Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

Improving Composability Support in User-Extensible Type Systems

拡張可能な型システムにおける型の合成機能の改善

Antoine Vinh Quang TU
ツ アントアン

Supervisor: Professor Shigeru Chiba

January 2018

Abstract

Most statically typed programming languages come with a built-in type systems providing a set of primitive types that are general enough for most purposes, but may not fit specific use cases. For that reason, there has been an interest in extensible type systems to allow users to bring their type systems to extend the host programming language's own, but those systems are not without limitations when it comes to composability, a feature that allows user-defined types to be joined to form composite types.

To that end, our proposed system brings two major contributions. Firstly, we improve the robustness of extensible type systems while maintaining flexibility by introducing composition rules restricting the otherwise unbound combinatory behavior in type systems that allow type combinations. Existing systems are either too restrictive or too liberal in allowing type combinations. Secondly, we propose an algorithm that optimizes the use of implicit conversions, which are necessary to improve the usefulness of type systems supporting composability, during type normalization.

概要

多くの静的型システムを採用しているプログラミング言語では一般なユースケースに対応するプリミティブ型があるが、そのプリミティブ型では特別なユースケースには不向きである。そのため、ユーザーが自分の型システムをホスト言語に持ち込めるようにユーザー定義型システムが検討されつつある。しかし、そのようなシステムでも型の組み立て（composability）は上手く設計されていない。

本提案手法には二つの貢献がある。一つ目として型システムの安全を保ちながらユーザーの表現力を拡大するべく、型の組み合わせを制限するcomposition ruleを提案する。従来のシステムはユーザーの表現力を制限するか制限をゆるめて静的型システムの安全を壊すかに大きく分けられる。二つ目の貢献として実行時の型変換の回数を最低限にする為のアルゴリズムを提案する。

Contents

Chapter 1

Introduction

A type system is a collection of *types* and *type rules* that formalizes the properties and allowed operations on those types [1]. The purpose of such a system is to allow the validation of programming code against those formal rules in order to detect type errors algorithmically using what is called a *type checker*. There are two main categories of type checkers: static and dynamic. With static typing, a program is type checked at compile time as part of the compilation process and the binary produced comes with a guarantee that the program will at least run coherently to the host language's type system. With dynamic typing, such check is deferred to thr runtime, with heavy use of run time type inference to determine the best-fit for each variable. Since dynamically typed programs are checked at runtime, there is no safety guarantee and the program may crash if a type error is encountered.

There has been a growing interest in recent years in static type checking, as evidenced by the advent of statically typed versions of dynamically typed programming languages like TypeScript, a statically typed subset of JavaScript. Static verification is a complex topic because the compiler needs to maintain a guarantee that any program it allows to compile satisfies the formal type rules at all time. It is nevertheless important to have static verification in some scenarios where it would not be acceptable for the program to crash whenever a type error is encountered. As such checks are performed at compilation time, static verification allows us to avoid such situations.

The built-in type system provided by a statically typed programming language has a set of primitive types that are designed to be general purpose enough to fit most use cases. However, for that reason compromises have to be made to balance the complexity of the built-in type system with the hard to predict needs of the programming language users. Namely, the built-in type system does not provide any way for programmers to define their own application semantics for static verification. For instance, it may be desirable for a *string* type to be non-nullable in some use cases, but most type systems provide a nullable instance of *string*. Supporting every possible use-case is not scalable for programming language designers as it affects maintainability. Furthermore, some application semantics might be conflicting, so it is not realistic for the language to support every such semantics out of the box. The onus is then on the programmers to include checks around their code using those primitive types to prevent errors that are not caught by the language semantics. Programmers can use techniques like object-oriented programming to define classes of values that fit their application's use-case, but those classes merely encapsulate

the built-in primitive types without changing their semantics. It is up to the programmer to introduce boundary checks and validation, the omission of which often being the cause of bugs and unexpected program behavior. Ideally, it should be up to the type system to define valid actualizations.

1.1 Extensible Type Systems

In this thesis, we focus on extensible type systems. Extensible or pluggable type systems are an approach that has been proposed to address the paucity of primitive types in the built-in static type system [2], which is a consequence of the lack of support for user-defined application semantics. It is important to note that those pluggable type systems do not overwrite the host language's own type system; they extend it by providing additional semantics that are used in combination with the host language's own. In typical implementations, users define their new types using a provided library or framework and annotate their code with the new semantics. Then, a special type checker provided by the library or framework designer first performs type checking on annotated variables against the user-defined semantics. Once that type checking is deemed successful, any annotations not belonging to the host language are removed and regular type checking by the host language's compiler can occur.

