the Checker Framework with whole-program type inference capabilities, using solvers to solve typing constraint sets.

Sec. 4.1 explains *PUnits*'s key features that reduce annotation effort. Sec. 4.2 presents the syntax for declaring base and derived units. Finally, Sec. 4.3 presents the constraint solving approach and the encoding of constraints.

4.1 Type System Features

PUnits supports annotation defaulting, method-local flow-sensitive type refinement, parametric polymorphism over units, and receiver-dependent units. These features minimize the need for unit annotations in method bodies, removing unnecessary clutter.

4.1.1 Defaulting Rules. The type rules forbid assignments and comparisons of values with incompatible units. To reduce the annotation burden in type check mode, the following default units are used for types with missing units:

- $\top$ is the default unit for local variables and for implicit upper bounds of type variables.
- $\bot$ is the default unit for implicit and explicit lower bounds of type variables.
- `@Dimensionless` is the default unit for all other type use locations, and the default unit of number literals.

Setting the default unit of local variables to $\top$ reduces the annotation burden, as the units of local variables are refined through flow-sensitive type refinement (Sec. 4.1.2).

Setting the default bounds of type variables to $\top$ and $\bot$ respectively allows users to instantiate generic data types and use generic methods with any unit, for example `List<@N Float>`.

Setting `@Dimensionless` as the default unit for all other type use locations restricts all flow of values with units in or out of unannotated formal parameters and returns of methods. It also forbids the use of methods with unannotated receiver parameter `this` on objects that are created with a unit, which prevents unintended use of methods that are incompatible with units.

4.1.2 Method-local Flow-sensitive Units Refinement. PUnits refines the unit of local variables through method-local flow-sensitive type refinement. If unannotated, the unit of a local variable starts as $\top$ and is refined upon every assignment.

```
1 void methodLocalFlowSensitiveRefinement() {
2   @1 double d;
3   (@2) d = (@deg double) 45.0;      // @2 <: @1, @2 = @deg
4   (@3) d = Math.toRadians(d@2);     // @3 <: @1, @3 = @rad, @2 <: @deg
5   Math.toRadians(d@3);              // @3 <: @deg => Error
6 }
```

Fig. 8. Method-local flow-sensitive units refinement.

```
1 @PolyUnit int max(@PolyUnit int x, @PolyUnit int y) { /*...*/ }
2 @m int mMax = max(m1, m2);
3 @s int sMax = max(s1, s2);
4 @m int bad = max(m1, s1);  // Error: assignment type incompatible
```

Fig. 9. Polymorphic method example. The variables `m1` and `m2` have the unit of `@m` (meters). The variables `s1` and `s2` have the unit of `@s` (seconds).

Method-local flow-sensitive refinement was not shown in Fig. 4 to avoid confusion. The example in Fig. 8 highlights the types and generated constraints. Constraint variables are introduced for types in variable declarations. @1 is the constraint variable for the declarations of `d`. During type checking, the types of local variables default to $\top$. On line 3, a refinement type @2 is introduced for the type of `d` after the assignment. The unit of *d* is refined from $\top$ to `@deg` (degrees). The refined type of `d` is safely used as an argument to `toRadians()` on line 4. Another refinement type @3 for `d` is generated as it is reassigned another value. The method `toRadians()` returns a `@rad` value, which is stored in refinement type @3 of `d`. Note how the type of `d` can change through these re-assignments. Line 5 causes an error, as `toRadians()` requires its argument to be `@deg` and the refined type of `d` is now `@rad`.

PUnits permits reusing local variables to store values with various units. However, it forbids the incorrect use of local variables. For each re-assignment of a local variable, a new refinement type is introduced. Each refinement type must be a subtype of the declared type of the variable and a corresponding subtype constraint is generated. Additionally, an equality constraint is generated between the expression type and the refinement type, to ensure the refinement type has the most specific type from the expression.

4.1.3 Parametric Polymorphism of Units. *PUnits* supports parametric polymorphism of units in two forms: using the special `@PolyUnit` type qualifier to parameterize methods, or using type polymorphism (Java generics) to parameterize methods or classes with units-qualified types.

PUnits supports parametric polymorphism of units for methods via the special type qualifier `@PolyUnit`. For example, a developer can declare a method that computes the maximum of any two values with corresponding units as shown in line 1 of Fig. 9. When this method is invoked with two variables in meters (line 2), it will return the maximum value with the unit of meters. When invoked with two variables in seconds (line 3), it will return the maximum value with the unit of seconds.

At every method call site, *PUnits* computes a unit which is used to instantiate `@PolyUnit` for the called method. The unit is computed as the least-upper-bound of the method receiver and/or argument units. The computed unit is $\top$ for the call on line 4, because the arguments use different units `@m` and `@s`, and $\top$ is used as the return type of the method call. This reflects the fact that it is statically impossible to determine which of the two values gets returned and thus which unit is associated with the returned value. The assignment on line 4 gives an error, because it requires the result to be a meter.

```
1 enum TimeUnit {
2   @s SECOND,
3   @ns NANOSECOND;
4   @RDU long convert(long duration, TimeUnit unit) {...}
5   @ns long toNanos(@RDU long duration) {...}
6 }
7 @s int good1 = SECOND.convert(10, NANOSECOND);
8 @ns int bad1 = SECOND.convert(10, NANOSECOND); // Error
9 @ns int good2 = SECOND.toNanos(s);
10 @ns int bad2 = SECOND.toNanos(ns); // Error
```

Fig. 10. Receiver-dependent units examples. Variables *s* has unit type `@s` (second) and variable *ns* has unit type `@ns` (nanosecond).

PUnits utilizes polymorphic method signatures in inference and annotate modes, and generates the corresponding constraints at each call site. Fresh type variables are generated at each call site and constrained to be the least-upper-bound of the arguments. Within a method declaration, the parameters of polymorphic methods are checked by substituting $\top$ for `@PolyUnit`, and the return is checked by substituting $\bot$ for `@PolyUnit`, since it can be instantiated with any unit.

PUnits does not infer that a method should be unit-wise polymorphic. None of the projects in our case study required a method to be annotated with `@PolyUnit` to satisfy constraints. We therefore did not need to design a strategy to infer `@PolyUnit` method signatures.

4.1.4 Receiver-Dependent Units. A return type or method parameter type sometimes needs to be sensitive to the actual method receiver type. *PUnits* supports receiver-dependent units by introducing the `@RDU` type qualifier for return types or method parameter types. A `@RDU` type is resolved using the actual receiver type, similar to how viewpoint adaptation works in ownership types [Dietl et al. 2007, 2011b]. The viewpoint adaptation operation takes two inputs, the receiver type and the declared type, and yields a single result type. When type checking a method invocation, any occurrence of `@RDU` in the signature is substituted with the receiver type; all non-`@RDU`-types stay unchanged. Type checking happens against the viewpoint adapted signature of the method. `@RDU` is never inferred. In inference and annotate mode, any occurrence of `@RDU` in the signature introduces a new constraint variable and an equality constraint is introduced between the receiver type and the new constraint variable. This constraint variable is used for further checks instead of the declared parameter or return type.

Consider the example in Fig. 10. The return type of `convert()` on line 4 and the parameter type of `toNanos()` on line 5 are `@RDU`. When these methods are invoked, we enforce the return type of `convert()` and the argument to `toNanos()` to be the same as their method receivers. An invalid type assignment is issued on line 8 as `SECOND.convert()` returns type `@s`. An invalid type argument is issued on line 10 as `SECOND.toNanos()` accepts type `@s` as its argument.

4.2 Declaring Units

PUnits allows developers to declare a base unit by defining a Java 8 type annotation [Ernst et al. 2012] with the meta-annotation `@BaseUnit`. A meta-annotation is an annotation written on the declaration of another annotation. The annotation name of a base unit is used as its symbol.

Type annotation `@UnitsRep` implements $\top$, $\bot$, and the normalized representation. The annotation contains two boolean fields for $\top$ and $\bot$, an integer field `p` for the base-10 prefix, and an array of `@BUC` annotations. The `@BUC` annotation contains an annotation class literal field to reference a base unit annotation, and an integer field to store an exponent. Each `@UnitsRep` instance is uniquely

```
@Target({TYPE_USE, TYPE_PARAMETER})
public @interface UnitsRep {
  boolean top() default false;
  boolean bot() default false;
  int p() default 0;
  BUC[] bu() default {};
}
```

```
public @interface BUC {
  Class<? extends Annotation> u();
  int e() default 1;
}
```

Fig. 11. The `@UnitsRep` and `@BUC` annotations which implement the normalized representation.

interpreted as either $\top$, $\bot$, or a scientific unit depending on the boolean and integer values. *PUnits* issues an error if an instance cannot be uniquely interpreted. Fig. 11 shows the definition of the `@UnitsRep` and `@BUC` annotations.