A common example of extensible type system is one that supports the usage of units of dimension such as *Length* and *Time*. Support for statically checking units of dimensions is useful to avoid type errors in special purpose applications that make extensive use of them such as physics engines or spacecrafts. Their implementation as a type system is however non-trivial as evidenced by the large number of libraries [3, 4, 5] attempting to provide support for them. Units of dimension have the property that they can be scaled (for example *milli-*, *kilo-*). Another property of units of dimension is that they can be combined to form new units of dimension. For instance, a dimension of *Length* such as *m* (meter) and a dimension of *Time* such as *sec* (second) can be combined to form a dimension of *Speed* such as *m/sec*. Expressing those properties in extensible type systems is however not trivial, as the possible combinations grow exponentially. Some extensible type systems [6] do not support type combinations, leaving it to the programmers to explicitly specify every units of dimension with limited support for transitions between them. Others [4, 5] take a more liberal approach by allowing any combination of dimensions, instantiating their definition as new combinations are encountered. The former approach results in a type system that lacks flexibility as each type is atomical, while the latter weakens the safety promises of a static type system by allowing the existence of types that may not be intended by the type system designer.

1.2 Improving Composability Support

To address those limitations, we present our extensible type system approach, called *composite type labels*. We developed a subset of the C# programming language that accepts a syntax for type annotations, that we call *type labels* and *composite type labels*. Our subset C# language is implemented via a modified version of the Roslyn compiler for C# and is fully compatible with regular C# without type annotations. Programmers can define their own type labels and their

attributes using a domain specific language (DSL) that we have provided for that purpose. The Roslyn compiler platform, via our modifications, then exposes type label information to our type checker and normalizer, which performs type checking on user-defined type annotations. Once that type checking is successfully passed, type label semantics are rewritten in pure C# and the user code is rewritten to be fully conforming to the original C# syntax.

To reconciliate flexibility with static type checking robustness, we introduce user-defined type composition rules. When defining a composite type label, users also define legal operations and operands on those composite type labels. Our type checker will use those rules to automatically perform combinations while rejecting any combination that is not explicitly defined through a composition rule.

To better handle type labels within the same dimension, we introduce user-defined implicit conversion rules. These rules are defined in a similar fashion to composition rules, and our type checker uses them to derive all possible conversion paths between type labels of the same dimension. Furthermore, to optimize the conversion behavior, we introduce an algorithm to reduce the number of implicit conversions performed during type normalization.

1.3 Contributions

In this thesis we present two major contributions. Firstly, we propose a mechanism to improve the robustness of extensible type systems while maintaining flexibility. In our system, extensible type system designers do not need to explicitly define every single type label and their combined version, while also having the assurance that unexpected type label instances are not being erroneously accepted as valid by the type checker. This mechanism is especially useful for a type system that deals with units of dimension, as such type systems are prone to require a very large number of type combinations that would not be practical to explicitly define, but whose combinations still need to be well-defined to avoid the instantiation of invalid combinations. We compare our implementation with other systems to demonstrate that our mechanism provides enough flexibility to allow programmers to define types that would be difficult or otherwise impractical to define in extensible type systems that do not permit type combinations, while providing more robustness than extensible type systems that liberally perform type combinations and incur the risk of permitting the instantiation of type combinations that programmers did not intend for.

Secondly, we provide a better mechanism for handling type labels within the same dimension by reducing the number of implicit conversions and automatically normalizing any ambiguity that may arise when merging multiple data flows. In the context of type systems supporting units of dimension, our approach provides better handling of scaled units within the same dimension by automatically converting scaled units to the best unit instance when merging data flows containing a mixed range of dimensional units. Our novel type normalization algorithm also provides an optimal way of handling implicit conversions in the context of extensible type systems by reducing superfluous type conversions. We later compare in chapter 4 our proposed system against some related work and show that ours allows more concise user-defined type definitions while addressing user-defined application semantics where existing systems only partially do. We also compare our normalization algorithm with the traditional normalization algorithm to show

that our algorithm provides better compiler performance through a reduction in the time spent by the compiler on normalization and delivers some improvements on the precision of floating-point values in specific cases.