Developers can use the `@UnitsRep` annotation to declare derived units. For example, Velocity can be declared as `@UnitsRep(bu={@BUC(u=m.class), @BUC(u=s.class, e=-1)})`. Developers do not have to enumerate every base unit in the `@BUC` array; *PUnits* automatically assumes an exponent value of 0 for any omitted base units. This also enables any existing annotated code to be used in future analyses with a larger set of base units.

The `@UnitsRep` annotation is also a meta-annotation for use in declaring nickname annotations. A nickname annotation is a syntactically friendlier version of its corresponding `@UnitsRep` instance. `@Dimensionless` is implemented as a nickname annotation for `@UnitsRep()` with all zero exponents. $\top$ and $\bot$ is implemented by the nickname annotations `@UnitsTop` and `@UnitsBottom` with appropriate boolean values. *PUnits* will always look up and utilize the `@UnitsRep` representation for analysis when it encounters nickname annotations. *PUnits* will also present errors using nickname annotations, for ease of understanding.

4.3 Encoding of Constraints for Solvers

PUnits abstracts away the low-level constraint encoding from the high-level constraints through a solver interface. The solver abstraction allows *PUnits* to experiment with different solving techniques and solver systems.

PUnits encodes constraint variables and constraints as a Maximum Satisfiability Modulo Theory (MaxSMT) problem using the linear integer arithmetic and boolean theories. A MaxSMT problem is similar to a Maximum Boolean Satisfiability (MaxSAT) problem, but incorporates additional theories. A MaxSMT problem consists of hard constraints and soft or breakable constraints with weights. A MaxSMT solver will generate a solution that satisfies all the hard constraints, while maximizing the weight of satisfied soft constraints. Encoding the constraint set Σ as a MaxSMT problem allows *PUnits* to take advantage of the optimizations that go into existing SMT solvers. *PUnits* uses Z3 as its SMT solver [De Moura and Bjørner 2008]. The complexity of MaxSAT solvers is appropriate for the inference of expressive type systems [Juma et al. 2020]. Our use of a MaxSMT solver is appropriate because the *PUnits* constraints additionally require integer theories for inferring exponents.

Each constraint variable is encoded as two boolean variables β^{top} and β^{bot} which respectively represents $\top$ and $\bot$, and one integer variable ω per exponent in the normalized representation, including the base-10 prefix exponent. Each constraint presented in Sec. 3.2 is encoded as a predicate expressed over the boolean and integer variables.

The encoding of the hard constraints along with the detailed encoding for Java's arithmetic operators are:

- Well-formed Constraint ($wf(\alpha)$): ensures each constraint variable represents a unique type.

$$wf(\alpha) \;:=\; (\beta^{top} \wedge \neg\beta^{bot} \wedge \omega^{z_1} = 0 \wedge \cdots \wedge \omega^{z_k} = 0) \vee (\neg\beta^{top} \wedge \beta^{bot} \wedge \omega^{z_1} = 0 \wedge \cdots \wedge \omega^{z_k} = 0) \vee (\neg\beta^{top} \wedge \neg\beta^{bot})$$

- Subtype Constraint (T_a <: T_b): given the type lattice, it is enough to check whether the subtype is the bottom type, the supertype is the top type, or if all exponents are equal.
  $T_a <: T_b \;:=\; \beta_b^{top} \vee \beta_a^{bot} \vee (\omega_a^{z_1} = \omega_b^{z_1} \wedge \cdots \wedge \omega_a^{z_k} = \omega_b^{z_k})$
- Comparison Constraint (T_a <:> T_b): two types are comparable if one is a subtype of the other.
  $T_a <:> T_b \;:=\; \beta_a^{top} \vee \beta_b^{top} \vee \beta_a^{bot} \vee \beta_b^{bot} \vee (\omega_a^{z_1} = \omega_b^{z_1} \wedge \cdots \wedge \omega_a^{z_k} = \omega_b^{z_k})$
- Equality Constraint ($T_a = T_b$): two types are equal if their encodings are equal.
  $T_a = T_b \;:=\; \beta_a^{top} = \beta_b^{top} \wedge \beta_a^{bot} = \beta_b^{bot} \wedge \omega_a^{z_1} = \omega_b^{z_1} \wedge \cdots \wedge \omega_a^{z_k} = \omega_b^{z_k}$
- Addition Constraint ($\alpha_c = T_a + T_b$): the operation-subtyping consistency axiom (Fig. 7) guides our definition for the encoding of operations. The resulting type is an upper bound of the two arguments. Note that we do not require that it is a least upper bound.
  $\alpha_c = T_a + T_b \;:=\; T_a <: \alpha_c \wedge T_b <: \alpha_c$
- Subtraction Constraint ($\alpha_c = T_a - T_b$): has the same encoding as addition.
- Multiplication Constraint ($\alpha_c = T_a * T_b$): if either argument is $\top$, the result has to be top. If one argument is $\bot$, while the other argument is not $\top$, the result is $\bot$. If neither of the arguments is $\top$ or $\bot$, the resulting exponents are added.
  $$\alpha_c = T_a * T_b \;:=\; ((\beta_a^{top} \vee \beta_b^{top}) \wedge \beta_c^{top}) \vee (\beta_a^{bot} \wedge \neg\beta_b^{top} \wedge \beta_c^{bot}) \vee (\beta_b^{bot} \wedge \neg\beta_a^{top} \wedge \beta_c^{bot}) \vee (\neg\beta_a^{top} \wedge \neg\beta_b^{top} \wedge \neg\beta_c^{top} \wedge \neg\beta_a^{bot} \wedge \neg\beta_b^{bot} \wedge \neg\beta_c^{bot} \wedge \omega_c^{z_1} = \omega_a^{z_1} + \omega_b^{z_1} \wedge \cdots \wedge \omega_c^{z_k} = \omega_a^{z_k} + \omega_b^{z_k})$$
- Division Constraint ($\alpha_c = T_a \div T_b$): has the same encoding as multiplication except the resulting exponents are subtracted.

In annotate mode, *PUnits* generates additional breakable clauses that express preferences. The solver will attempt to satisfy all breakable clauses. The additional breakable clauses (enclosed in []) for the well-formed, subtype, and comparison constraints are:

- Well-formed Constraint ($wf(\alpha)$): the breakable constraints prefer solutions that are units (that is, not $\top$ or $\bot$) and prefers dimensionless over other units.
  $wf(\alpha) := \ldots \wedge [\neg\beta^{top} \wedge \neg\beta^{bot}] \wedge [\omega^{z_1} = 0 \wedge \cdots \wedge \omega^{z_k} = 0]$
- Subtype Constraint (T_a <: T_b): the breakable constraint prefers that the subtype and the supertype are equal.
  $T_a <: T_b := \ldots \wedge [T_a = T_b]$
- Comparison Constraint (T_a <:> T_b): the breakable constraint prefers that the subtype and the supertype are equal.
  $T_a <:> T_b := \ldots \wedge [T_a = T_b]$

5 EXPERIMENTS

This section presents several experiments to evaluate *PUnits*'s type checking and whole-program type inference capabilities. We answer three research questions:

RQ1: What are the benefits and trade-offs of using *PUnits* versus existing Java unit libraries?
RQ2: Can *PUnits* detect and prevent units-related errors in real-world projects?
RQ3: What is the compilation overhead of *PUnits* in each of its three modes?

To answer these research questions, we provided fully annotated JDK specifications, including the `Math` trigonometry methods, `System.nanoTime()`, `System.currentTimeMillis()`, and `Thread.sleep()`. In total, we added 33 annotations to 18 JDK methods. These methods were selected because they are used in a wide range of projects and are prone to misuse.

The benchmark consists of open-source Java scientific-computing projects that utilize the annotated methods, and we can successfully build them (unfortunately this is not true for all open-source

Project	Purpose	Files	SLOC	Arith	Comp	Time	Thread	Math
Daikon	Invariant detection Tool	734	168983	12671	16497	59	1	0
exp4j	Math Expression Evaluator	30	5129	945	229	7	1	54
GasFlow	Gas Pipeline Graph Model	189	15233	156	768	4	0	2
imgscalr	Image-scaling Library	11	1178	18	79	12	0	3
jblas	Matrix Library	74	11841	471	1055	7	1	39
JLargeArrays	64-bit-index Arrays	17	11337	868	2161	54	0	2
jReactPhysics3D	3D Physics Engine	105	10455	672	465	1	0	22
react	Real-time Physics Library	63	10095	579	463	4	2	50
Total		1223	234251	16380	21717	148	5	172

Fig. 12. The purpose, size, and number of used arithmetic operators, comparison operators, and annotated JDK methods of the projects. The SLOC column counts the number of non-comment, non-blank lines of code. The Arith column counts uses of all arithmetic operators. The Comp column counts uses of all numeric comparison operators. The Time column counts uses of `System.currentTimeMillis()` and `System.nanoTime()`. The Thread column counts uses of `Thread.sleep()`. Finally, the Math column counts uses of all Math trigonometry methods.

projects, due to bit-rot, lack of documentation, etc.). In total, we selected eight projects. Some projects are small but they perform interesting arithmetic operations. Fig. 12 summarizes the size and purpose of each project, together with the number of used arithmetic operators, numeric comparison operators, and annotated JDK methods. The size of the projects ranges from 538 to 168983 lines of source code, totalling 234251 lines.