1.4 Organization

We present an explanation of type systems and extensible type systems, as well as related work that have been done in relation to extensible type systems in chapter 2. In chapter 3, we present an overview of our system, called composite type labels, and how we address some of the issues with current implementations of extensible type systems. We also present the details of our type checker and normalizer implementation, as well as the details of our optimized normalization algorithm. Chapter 4 presents a comparison of our system against existing systems, as well as the results of our experiments measuring the performance of our proposed normalization algorithm against a traditional normalization algorithm implementation, and a discussion of the results. Chapter 5 concludes this thesis with a summary of our contributions and the limitations of our system with an outlook on future work.

Chapter 2

Background

A type system is a powerful tool for the purpose of program verification to avoid or minimize type errors during program execution. The type system provides a finite set of types and verifiable rules that a special algorithm called the type checker uses to validate the user program and generate error messages. While the type system that comes built-into the host programming language is a closed set, there has been an interest in providing ways for programmers to define their own type systems as extensions to the host language's own [2] in order to address the need of use-case specific types.

This chapter provides an introduction to type systems, type checking and extensible type systems, along with existing related works pertaining to extensible type systems and the limitations motivating our research.

2.1 Background Topics

This section presents background information relevant to the understanding of type systems and the necessity of providing user-extensible type systems.

2.1.1 Type Systems

A type system is comprised of types, like `Boolean` or `Integer`, which are annotations that define the behavior and range of the variables they are attached to via type rules. For instance, a value of type `Boolean` may only have two possible values (`true` or `false`) while a value of type `String` may not be allowed as part of a subtraction expression. The amount of memory allocated to each type may also be different. For instance, a `Boolean` gets allocated 1 byte whereas an `Integer` gets allocated 4 bytes and therefore merging the two could be a dangerous operation if the type system doesn't specify if such an operation is permitted or what the outcome should be so that variables are not granted illegal access to memory locations they should not have access to. At the low level, the type system assigns meaning to arbitrary sequences of bits that are otherwise undistinguishable to the underlying hardware.

Type systems are further comprised of type rules, which are a collection of type judgements that determine the validity of types. Those validity of those judgements are also put in context

with the other judgements that form the type system in a modular fashion. For example, the type judgement for the type `Boolean` is written independently to the type judgement of the type `Integer` in a way that new types and type judgement can be added to the type system.

Those rules define the semantics of the program and can be formally verified. A program that has a valid judgement, in other words a program for which each fragment can be associated to a type, is said to be well-typed. If such a condition is not met, the program is said to contain a type error. The association of a program fragment to a type can be performed explicitly through type annotations inserted by the programmer. In the absence of such type annotations, the correct type to be associated to the program fragment can be inferred through a more complex mechanism called type inference as part of the type checking algorithm. In all cases, the program is either well-typed or must return a type error.

2.1.2 Abstract Syntax Tree

It is also a good time to mention the Abstract Syntax Tree (AST), which is what type checkers usually work with. During the compilation process, the source code is first segmented into tokens which are then presented in the form of a tree representing the abstract syntactic structure of the source code by another part of the compilation pipeline called the syntax analysis. The AST can then be traversed by various processes such as the type checker, and some nodes of the tree can also be replaced, for instance to perform type normalization or to insert additional metadata.

2.1.3 Type Checking

Type checking is the process with which a program's source code is validated according to the rules defined by the type system in order to detect type errors. Such a process may occur at compile-time or during the run time of the program with different implications. It is also worth mentioning that while type checking aims to detect type errors, it may not be able to detect all type errors and does not entirely eliminate the occurrence of run time errors.

Cardelli [1] lists three basic properties that a type system should adhere to:

- Decidably verifiable: as the type system is meant to catch type errors, there must be an algorithm, referred to as the type checker, which can ensure that in a deterministic way.
- Transparent: the rules of the type system should be known to the programmers such that they can expect a program to type check properly or fail in a self-evident way. Type annotations inserted throughout the program contributes to the former, while compiler error messages or exceptions in the case of run time errors contribute to the latter.
- Enforceable: the rules of the type system must be checked statically wherever possible, falling back to dynamic verification whenever that cannot be realized.

Static Type Checking

A program is said to be statically type checked when the type checking process is performed during its compilation. Since the type checking is performed as part of the compiler pipeline, a

program that passes the static type checker is guaranteed to satisfy the type rules defined by its type system.