5.1 RQ1: What are the benefits and trade-offs of using *PUnits* versus existing Java unit libraries?

We discuss the benefits and trade-offs from three aspects: error detection, program execution performance (time and memory), and features.

In this RQ, we focus on analyzing *PUnits*'s effects on the GasFlow project, the only project that uses a unit library. GasFlow uses the JScience library, or JSR 275/363 [Dautelle and Keil 2010; Dautelle et al. 2016], which is one of the most popular unit libraries for Java. We replace uses of JScience unit wrapper classes with *PUnits* specifications and primitive types and run modular type checking on the annotated code.

The JScience API uses generics to provide unit type-safety. JScience uses the `Amount<Quantity`[2]`>` class for storing the exact `Unit`[3] and performing arithmetic operations with the units. To replace the abstract data types with primitives, we create helper class `UnitsTools`, which stores a list of units that are represented using annotated primitives with a value of 1 (eg. `@m int m = 1`). `UnitsTools` also contains conversion methods that are used to replace unit conversion functions in JScience when it is possible to determine the type to be converted statically. All the unit conversions that appear in the GasFlow project can be determined statically. A detailed description of the JScience library and a breakdown of the units and functionalities replaced are described in Sec. C of the appendix. Fig. 13 shows an example of a converted function.

In total, 510 JScience method invocations and 503 variables containing 17 dimensions and 22 units are replaced with 647 *PUnits* qualifiers and 242 `UnitTools` uses.

5.1.1 Error Detection. After replacing units wrappers with *PUnits* specifications and primitive types, we used *PUnits* in type checking mode to see whether there are any units errors. We found

[2]Part of the `javax.measure` package, provides dimension handling.
[3]Part of the `javax.measure` package, provides unit handling.

```
1  Amount<Length> bad() {
2      Amount<Length> l;
3      l = (Amount<Length>) valueOf(1,METER).times(valueOf(1,METER));
4      Amount a = valueOf(1, METER).times(valueOf(1, METER));
5      return a;
6  }
7
8  @m double good() {
9      double l = (@m double) 1*UnitsTools.m * 1*UnitsTools.m; // Error
10     double a = 1*UnitsTools.m * 1*UnitsTools.m;
11     return a; // Error
12 }
```

Fig. 13. A simple example comparing the difference between JScience and *PUnits*.

three units errors, whereas JScience failed to detect them. The reason why JScience cannot detect these errors is due to the use of casting and raw types. The original GasFlow contains 130 raw unit data types and 51 unit casts. Line 3 in Fig. 13 illustrates an illegal cast of an area to a length. The return type of `times()` is `Amount<?>`, because JScience cannot statically express the return unit. Developers are therefore required to add casts that cannot be checked statically, to make the result usable. Similarly, on line 4, a raw type is used to circumvent this weakness. The cast from a wildcard to a type produces an unchecked warning, as does the usage of a raw type. 181 such warnings are produced in GasFlow. Making matters worse, no runtime exception is raised within method `bad()`, as all method invocations and casts are valid. However, even though the type argument is incorrect, the unit is still preserved dynamically. If we try to read the return value of `bad()` in a unit that is in the dimension of the signature on line 1, such as meter or kilometer, a runtime exception will be raised. As such invocations can happen a long time after the call to `bad()`, it is difficult to pinpoint where the incorrect `Amount` object came from.

With *PUnits*, this kind of casting will not be allowed since `m` is not a subtype of $\mathtt{m}^2$. Not only that, with *PUnits*, such casts are not needed. Variable *a* has type $\mathtt{m}^2$ and it is not a subtype of `m`. An invalid cast error will be issued on line 9 and an invalid return type will be issued on line 11. *PUnits* is able to detect errors early and it pinpoints the exact location of the problem.

5.1.2 Execution Time & Memory Consumption. We executed the GasFlow project by invoking the main method of the program. The program calculates and generates data points at every time step over a fixed amount of time. The time step was originally set to 1 second. To better analyze its performance, we decrease the time step to 1 millisecond to increase the program execution time. This case study is performed using a laptop with Intel i7-6700HQ 2.60 GHz four-core CPU and 16GB DDR4 RAM, running 64-bit Ubuntu 18.10.

The average execution time is 3966ms when we are using JScience, and 3367ms when using *PUnits* and all the abstract data types are replaced with primitives. The execution time has significantly reduced by 15.1%. The execution time is measured using Java's System library.

The average memory consumption is 794 kBytes when we are using JScience, and 697 kBytes when using *PUnits* with primitive types. The memory consumption has significantly reduced by 12.2%. The memory consumption is measured using Java's Runtime library.

The performance overhead of `UnitsTools` is insignificant compared to JScience, as all of the `UnitsTools` operations are using primitives. As *PUnits* is a static analysis tool, it will not add any overhead to the program during runtime and reap the benefits of using primitives, unlike JScience which uses abstract data types to keep track of units.

5.1.3 Features. GasFlow contains one heterogeneous method, which accepts or returns a range of dimensions that differ in its specific class or type argument. For this method, *PUnits* loses the unit of that variable, while JScience preserves the unit dynamically through abstract data types. Nevertheless, we believe it is good coding practice to avoid type-unsafe heterogeneous methods when possible, as heterogeneous methods increase the risk of a runtime exception. *PUnits* enforces this programming practice.

PUnits uses primitives instead of abstract data types, which also prevents problems with null values. As a value should always have a unit, we do not see this as a restriction. One simple way to allow the primitives to have a null-like behaviour is by creating a boolean flag for each variable to indicate if a variable has been initialized or not. In GasFlow, there are five variables that required this work-around.

It is possible to combine *PUnits* with a units library, by annotating the abstract data types of the units library with *PUnits* specifications to gain the benefits of both static error detection and dynamic library features. However, this will lose the performance benefits of using primitive types. Overall, for projects that require minimum uses of dynamic unit computations, *PUnits* is a better alternative than JSR 275/363.

5.2 RQ2: Can *PUnits* prevent units-related errors in real-world projects?

PUnits is able to type check all eight projects. In total, 106 errors were issued as the projects were unannotated and flows were detected which propagate units from annotated methods to defaulted `@Dimensionless` method parameters or returns. These errors are expected from unannotated projects.

The Firefox Android code at the commit where the error from Fig. 1 was introduced no longer builds, due to many of the project's dependencies no longer being available. We run *PUnits* on a minimized snippet extracted from the projects, which reveals the reported error presented in Fig. 1.

GasFlow is fully annotated, originally with JScience units types, which we then translated to *PUnits* specifications. Therefore, we only ran modular type checking on it, revealing three errors. Two of these errors are related. Function `computeSiamCoefficient` is declared as dimensionless. However, the return type is $m^{-1}s^{-2}$ and an invalid return error is issued. This function is then invoked by a function that requires the return type to be $m^{-1}s^{-2}$, but as `computeSiamCoefficient` is declared as dimensionless, another return type error is issued. We changed the return type to $m^{-1}s^{-2}$ to fix these two errors. The other error is related to function `getReynoldsNumber`. A Reynolds Number [Benson 2014] is a dimensionless value, but the function returns type m^3/kg. This function is missing the density (volume per weight) variable to give the correct Reynolds Number. This error causes all future computations in this project to be incorrect.

Daikon at the commit containing both errors [Daikon 2003, 2004] cannot be compiled successfully using modern versions of the tools, as the newer versions generated syntactically invalid source code. We decided to insert the errors back into the current source code to see whether or not *PUnits* can detect them. The file containing the first error [Daikon 2003] no longer exists, so only the second error [Daikon 2004] is inserted. We manually annotated the project and type checked the program. *PUnits* successfully detected the second error [Daikon 2004] and issued an argument incompatible error. We fully annotated the project with 27 scientific units and can now guarantee that these kinds of errors will not be introduced again.