However, statically proving that all inputs of the program satisfy type rules is a difficult task and the compiler may not always be able to arrive to such proof. In such cases, dynamic safety checks that are consumed at run time need to be emitted by the compiler. In the case that no dynamic safety checks are required, the program is deemed to be well-typed and the resulting binary is expected to run faster in theory. This last point is often hailed as one of the major benefits of static verification.

Typical situations where static type checking may not be conclusive include downcasting. For example, if we had a supertype `Length` and know what the expected run time instance is, we can instruct the compiler to downcast `Length` to a specific instance like `Meter`. However, in such case some run time checks will be necessary to make sure that this conversion makes sense at all code points. For example, Java allows programmers to insert downcasts wherever there is a possibility that the downcasting succeeds at run time, but because of this it is not possible for it to provide a compile time guarantee that all cases will succeed.

Dynamic Type Checking

Other than as a fallback to static type checking when cases that cannot be statically proven are encountered, a programming language can rely on dynamic type checking exclusively. Dynamic type checking refers to the run time verification of the type safety of a program. The syntax of purely dynamically typed programming languages usually does not contain type annotations. Instead, each variable may embed their type information in the form of type tags in order to determine their type at run time. The embedded type tag, often a pointer, can then be extracted and used for type checking.

One of the hailed advantages of dynamic type checking is that dynamically typed languages provide more expressiveness to programmers by allowing variables to hold arbitrary types. However, the downside is that since type checking is performed at run time, there is an execution overhead incurred due to the type checking routine. Safety also becomes a concern since type checking failures can often result in program crashes when the program is not able to recover from those failures or exit gracefully.

Other Forms of Type Checking

Some programming languages provide a way for programmers to turn off static type checking on some variables and instead rely on run time verification. For instance, the C# programming language includes the type annotation `dynamic` which signals to the compiler that it should not type check the variable this type annotation is attached to and instead generate metadata for run time checking. Conversely, some programming languages such as Clojure and Common Lisp which are traditionally dynamically type-checked may allow programmers to annotate variables in order to solicit the compiler to statically check them.

The term *gradual typing* [7] has been coined to qualify programming languages that support both static and dynamic type checking and provides programmers with an interface to specify the desired mode of type checking in the source code.

2.1.4 Extensible Type Systems

One of the criticisms of static type checking is that it imposes upon programmers a strict set of typing rules that they must adhere to in order to write their programs, which restricts their freedom of expressiveness. Furthermore, as the built-in primitive types of a programming language are designed to be suitable for most general-purpose applications, they may not be appropriate for all specific-use cases. Programmers relying on the built-in type system must therefore work around its peculiarities, for example by inserting null checking code on string types or range checking on numerical types, since while their program may successfully pass type checking, it may still cause errors due to their application-specific requirements. For that reason, there is an interest [8, 9] in extending static checking in existing programming languages and in providing more powerful mechanisms.

Bracha [2] proposed treating type systems as modules of a programming language rather than as an integral part of it. His argument is that the mandatory nature of a programming language's type system can paradoxically reduce safety guarantees because of the need to adapt built-in types to application-specific workflows. From this idea a number of systems providing support for user-extensible, or pluggable, type systems have been proposed. While those systems are successful at atomically dealing with user-defined types, their support for more complex workflows such as allowing for combinations of user-defined types is still lacking.

2.2 Motivating Problems

The goal of user-extensible type systems being to allow users to extend the built-in type system to fit their specific use cases, any such system must be able to accommodate complex type systems. We found that existing support for user-defined application semantics is still lacking in some respects. Some application semantics may require the grouping of types. One example of that is an application where users want to define units such as `meter`, `second`, etc. and group them semantically into dimensions such as `Length` and `Time` and be able to automatically combine them to form the dimension of `Speed`. In such case, defining the application semantics granularly may not be practical since the user will need to express every possible combination of `Lenght` and `Time` in their definition for `Speed`. Existing systems provide limited support to define such semantics.

Figure 2.1 shows an example where we would like our type system to automatically recognize combined measurement units in the context of their component types. We have identified two main problems in existing systems that make implementing such type systems difficult.