For the other six projects with no known errors, we use *PUnits* to infer and fully annotate the projects. *PUnits* is able to infer and annotate units for two of the projects without modifications. The other four projects reached unsatisfiable constraints in inference, and we analyzed them further to determine the root issues. The findings are discussed in Sec. 5.2.1 below. For three of the

	PUnits			SMT				Inferred & Annotated Units						
Project	α	E	Σ	Bool	Int	Assert	Soft Assert	$\top$	D	ms	ns	deg	rad	rad^{-1}
exp4j	1438	1	1535	2876	1438	8629	2785	UNSAT						
imgscalr	835	4	1018	1670	3340	3597	1768	1	536	12	0	3	0	0
jblas	17506	3	22812	35012	52518	119940	39398	UNSAT						
JLargeArrays	12595	3	14156	25190	37785	78514	25322	UNSAT						
jReactPhysics3D	9498	2	15839	18996	18996	73774	23352	UNSAT						
react	12693	3	18092	25386	38079	59069	28392	0	8545	0	1	0	5	3
Total	54565		73452	109130	152156	343523	121017	1	9081	12	1	3	5	3

Fig. 14. Annotate mode results for the projects. Columns α and Σ show the number of constraint variables and constraints generated by *PUnits*. Column E shows the number of exponents encoded for each constraint variable. Column Assert shows the number of (mandatory) formulas encoded for SMT and MaxSMT. Column Soft Assert shows the number of breakable formulas encoded for MaxSMT. The units shown are the solutions given by *PUnits* in annotate mode. Note that $\bot$ did not appear in the solutions for any of the projects. D stands for `@Dimensionless`.

	PUnits			SMT				Inferred & Annotated Units						
Project	α	E	Σ	Bool	Int	Assert	Soft Assert	$\top$	D	ms	ns	deg	rad	rad^{-1}
exp4j	1503	1	1493	3006	1503	5793	2828	0	974	0	0	0	17	0
jblas	17537	3	22844	35074	52611	79773	39441	0	11793	4	6	0	35	0
jReactPhysics3D	9499	2	15850	18998	18998	48568	23355	0	5701	0	1	0	0	0
Total	28539	6	40187	57078	73112	134134	65624	0	18468	4	7	0	52	0

Fig. 15. Annotate mode results for the patched projects. The number of constraint variables, constraints, SMT variables and formulas vary slightly compared to the numbers reported in Fig. 14. This is due to the code changes introduced in the patches.

unsatisfiable projects, we addressed the diagnosed issues and subsequently inferred and annotated units. Fig. 14 summarizes the number of variables and constraints generated for the projects, the number of SMT variables and formulas encoded for the constraint system, and the units inferred in annotate mode. The annotate mode results for the three patched projects are shown in Fig. 15.

We manually inspected the annotate mode results. The majority of the scientific units are inferred for the types of variables and method parameters. The single $\top$ unit is inferred for a logging method's parameter. The method is inspected and the annotation is deemed correct as the method is unit-wise agnostic. All of the `@Dimensionless` annotations are inferred for variables and parameters that do not interact directly or indirectly with the annotated JDK methods. *PUnits* does not insert a unit into source code if it matches the default unit for the corresponding type use location. The numerous `@Dimensionless` units in the results are not inserted into the source code.

The overall annotation effort is very low. Given 33 manually added JDK unit annotations, *PUnits* was able to infer and annotate 24 scientific units for the two successful projects and 63 units for the three patched projects. The number of inferred units depends on the number of annotations manually added, on how often the annotated variables and methods are used, and on how interconnected the constraints expressed over the variables are. However, note that simply adding more JDK unit annotations would not make the inference problem harder: the same set of constraints would be generated for the programs and different solutions would be found. Using the 33 JDK annotations gives a good impression of the inference approach, even if most inferred annotations are dimensionless.

5.2.1 Unsatisfiable Inference Scenarios. Projects exp4j, jblas, JLargeArrays, and jReactPhysics3D failed to infer any solutions as they contain unsatisfiable constraints. We investigated the code locations which generated the unsatisfiable constraints. The following discusses the insights we learned from the analysis and from the efforts to patch the code.

exp4j is a mathematical expression evaluator for Java. The program allows users to declare variables and specify a complex expression, including calls of math library functions, in a string, for example, `"sin(x)-log(3*x/4)"`. The program evaluates these strings and gives the user a final answer. Math library functions are abstracted through method `double apply(double... args)`. The functions are recursively applied to partial answers according to the user input, until a final answer is computed. Each implementation calls a concrete math library method, some of which we had annotated with unit annotations.

The project reached UNSAT in inference due to mismatches in the expected parameter and return units of `apply()`. Some implementations expect a `@rad` argument and others expect a `@Dimensionless` argument. As math functions return results in different units, the common abstraction requires the return type of `apply()` be $\top$. Due to the recursive application of the functions, the parameter also has to be typed $\top$. This then results in an unsatisfiable constraint set given the JDK math annotations.

Fundamentally, the dynamic evaluation of the input string prevents static checking of the unit annotations. Encapsulating values in this program using JSR 363's units API [Dautelle et al. 2016] would check the units of values at run-time.

We patched this project by always treating computed answers as `@Dimensionless` and suppressing mismatches when evaluating the expression string. The inferred units annotate the trigonometry functions with a parameter or return type of radians.

jblas is a matrix library for Java. It provides a number of floating point matrix abstractions and defines mathematical functions for matrices. In particular, classes `DoubleMatrix` and `FloatMatrix` each define the overloaded methods `get(i)` and `put(i, v)` which returns or inserts a value at index `i` in the matrix, traversing row-wise first. Internally, the classes store matrix values in a one-dimensional array. Class `MatrixFunctions` defines methods to perform in-place updates to the matrix.

The project reached UNSAT because the operations read and write to the same array even if the values have different units. Although the numeric data type of the array is fixed, the array is used in a unit-wise heterogeneous way in which there is no possible typing with a unit. Users of this library will need to be careful with the order of using the methods. To solve this fundamental issue, the library would need to be rewritten to construct and return new matrices with different units instead of performing in-place updates.

We introduced a simpler, non-ideal, patch by always storing `@Dimensionless` values in the array, and casting the units as needed before they are passed to a math method. The inferred units annotate the return types and parameter types of the trigonometry methods in `MatrixFunctions` with radians, and the variables in a timer class with time units.

JLargeArrays defines arrays that can store up to 2^{63} elements. This project reached UNSAT through two different scenarios.

For the first scenario, the project uses low-level unsafe methods provided by the JDK to store and retrieve array values in order to support 64-bit indexing. Operations within the project can store `@rad` values into memory. If unannotated, JDK and external library methods are treated as `@Dimensionless` in inference mode. The flow of `@rad` into the unannotated library methods causes an unsatisfiable typing. Six additional $\top$ annotations are added to the low-level JDK method parameters to focus the inference on detecting problems in the project's code.

For the second scenario, the project defines superclass `LargeArray` which declares common methods such as `Object get(long i)` and `void set(long i, Object v)` to provide access to array elements. A method converts arrays from one subclass to another by repeatedly executing the code `out.set(i, src.get(i))` to copy values. The project reached unsat in inference as some subclasses return `@rad` values for `get()`, and some subclasses expect `@Dimensionless` values passed to `set()`. The conversion method should also be refactored to be precise in terms of the types of arrays it reads from and writes to, so that a flow of values with incompatible units does not occur.

We did not fix this project as handling the `get()` and `set()` methods correctly requires a significant refactoring.

jReactPhysics3D is a 3D physics engine. In one of its methods, a raw `Iterator` was used to iterate through a collection of integer values. *PUnits* assumes the raw type `Iterator` is actually `Iterator<@UnitsTop Object>`, applying the same conservative assumptions that Java applies. The project reached UNSAT in inference because of a flow of a value obtained from this iterator into a parameter that expects a `@Dimensionless` value. Declaring the iterator with a concrete type argument fixes the problem. The inferred nano-second unit annotates a variable in a timer class.

Overall, two reasons cause a project to reach UNSAT in inference mode:

(1) Presence of units errors.
Projects exp4j, jblas, and JLargeArrays do not make any attempts to prevent a user from mistakenly passing values of one unit into a function that expects a different unit. These errors can lead to serious failures if the libraries were used incorrectly in mission-critical systems. *PUnits* helps identify such fundamental design issues and forces developers to restructure their applications in a way that allows the safe use of units.

(2) Insufficient annotations for library methods.
Applications have large dependencies on binary-only libraries, like the JDK. *PUnits* can infer annotations for the available source code, but requires manual annotations for binary-only dependencies. *PUnits* can use optimistic defaults, for example by assuming that unannotated method signatures have $\top$ receiver and parameter types, and $\bot$ return types. Optimistic defaults can temporarily help a developer pinpoint whether the problem is in their code or is due to an unannotated API. Manual annotation efforts for libraries can be re-used to type check and infer units in other projects utilizing the same APIs.

PUnits inferred 87 scientific units and generated well-specified applications for project exp4j, imgscalr, jblas, and jReactPhysics3D. The annotations are available for human inspection to ensure specification correctness. In summary, *PUnits* is able to both detect and prevent unit related errors in real-world projects.

5.3 RQ3: What is the compilation overhead of *PUnits* in each of its three modes?

This case study is performed using a commodity desktop (Intel i7-8700K 3.70 GHz Six-Core CPU, 32GB DDR4 RAM, 256GB SATA SSD) running 64-bit Ubuntu 18.10.

Fig. 16 reports the arithmetic mean of the execution times of *PUnits* in each of its three modes across 5 executions for six projects. The projects take on average 2.92 sec to compile (17.53 sec total) using the OpenJDK 8 compiler, and on average 12.10 sec to type check (72.62 sec total) using *PUnits* in type check mode. The approximately 4.14x overhead is consistent with other type systems developed using the Checker Framework [Chen and Dietl 2018; Dietl et al. 2011a].

In inference mode, the projects took 396.37 sec (7 mins) to check and infer a type-safe solution. The overhead is approximately 5.5x compared to type check mode. For small projects, *PUnits* takes

			Inference				Annotate	
Project	**OpenJDK 8 Compile**	**Type Check**	Encode	Solve	UNSAT Encode	UNSAT Solve	Encode	Solve
exp4j	1.66	5.00	1.19	0.18	1.66	0.23	1.56	2.91
imgscalr	1.62	4.04	0.80	0.46	N/A		1.14	9.59
jblas	7.62	18.60	16.79	2.45	16.51	7.11	17.26	3883.10
JLargeArrays	2.12	13.23	11.61	46.66	11.66	32.42	No Patch	
jReactPhysics3D	2.49	13.00	10.27	0.99	10.05	5.93	10.26	280.19
react	2.01	18.75	11.80	345.63	N/A		15.06	990.04
Total	17.53	72.62	52.45	396.37	39.87	45.70	45.27	5165.84

Fig. 16. Performance of *PUnits* running in the three modes. All values are in seconds. OpenJDK 8 compilation times for the projects are presented for comparison to type check mode performance. The type check times reported are for checking the entire project. The Encode and Solve columns, respectively, report the time taken by *PUnits* to encode a constraint system into SMT formulas, and for Z3 to solve the formulas. The Encode column does not count the time required to traverse the AST to generate constraints.

more time to generate SMT encodings than it takes for the solver to solve the constraints. For large projects such as react, the major performance bottleneck is the SMT solver.

In annotate mode, successful projects take 5165.84 sec (86 mins) to infer precise units. jblas and react contribute the most to the tally. The overhead is approximately 71x vs. type check mode, and 13x vs. inference mode. The time is spent by the SMT solver to optimize the solutions.

The performance of *PUnits* in type check mode enables it to be used in edit-compile-unit-test development workflows. As type check mode is modular, faster performance is expected when checking one source file at a time. The performance in inference mode for whole-program satisfiability checking enables it to be used in continuous integration workflows. The performance in annotate mode is slow, but this mode is expected to be used less frequently. The annotations inserted into source code enable subsequent uses of *PUnits* in inference or annotate modes to be executed faster, as fewer constraint variables are generated. Developers can also incrementally infer and annotate their projects, starting with core libraries.

Overall, *PUnits* performs adequately in each of its three modes. Its performance is suitable for use in a real-world software development environment. Improvements to the Checker Framework and Checker Framework Inference will improve the performance of *PUnits* and other type systems developed using the frameworks.

6 RELATED WORK

Two general approaches have been applied in prior work to add support for units to a programming language: (1) through designing abstract data types and libraries to encapsulate numbers with units, (2) through modifying a language to add units syntax and static analysis.

6.1 Encapsulation Approach

The encapsulation approach uses the existing type system of a language to perform limited static analysis, with the capability to perform fully run-time-based unit analysis for units and quantities that are given as run-time inputs. The design and features of 38 different units libraries are extensively compared in a recent paper [Bennich-Björkman and McKeever 2018]. Using such libraries incurs additional performance and memory overhead. For example, using boxed number types for numeric computations in Java is 3x slower and uses 3x more memory compared to using primitive types [Melzer 2015]. Units of measurement libraries exist for all programming languages, from

Ada [Gehani 1977, 1985] and C++ [Schabel and Watanabe 2010] to Fortress [Allen et al. 2004] and .NET [Larsen 2018], and many, many others.

JSR 275/363 [Dautelle and Keil 2010; Dautelle et al. 2016] is the most popular option for Java. It defines an API for units of measurement and provides a reference implementation for SI units. The benefits and trade-offs of using *PUnits* versus such libraries are discussed in Sec. 5.1.

6.2 Static Approaches

The static approaches in prior work differ in how units are internally represented, in how typing constraints are solved, and in terms of usability.

Osprey is a constraint-based units type checker for C [Jiang and Su 2006]. *PUnits*'s representation of units is similar to Osprey's. Osprey introduces a units language whereby compound units are represented as products and inverses of SI base units, a "dimensionless" unit, and constant factors. *PUnits*'s representation compresses Osprey's representation, and stores the exponents. Osprey does not allow developers to utilize a different set of base units, and does not have a top or bottom type. There is no subtyping between units. Osprey's minimized constraint set is translated into a system of linear equations and then solved using Gaussian elimination. We also implemented a Gauss-Jordan elimination solver for the integer subsets of the constraints. However, the initial performance results of this solver did not look promising and we did not pursue the effort further.

F#'s units type system [Kennedy 2009, 1997; Wlaschin 2012] represents units as symbols or products of symbols, each with an optional integer exponent. The set of symbols can be declared by the developer, however there is no support for representing prefixed units in terms of their base units, as prefixed units are considered distinct unit symbols. Consequently, a value with unit `<kN/km>` cannot be assigned to a variable with unit `<N/m>` in F# whereas it is permitted in *PUnits*. F# infers under-constrained type variables as polymorphic unit types, the most general typing in F#. In *PUnits*, the most general type is $\top$, which is not a useful annotation to insert into source code. *PUnits* chooses to infer the most precise type, and prefers to infer `@Dimensionless` for under-constrained type variables. Functions that are intended to be polymorphic must be explicitly annotated in F#. F# allows polymorphic functions to express more rich relationships such as polymorphic multiplication and division where the result unit is a function of two or more polymorphic units given as the parameters. *PUnits* currently supports a less expressive form of polymorphism, but scaling the current design to support F# style polymorphism is not difficult, as relationships between generic units can be explicitly declared via a set of meta-annotations, or inferred by treating the typing constraints generated for a method body as a set of relationships and minimizing the set. We did not need F# style polymorphism in the case studies, but plan to extend our implementation as future work. F#'s annotation burden is improved upon in a type system for Fortran [Hangal and Lam 2009; Orchard et al. 2015]. *PUnits* inserts all inferred non-default units into source code, reducing the annotation burden for subsequent type checking or inference.

A dimensional analysis system for Simulink [Owre et al. 2012] expresses dimensions as products of SI base dimensions, each with an integer exponent. Dimensional consistency can be enforced through this representation. However, unit-wise consistency cannot be enforced: an expression of `1 km + 1 m` would not result in any errors as both are in the dimension of length. A proper calculation would require either unit to be converted prior to the addition.

A units type system for the B language [Krings and Leuschel 2013] represents units as products of SI units where each base unit has a prefix and an exponent. Some derived units can be expressed via multiple representations in this design. For example, a kilo-newton can be defined as both `Mg * m * s`$^{-2}$ and `kg * km * s`$^{-2}$ with corresponding triples. This leads to combinatorial explosion of representations since multiplication and division produce new units as a function of the units of its

two arguments. *PUnits* extracts all prefixes into one constant, forcing each unit to have a unique normalized representation and avoids the combinatorial explosion problem.

CPF[UNITS] [Hills et al. 2012] defines a units analysis policy, annotation language, and specification that plugs into the C Policy Framework (CPF) to debug and verify C programs. Units are represented as products of base units, each with an exponent. The set of base units is customizable and can be given long and short names. The system is modular, and can infer units for local variables. However, it does not perform whole-program inference.

Type Qualifiers as Composable Language Extensions [Carlson and Van Wyk 2017] proposes adding pluggable type checkers to the *ableC* [Kaminski et al. 2017] language as grammar and type checking extensions, including a units of measurement system. In *ableC*, units are not organized in a type lattice and there is no concept of `@UnitsTop` or `@UnitsBottom` types. *ableC*'s units type system is implemented exclusively for SI units, and does not extend support to other units of measurements without further extending the grammar and implementing the extension. The pluggable type systems of *ableC* lack type inference capabilities.