2.2.1 Inter-dimension Problem

We call the inter-dimension problem the issue related to handling type combinations between unit types of different dimensions, for example resolving combinations of `Length` and `Time` into the dimension of `Speed`. Most current systems can be broadly divided into two types: those that forbid any form of type combinations, and those that permit any and all form of type combinations.

```
1 float@m m_val = 1.0f;
2
3 float@sec sec_val = 60.0f;
4 float@min min_val = 1.0f;
5
6 // Speed types automatically recognized
7 float@m/sec speed_val1 = m_val/sec_val;
8 float@m/min speed_val2 = m_val/min_val;
```

Figure 2.1. Example of type combination

User-extensible type systems that forbid any form of type combinations lack the flexibility that would make such systems useful. Indeed, users of such systems need to explicitly define every single unit of dimension. Furthermore, in such systems, units of `Speed` such as `km/hour` (kilometer per hour) are completely distinct from their components (`km` and `hour`) making the merging of data flows of `Length` (`km`) and `Time` (`hour`) into `Speed` (`km/hour`) a non-trivial exercise since programmers need to explicitly define any such operation as well as their semantics. For a multi-dimensional type system with N dimensional types of m discrete units each, the definition problem quickly becomes a polynomial order effort and therefore makes such a type system not only impractical to create, but also difficult to maintain.

In turn, user-extensible type systems that permit any and all form of type combinations offer great flexibility but do so at the expense of type safety. This is bad because those systems tend to allow any user-defined types to be combined, which means that even type combinations that the user did not intend for will be allowed by the type checker. Therefore, the safety promises of the static type checker are not fulfilled as some type errors related to type combinations will not be caught.

In addition, the unrestricted automatic type combination behavior may not be desirable in all application semantics. Figure 2.2 shows a code example of a permissive user-extensible type system. Here, line 5 results in a correct type, but the type checker has no way to decide that the output on lines 8 and 11 should be type errors since it is a question of programmer intent. Indeed, since there is no mechanism in existing systems for type system designers to define that expressions such as those in lines 8 and 11 should be rejected by the type checker, the resulting type checking behavior does not completely fit the application semantics they intended. Type errors in such situations are also difficult for end-user programmers to catch, since at a quick glance nothing appears to be wrong, and the compiler will not generate any error message.

While this can be alleviated by enforcing the use of type annotations, type systems that permissively support type combinations typically do not support annotating combinatory types since programmers did not explicitly define them. Instead, those systems require programmers to rely on type inference as the combinatory types are only understood by the type checker during compile time where symbols are automatically generated for them. This is illustrated by lines 8 and 11 with the use of `var` to declare variables.

```
1  float@man_hour workload = 40.0f;
2  float@man workers = 4.0f;
3  float@km distance = 1.0f;
4
5  float@hour time = workload/workers;
6
7  // Some type errors caught
8  var error = distance + workload;
9
10 // Non-sensical types can be created
11 var nonsense = workload/distance;
```

Figure 2.2. Creation of non-sensical types in permissive user-extensible type systems

2.2.2 Intra-dimension Problem

The other problem we have identified with supporting type combinations in user-extensible type systems is what we call the intra-dimensional problem. This refers to the handling of units within the same dimension. More generally, it is desirable for a system supporting user-defined semantics to support modern concerns such as type coercion and type inference. In fact, it is necessary for them to support such concerns to be truly useful. Indeed, in a useful type system that deals with units of dimension it should be possible to merge data flows containing different units in a coherent way. For instance, since `meter` and `kilometer` are semantically scaled versions of the same `Length` dimension, we naturally want to be able to merge them. Furthermore, this behavior should remain compatible with type inference, as we have previously shown that type inference is necessary to handle implicitly defined dimensional units.

Figure 2.3 illustrates an example where type inference applied to the result of data flow merging within the same dimension can yield uncertain results. On line 4, the programmer coerced the result type to a `meter` dimensional unit, but on line 6 no such coercion is declared, and it is not immediately obvious whether the result should be of type `meter` or `kilometer`.

Here, existing systems again broadly adopt one of two approaches. Some systems treat each dimensional unit type atomically and do not permit any data flow merging operation to occur. Others require the selection of a default dimensional unit, for example `meter`, and implicitly convert all scaled instances to the default one. While the latter approach offers better flexibility than the former, we find both approaches lacking, as this forbids programmers to apply type coercion to non-default dimensional units. Figure 2.4 illustrates an example a type system where the dimensional unit `meter` would be the default unit for the dimension of `Length`. If programmers add two values of type `cm` (centimeter) as seen on line 4, the resulting type of `sum_value` will be forced back to `m`.