7 CONCLUSION AND FUTURE WORK

This paper presents *PUnits*, a pluggable type checking and whole-program type inference system for units of measurement. Three usage modes provide fast modular type checking, whole-program satisfiability checking, and whole-program inference of precise annotations that can be used for further analyses. *PUnits* is formalized and proven sound. *PUnits* is evaluated in Java, showing *PUnits* can reap the benefits of using primitives over abstract data types, and prevent units-related errors in real-world projects. *PUnits* requires only a few manual annotations to infer units, and manual effort can be reused when annotating additional projects. *PUnits* strikes a novel balance between expressiveness, inference complexity, and annotation effort. It is a practical tool for improving the quality of software and it can help developers discover insights into how their code and APIs are used.

The work in this domain is not yet finished. The type inference approaches employed thus far either assume that unannotated method parameters and returns are polymorphic over units, or require explicit annotations to indicate polymorphism. No approach to date can infer that a method should be polymorphic over units. *PUnits* is the first units type system to adapt the concept of receiver-dependent types. *PUnits* currently does not infer whether a method parameter or return type should be receiver dependent over units. Finding an efficient encoding is left as future work.

Implementing prefix p using floating-point arithmetic with safe comparisons instead of the current base-10 prefix would allow *PUnits* to support Imperial units more easily. *PUnits* performance in optimizing inference mode also warrants further investigation. We remain optimistic in this regard: SMT solver performance has been steadily improving, we can explore different constraint encodings, and we can employ alternative solvers.

We plan to extend support for a more expressive form of parametric polymorphism which will enable support for methods such as `myDivide(x, y)`, which returns a unit computed as a function of the method arguments' units.

Units of measurements are important in many software domains. *PUnits* provides an expressive and precise analysis with strong static guarantees. The system is easy to use and can make unit-correct code a reality.

REFERENCES

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L Steele Jr. 2004. Object-oriented units of measurement. *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)* 39, 10 (2004), 384–403.

Anonymous-Authors. 2020. pUnits formalization and soundness proof. http://omitted.per.double.blind.reviewing

Oscar Bennich-Björkman and Steve McKeever. 2018. The next 700 unit of measurement checkers. In *Software Language Engineering, (SLE)*. 121–132. https://doi.org/10.1145/3276604.3276613
Tom Benson. 2014. Reynolds Number. https://www.grc.nasa.gov/WWW/BGH/reynolds.html
Mishap Investigation Board. 1999. Mars Climate Orbiter Mishap Investigation Board Phase I Report.
Gilad Bracha. 2004. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, Vol. 4.
Travis Carlson and Eric Van Wyk. 2017. Type qualifiers as composable language extensions. In *Generative Programming: Concepts & Experiences (GPCE)*. ACM, 91–103.
Charles Zhuo Chen and Werner Dietl. 2018. Don't Miss the End: Preventing Unsafe End-of-File Comparisons. In *NASA Formal Methods Symposium*. Springer, 87–94.
The Coq Development Team. 2004. The Coq Proof Assistant. https://coq.inria.fr
Daikon. 2003. Fix bug that printed milliseconds, not seconds. https://github.com/codespecs/daikon/commit/a421b229bbed7da94bd3636c0eb03b716d151d5d
Daikon. 2004. Multiply dkconfig_compile_timeout by 1000 to convert from seconds to milliseconds. https://github.com/codespecs/daikon/commit/1c74cbe4e1fe74d2ef2fc30c577482786d0aca5c
Jean-Marie Dautelle and Werner Keil. 2010. JSR 275: Units Specification API. https://jcp.org/en/jsr/detail?id=275
Jean-Marie Dautelle, Werner Keil, and Leonardo Lima. 2016. JSR 363: Units of Measurement API. https://jcp.org/en/jsr/detail?id=363
Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.
Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. 2011a. Building and using pluggable type-checkers. In *International Conference on Software Engineering (ICSE)*. ACM, 681–690.
Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 28–53.
Werner Dietl, Michael D Ernst, and Peter Müller. 2011b. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 333–357.
Michael D Ernst, Alex Buckley, Werner Dietl, Doug Lea, Srikanth Sankaran, and Oracle. 2012. JSR 308: Annotations on Java Types. https://jcp.org/en/jsr/detail?id=308
Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
Narain H. Gehani. 1977. Units of measure as a data attribute. *Computer Languages* 2, 3 (1977), 93–111.
Narain H. Gehani. 1985. Ada's derived types and units of measure. *Software: Practice and Experience* 15, 6 (1985), 555–569.
Sudheendra Hangal and Monica S Lam. 2009. Automatic dimension inference and checking for object-oriented programs. In *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 155–165.
Mark Hills, Feng Chen, and Grigore Roşu. 2012. A rewriting logic approach to static checking of units of measurement in C. *Electronic Notes in Theoretical Computer Science* 290 (2012), 51–67.
Lingxiao Jiang and Zhendong Su. 2006. Osprey: a practical type system for validating dimensional unit correctness of C programs. In *International Conference on Software Engineering (ICSE)*. ACM, 262–271.
Wesley Johnston. 2012. Close by swipe velocity checks are wrong. https://bugzilla.mozilla.org/show_bug.cgi?id=765069
Nahid Juma, Werner Dietl, and Mahesh Tripunitara. 2020. A computational complexity analysis of tunable type inference for Generic Universe Types. *Theoretical Computer Science* 814 (2020), 189–209.
Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 98.
Andrew Kennedy. 2009. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*. Springer, 268–305.
Andrew J Kennedy. 1997. Relational parametricity and units of measure. In *Principles of Programming Languages (POPL)*. ACM, 442–455.
Sebastian Krings and Michael Leuschel. 2013. Inferring physical units in B models. In *Software Engineering and Formal Methods (SEFM)*. Springer, 137–151.
Andreas Gullberg Larsen. 2018. Units.NET. https://github.com/angularsen/UnitsNet
Jens Melzer. 2015. Autoboxing Performance. https://effective-java.com/2015/01/autoboxing-performance
Dominic Orchard, Andrew Rice, and Oleg Oshmyan. 2015. Evolving Fortran types with inferred units-of-measure. *Journal of Computational Science* 9 (2015), 156–162.
Sam Owre, Indranil Saha, and Natarajan Shankar. 2012. Automatic dimensional analysis of cyber-physical systems. In *International Symposium on Formal Methods*. Springer, 356–371.

Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 201–212.

Matthias C. Schabel and Steven Watanabe. 2010. Boost.Units. http://www.boost.org/doc/libs/1_65_1/doc/html/boost_units.html

Scott Wlaschin. 2012. Units of measure - Type safety for numerics. https://fsharpforfunandprofit.com/posts/units-of-measure

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94.

$$\text{Variable Declarations}: \boxed{\Gamma \vdash \overline{vd} : \Gamma', \Sigma} \quad \frac{\Gamma \vdash vd : \Gamma_{vd}, \Sigma_{vd} \quad \Gamma_{vd} \vdash \overline{vd} : \Gamma_{\overline{vd}}, \Sigma_{\overline{vd}}}{\Gamma \vdash vd; \overline{vd} : \Gamma_{\overline{vd}}, \Sigma_{vd} \cup \Sigma_{\overline{vd}}} \quad \frac{}{\Gamma \vdash [] : \Gamma, \emptyset}$$

$$\text{Statements}: \boxed{\Gamma \vdash \overline{s} : \Sigma} \quad \frac{\Gamma \vdash s : \Sigma_s \quad \Gamma \vdash \overline{s} : \Sigma_{\overline{s}}}{\Gamma \vdash s; \overline{s} : \Sigma_s \cup \Sigma_{\overline{s}}} \quad \frac{}{\Gamma \vdash [] : \emptyset}$$

Fig. 17. Constraint generation rules for the sequence forms of variable declarations $\overline{vd}$, and statements $\overline{s}$. The notation $X; \overline{X}$ denotes a sequence of elements consisting of X followed by the sequence $\overline{X}$. The notation [] denotes an empty sequence.

$$\text{Well-formedness}: \boxed{\vdash T \text{ OK} : \Sigma} \quad \frac{}{\vdash \top \text{ OK} : \emptyset} \quad \frac{}{\vdash \bot \text{ OK} : \emptyset} \quad \frac{\forall_{i,j \in \{1 \ldots |\overline{u}|\}}\ i \neq j \implies u_i \neq u_j}{\vdash p\ \overline{u^z} \text{ OK} : \emptyset}$$

$$\frac{}{\vdash \alpha \text{ OK} : \{wf(\alpha)\}}$$

$$\text{Subtyping}: \boxed{\vdash T_1 <: T_2 : \Sigma} \quad \frac{}{\vdash T <: T : \emptyset} \quad \frac{}{\vdash T <: \top : \emptyset} \quad \frac{}{\vdash \bot <: T : \emptyset}$$

$$\frac{T_1 = \alpha_1 \vee T_2 = \alpha_2}{\vdash T_1 <: T_2 : \{subtype(T_1, T_2)\}}$$

Fig. 18. Helper judgments.

A *πUNITS* SUPPORTING DEFINITIONS

The sequence forms of the constraint generation rules for variable declarations and statements are presented in Fig. 17.

Helper judgments and functions are shown in Fig. 18. The well-formedness judgment ensures that types are well-formed. The types $\top$ and $\bot$ are always well-formed. A type declaring a scientific unit of the form $p\ \overline{u^z}$ is well-formed if the base units present in the declaration are pair-wise distinct. A well-formedness constraint is generated for constraint variables. The subtyping judgment is reflexive. All types are subtypes of $\top$ and supertypes of $\bot$. A subtype constraint is generated when one or both of the types are constraint variables.

Fig. 19 shows the small-step operational semantics. A program starts execution with an empty frame, and evaluates all of its variable declarations and then all of its statements. A variable declaration $T\ v = z$ adds a mapping between the variable v to its statically declared type T and labeled value $T\ z$ to the frame. An assignment statement $v = e$ will reduce e if e can take a step. With e fully reduced to a labeled value $T_l\ z$, the statement will update the frame by mapping v to $T_l\ z$ as its new value if and only if the label type T_l is a subtype of the variable T_v. A variable read expression v will retrieve and return the labeled value stored in the frame for the variable. An arithmetic expression $e_1\ op\ e_2$ will first reduce e_1 to $T_1\ z_1$ and then reduce e_2 to $T_2\ z_2$. Finally, a result value $T_3\ z_3$ is computed where $T_3 = T_1\ op\ T_2$ and $z_3 = z_1\ op\ z_2$ for some given arithmetic operation op.

The normal forms of *πUnits* are presented in Fig. 20. An expression is a normal form if it is a value. A sequence of statements is in normal form if all statements have been evaluated, resulting in an empty sequence. Similarly, a sequence of variable declarations is in normal form if all declarations have been evaluated. A program is in normal form if all of its variable declarations and statements have been evaluated.

$$\text{Stack Frame}: F = \{\overline{v \rightarrow \{T_v, T_l, z\}}\}$$

$$\text{Program}: \boxed{F,\ P \Rightarrow F',\ P'} \quad \frac{F,\ \overline{vd} \Rightarrow F',\ \overline{vd'}}{F,\ \overline{vd}\ \overline{s} \Rightarrow F',\ \overline{vd'}\ \overline{s}} \quad \frac{F,\ \overline{s} \Rightarrow F',\ \overline{s'}}{F,\ \emptyset\ \overline{s} \Rightarrow F',\ \emptyset\ \overline{s'}}$$

$$\text{Variable Declarations}: \boxed{F,\ \overline{vd} \Rightarrow F',\ \overline{vd'}} \quad \frac{}{F,\ T\ v = z\ ;\overline{vd} \Rightarrow F\{v \rightarrow \{T, T, z\}\},\ \overline{vd}}$$

$$\text{Statements}: \boxed{F,\ \overline{s} \Rightarrow F',\ \overline{s'}} \quad \frac{VarType(F, v) : T_v \quad \vdash T_l <: T_v : \emptyset}{F,\ v = T_l\ z\ ;\overline{s} \Rightarrow F\{v \rightarrow \{T_v, T_l, z\}\},\ \overline{s}} \quad \frac{F,\ e \Rightarrow e'}{F,\ v = e\ ;\overline{s} \Rightarrow F,\ v = e'\ ;\overline{s}}$$

$$\text{Expressions}: \boxed{F,\ e \Rightarrow e'} \quad \frac{}{F,\ v \Rightarrow VarValue(F, v)} \quad \frac{T_3 = T_1\ op\ T_2 \quad z_3 = z_1\ op\ z_2}{F,\ T_1\ z_1\ op\ T_2\ z_2 \Rightarrow T_3\ z_3}$$

$$\frac{F,\ e_1 \Rightarrow e_1'}{F,\ e_1\ op\ e_2 \Rightarrow e_1'\ op\ e_2} \quad \frac{F,\ e_2 \Rightarrow e_2'}{F,\ T_1\ z_1\ op\ e_2 \Rightarrow T_1\ z_1\ op\ e_2'}$$

Fig. 19. Small-step operational semantics. The notation $X;\overline{X}$ denotes a sequence of elements consisting of X followed by the sequence $\overline{X}$. $VarType(F, v)$ is a function which returns the static type T_v of variable v if the variable exists in the frame. $VarValue(F, v)$ is a function which returns the labeled value $T_l\ z$ stored by variable v if the variable exists in the frame.

$$\text{Expression}: \boxed{\vdash e\ \text{NF}} \quad \frac{}{\vdash T\ z\ \text{NF}} \qquad \text{Statement}: \boxed{\vdash \overline{s}\ \text{NF}} \quad \frac{}{\vdash \emptyset\ \text{NF}}$$

$$\text{Variable Declaration}: \boxed{\vdash \overline{vd}\ \text{NF}} \quad \frac{}{\vdash \emptyset\ \text{NF}} \qquad \text{Program}: \boxed{\vdash P\ \text{NF}} \quad \frac{\vdash \overline{vd}\ \text{NF} \quad \vdash \overline{s}\ \text{NF}}{\vdash \overline{vd}\ \overline{s}\ \text{NF}}$$

Fig. 20. Normal forms.

$$\frac{\forall v \in F\ \exists T_v, T_l, z: \quad VarType(F, v) = T_v \quad VarValue(F, v) = T_l\ z \quad \vdash T_l <: T_v : \emptyset}{\vdash F\ \text{OK}}$$

Fig. 21. Well-formed frame judgment. $VarType(F, v)$ is a function which returns the static type T_v of variable v if the variable exists in the frame. $VarValue(F, v)$ is a function which returns the labeled value $T_l\ z$ stored by variable v if the variable exists in the frame.

B THEOREMS, LEMMAS AND SUMMARIES OF THEIR PROOFS IN COQ

The well-formed frame judgment $\vdash F$ OK, presented in Fig. 21, captures the conditions required to not have a *STUCK* program. If any operation causes a frame to not be well-formed, then the program evaluates to *STUCK*. In the proofs, a frame is assumed to be well-formed before any term reductions. We prove that the frame produced by a reduction remains well-formed.

Function $ExtendGamma(\Gamma, P)$ returns a Γ' consisting of Γ updated with the first variable declaration $v = T\ z$ of program P if it exists (that is $\Gamma' = \Gamma \cup \{v \rightarrow T\}$), otherwise it returns Γ unchanged.

We now present the progress and preservation theorems for *πUnits* and discuss the key supporting lemmas proven in Coq.

Theorem B.1 (Progress). $\vdash P : \Gamma, \Sigma \wedge \vdash F\ \text{OK} \implies$ `normal form` $P \vee \exists F', P' \cdot F,\ P \Rightarrow F',\ P'$.

Theorem B.2 (Preservation). $\Gamma \vdash P : \Gamma' \wedge \vdash F \text{ OK } \wedge F,\ P \Rightarrow F',\ P' \implies \vdash F' \text{ OK } \wedge ExtendGamma(\Gamma, P) \vdash P' : \Gamma'$.

Following are the supporting lemmas and summaries of their proofs.

Lemma B.3 (Expression Progress). $\Gamma \vdash e : T \wedge \vdash F \text{ OK } \implies \vdash e \text{ NF} \vee \exists\, e' \cdot F,\ e \Rightarrow e'$.

Proof: by induction on typing of e.
Case: e is a variable read expression v. Since $\Gamma \vdash e : T_v$ and $\vdash F$ OK, there exists a value $T_v\ z$ in frame F for v, and v steps to $T_v\ z$.
The other cases are easy.

Lemma B.4 (Statement Progress). $\Gamma \vdash s \wedge \vdash F \text{ OK } \implies \vdash s \text{ NF} \vee \exists\, F', s' \cdot F,\ s \Rightarrow F',\ s'$.

Proof: by induction on typing of s.
Case: s is a statement $v = e$, and $\Gamma \vdash e : T_e$ by the typing of the statement. By Lemma B.3, either e is a value $T_e\ z$ or it can reduce to e' without modifying the frame. If e is a value then $F,\ v = T_e\ z \Rightarrow F\{v \rightarrow \{T_v, T_e, z\}\},\ \emptyset$ as required. If e can be reduced to e' then $F,\ v = e \Rightarrow F,\ v = e'$ as required.
The other cases are easy.