Another problem with existing systems is the way implicit conversions are handled. Indeed, implicit conversions during type normalization are handled in with the same algorithm as traditional type systems. In traditional type systems, variables of the lower precision type are first up-casted to the higher precision type, then down-casted at the end of the expression when nec-

```
1 float@m m_value = 1000.0f;
2 float@km km_value = 1.0f;
3
4 float@m sum_value = m_value + km_value;
5 // the resulting type below is uncertain
6 var uncertain_value = m_value + km_value;
```

Figure 2.3. Example where the result of type inference is uncertain

```
1 float@cm cm_value1 = 10.0f;
2 float@cm cm_value2 = 1.0f;
3
4 var sum_value = cm_value1 + cm_value2; //treated as @m
```

Figure 2.4. Example showing where implicit conversions result in unintended result types

essary. This is illustrated in Figure 2.5 where `integer` values are first casted to `float` values before the final `integer` conversion required by the casting operator.

While this makes sense in traditional systems, this algorithm is actually harmful for handling composite types in extensible type systems. Indeed, unlike traditional systems where the type itself carries implications for precision, dimensional units themselves act as labels or tags refining the underlying built-in primitive type of the host programming language. Furthermore, dimensional units such as `cm` or `km` carry information pertaining to the scale of the dimension, and therefore repeated implicit conversions can actually incur the risk of harming precision. If we go back to the example on Figure 2.4, the expression on line 4 will have each of its components (`cm_value1` and `cm_value2`) individually converted to `m`, with `sum_value` being assigned a type of `m` by type inference where it may make more sense to not insert any implicit conversion and infer `cm` as the resulting type of `sum_value`. Additionally, those repeated conversions, which are inserted by the type checker by modifying nodes in the AST, may also come with a performance cost.

2.3 Related Works

The need for user-extensible type systems has been identified by Bracha, who used the term *pluggable* types to describe such systems [2]. We have found several systems created with this idea of extensibility in mind, as well as systems supporting units of dimension that to some

```
1 var foo = (int)(1.0f + 1 * (5 + 1));
```

Figure 2.5. Example of traditional type normalization

```
1  rule checkMeter (Assign a) {
2      where (isMeter(a.lhs)) {
3          require (isMeter(a.rhs)) :
4              error(a, "Invalid type assignment to @Meter")
5      }
6  }
7  declare isMeter(Symbol s) {
8      require (s.hasAnnotation("Meter"));
9      require (isNumeric(s));
10 }
```

Figure 2.6. JavaCOP example of a rule for validating assignments of the measurement type @Meter

extent achieve user-extensibility of the type system and therefore fit the idea of user-extensible or pluggable type systems. This section discusses such systems, as well as some of their limitations relevant to the subject of this thesis.

2.3.1 JavaCOP

JavaCOP [10] is a pluggable type system implementation for the Java 5 programming language. The user-defined type rules act as refinements to the Java AST to provide additional type checking information to the compiler. User-defined types, implemented as annotations, are attached to existing Java types, allowing them to interact with Java's existing type system. Those user-defined type rules are type checked only after the regular Java type checking has successfully completed. Programmers can define annotations such as `@NonNull` which enacts some restraints on existing Java types receiving those annotations. The `@NonNull` example puts a constraint on reference types indicating that they cannot be null-valued. Figure 2.6 shows an example of a rule definition for checking assignment expressions containing the unit of measurement `@Meter` in JavaCOP. The rule is applied to assignment expressions where the left-hand side is of type `@Meter` and enforces the constraint that the right-hand side of the expression must also be of type `@Meter` or an error is thrown. The `isMeter(Symbol s)` declaration performs a check on the given symbols to determine if the relevant annotation has been inserted and the value is numerical.

JavaCOP contains its own abstraction of the AST generated by the *javac* Java compiler, containing additional nodes used for type checking user-defined types. Since the traversal of the JavaCOP AST occurs after regular type-checking, the JavaCOP traversal can make use of type information provided by *javac*.