Lemma B.5 (Variable Declaration Progress). $\Gamma \vdash vd : \Gamma' \implies \vdash vd \text{ NF} \vee \exists\, F', vd' \cdot F,\ vd \Rightarrow F',\ vd'$.

Proof: by induction on typing of vd. All cases are easy.

Lemma B.6 (Expression Preservation). $\Gamma \vdash e : T \wedge \vdash F \text{ OK } \wedge F,\ e \Rightarrow e' \implies \exists\, T' \cdot \vdash T' <: T \wedge \Gamma \vdash e' : T'$.

Proof: by induction on typing of e.
Case: e is a variable lookup expression v which reduces to $e' = T_v\ z$ for some T_v and z. Since $\Gamma \vdash v : T_v$ and $\Gamma \vdash F$ OK, T_v as retrieved from frame F must be a subtype of T_v and $\Gamma \vdash T_v\ z : T_v$ as required.
Case: e is an arithmetic expression $e_1\ op\ e_2$ for some given operation op, $\Gamma \vdash e_1\ op\ e_2 : T_1\ op\ T_2$, $\Gamma \vdash e_1 : T_1$, and $\Gamma \vdash e_2 : T_2$. We break this case down by induction on the small-step relation $F,\ e_1\ op\ e_2 \Rightarrow e'$.
Subcase: e_1 and e_2 are values $T_1\ z_1$ and $T_2\ z_2$ respectively, and the expression steps to $e' = (T_1\ op\ T_2)(z_1\ op\ z_2)$. We have $\Gamma \vdash e' : T_1\ op\ T_2$ as required.
Subcase: e_1 can step to e_1'.
Subcase: e_2 can step to e_2'.
The last two subcases are discussed in Sec. 3.5.

Lemma B.7 (Statement Preservation). $\Gamma \vdash s \wedge \vdash F \text{ OK} \wedge F,\ s \Rightarrow F',\ s' \implies \Gamma \vdash s' \wedge \vdash F' \text{ OK}$.

Proof: by induction on typing of s.
Case: s is a statement $v = e$. $\Gamma \vdash v : T_v$, $\Gamma \vdash e : T_e$, and $T_v <: T_e$ by the typing of the statement. We break this case down by induction on the small-step relation $F,\ v = e \Rightarrow F',\ s'$.
Subcase: e is a value $T_e\ z$ and the statement steps to $F\{v \rightarrow \{T_v, T_e, z\}\},\ \emptyset$ by updating the frame. $\Gamma \vdash \emptyset$ is always true. Since $T_v <: T_e$, we have $\Gamma \vdash F\{v \rightarrow \{T_v, T_e, z\}\}$ OK as required.
Subcase: e can reduce to e' without modifying the frame, and the statement steps to $F,\ v = e'$. Thus we have $\vdash F$ OK by the premise. By Lemma 3.1, there exists a $T_{e'} <: T_e$ such that $\Gamma \vdash e' : T_{e'}$, by transitivity of subtyping we have $T_{e'} <: T_v$ and $\Gamma \vdash v = e'$ as required.

The static semantics ensures that T_e is always a subtype of T_v during an assignment, which is required for a valid execution of an assignment statement.By induction on the list of statements, we can show that $\vdash F$ OK holds after executing any number of statements.

Lemma B.8 (Variable Declaration Preservation). $\Gamma \vdash vd : \Gamma' \wedge \ \vdash F$ OK $\wedge \ F,\ vd \Rightarrow F',\ vd' \implies \vdash F'$ OK $\wedge$ $ExtendGammaVariable(\Gamma, vd) \vdash vd' : \Gamma'$.

where $ExtendGammaVariable(\Gamma, vd)$ is a function which returns a gamma Γ' consisting of Γ updated with the first variable declaration $v = T\ z$ of vd if vd is not empty (that is $\Gamma' = \Gamma \cup \{v \rightarrow T\}$), otherwise it returns Γ unchanged.

Proof: by induction on typing of vd.

Case: vd is a variable declaration $T\ v' = z$ and $F,\ vd \Rightarrow F\{v' \rightarrow \{T, T, z\}\}, \emptyset$. By the typing of $T\ v' = z$ we have $\Gamma' = \Gamma \cup \{v' \rightarrow T\}$. It is easy to show that $ExtendGammaVariable(\Gamma, T\ v' = z) \vdash \emptyset : \Gamma'$. By construction, Γ and F are built up by adding one variable declaration at a time from $\overline{vd}$. Both Γ' and F' start from empty. Note that *πUnits* does not forbid duplicate declarations of variables in $\overline{vd}$. Given $\vdash F$ OK, every v in Γ must also exist in F. If $v' = v$, then F' updates F by mapping $v \rightarrow \{T, T, z\}$. If $v' \neq v$ then F' extends F with a new variable $v' \rightarrow \{T, T, z\}$. In both cases we have $\vdash F'$ OK as required.

The other cases are easy.

πUnits does not forbid duplicate variable declarations. The proof shows that regardless of whether a variable declaration replaces the type and value of a prior declaration or declares a new variable, the frame is always well-formed. By induction on the list of variable declarations, we can show that $\vdash F$ OK holds after executing any number of variable declarations.

Overall, since we can show that $\vdash F$ OK holds after executing any number of well-typed variable declarations and statements, $\vdash F$ OK always holds for a well-typed program, and a well-typed program never gets *STUCK*.

C GASFLOW REPLACEMENT DETAILS

GasFlow only uses ten functionalities within the JScience library:

- Initialization: `valueOf(double, Unit)` and `valueOf(String)` can take in a double and its corresponding `Unit` or a string in the format of `"value unit"` (e.g., `"50 m"`). 181 uses.
- Conversion: `to(Unit)` and `doubleValue(Unit)` return the value converted to the unit specific by the argument. It is required for the argument to be in the same dimension as its receiver. 81 uses.
- Arithmetic: `plus(Amount)`, `minus(Amount)`, `times(Amount)`, and `divide(Amount)` perform arithmetic operations between two Amounts. Plus and minus operations require the receiver and the argument to be in the same dimension. 236 uses.
- Comparison: `isGreaterThan(Amount)` and `isLessThan(Amount)` perform comparison operations between two Amount. 12 uses.

All the wrapper initialization are replaced with UnitsTools annotated primitives. For example, `Amount.valueOf(2, METER)` is replaced with `2 * UnitsTools.m`. All the wrapper arithmetic operations are replaced with the standard Java operations +, -, *, /. All the wrapper comparison operations are replaced with the standard Java operations > and <. These three operations are replaced using a small script with sed operations. For conversion, we manually ensured that the variable is in the unit specified by the argument by specifying the declaration to its specified unit. For example, `Amount m = amountVal.to(METER)` becomes `@m double m = amountVal` with

Dimension	Unit	Original			*PUnits*	
		Amount	Javax	String	Anno	UTools
Length	`m`	75	79	1	46	25
	`mm`		5	1	26	17
	`km`		2	0	0	1
Pressure	`Pa`	90	3	0	0	3
	`Ba`		46	3	111	31
Duration	`s`	16	24	12	14	11
	`hr`		4	0	0	48
Temperature	`K`	18	8	1	18	14
	`°C`		1	2	3	3
Mass	`g`	2	4	0	0	13
	`kg`		16	0	6	9
Angle	`rad`	0	0	0	10	0
Velocity	`m/s`	16	4	1	11	5
Area	m^2	4	0	0	1	2
Volume	m^3	12	7	0	9	59
Power	`W`	4	4	0	4	1
Mass Flow Rate	`kg/s`	8	2	0	14	0
Molar Mass	`g/mol`	5	2	2	12	0
Volumetric Flow	m^3/hr	71	0	52	49	0
Calorific Value	MJ/m^3	3	0	1	5	0
Heat Transfer Coefficient	$W/m^2/K$	3	0	0	3	0
Dimensionless	-	46	21	0	39	0
Raw Data Type	-	130	9	0	266	0
Total	-	503	220	105	647	242

Fig. 22. A summary of all the dimensions and units used in the GasFlow project. Column Amount counts the uses of `Amount<Q>`. Column Javax counts uses of `javax.measure.unit` and column String counts units represented in string format. Column Anno counts unit qualifiers after converting to use *PUnits*. The qualifiers replace the `Amount<Q>` uses in the project. Column UTools counts uses of `UnitsTools` that are used to replace `javax.measure.unit` and units in string format. Velocity can be represented as `UnitsTools.m/UnitsTools.s` and therefore is counted toward `m` and `s` instead of `m/s`.

`amountVal` now a primitive. Fig. 22 give a summary of the ranges of dimensions and units used in the GasFlow project.