The JavaCOP implementation of user-extensible type systems support is, as the name JavaCOP (Constraints On Programs) suggests, a system that enforces constraint rules on existing Java types. JavaCOP is actually divided into two major parts; it has a compiler component and a framework component. Users first write constraint rules in the JavaCOP rule language, the set of which is parsed by the JavaCOP provided compiler into a set of compiled constraints in Java

code. The compiled constraints have access to the program's AST through an interface provided by the framework component of JavaCOP, allowing them to act upon the AST. This allows the JavaCOP type checker to emit errors or compile Java bytecode as a final step if all the constraints are preserved.

Other than transforming source code into bytecode, JavaCOP does not perform any additional code rewriting or modification. Furthermore, the type rules in JavaCOP are atomical; JavaCOP does not support type combinations, making support for units of dimension difficult. The lack of code rewriting support also means that it does not support implicit conversions. As a result, the system provided by JavaCOP lacks the flexibility needed for supporting units of dimension.

Support for type inference was later partially addressed by the authors with the addition of a dataflow engine that can perform data flow analysis based on user-defined semantics [11]. While this allows some form of type inference, there is no indication of that mechanism allowing users to define implicit conversions when necessary. What the dataflow engine in JavaCOP allows for is to refine the type of a variable based on data flow. For instance, to allow a regular string to receive the `@NonNull` refined type if dataflow analysis can determine that the refined type better describes it.

2.3.2 Checker Framework

The Checker Framework [6] is a pluggable type system implementation for Java that uses a new Java specification, JSR308, that provides better support for annotations. This provides better expressiveness and less false positives than with JavaCOP. The Checker Framework, which works in a similar way to JavaCOP, is also designed with better support for object-oriented language support. For instance, it supports method overriding and generics.

Similarly to JavaCOP, the Checker Framework performs static type checking of user-defined types once the regular Java compiler type checking has completed, allowing its type checker access to metadata generated by the regular Java compiler. With the better annotation support provided by JSR308, the Checker Framework avoids running into previously encountered issues where some annotations were rejected by the regular Java compiler because they were not recognized.

Another difference with the Checker Framework is that it brings support for user-defined type hierarchies, which could form an arbitrary directed acyclic graph (DAG). Unannotated variables can be automatically assigned to a default type qualifier, and through its data flow analysis mechanism the Checker Framework can assign variables to a more specific type qualifier through a mechanism the author call type refinement. Figure 2.7 shows an example of user-defined types having a hierarchical relationship. The `@Length int` is the most abstract type qualifier, and when a more specific qualifier is found for a variable via data flow analysis, it can be assigned the more specific type, for instance `@m int`.

One problem with the Checker Framework when it comes to supporting a type system for units of dimension is that it doesn't support combining data flow merging of scaled units within the same dimension. Indeed, the Checker Framework doesn't support implicitly converting from one scaled unit to another within the same dimension, resulting in any expression attempting to merge the two will not type check. Furthermore, we have found cases where type checking is not performed, and erroneous expressions are accepted as valid. Figure 2.8 illustrates such situations. On line 4, type checking is not performed within the call to `System.out.println`

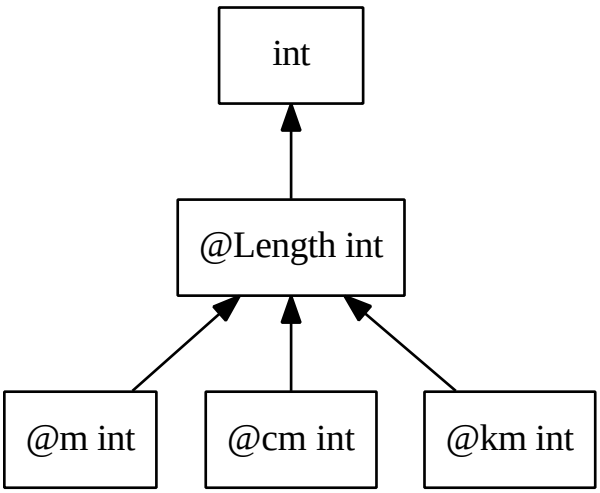

Figure 2.7. Type hierarchy in the Checker Framework

```
1 @m double m_value = 5 * m;
2 @mm double mm_value = 5 * mm;
3
4 System.out.println(mm_value + m_value); //incorrect
5 @m double foo = m_value + mm_value; // type error
```

Figure 2.8. Measurement unit example in Checker Framework, written in Java, where incorrect results are output

and a naive value addition is performed. On line 5, since implicit conversions between `mm` and `m` are not supported, the addition fails with a type error. This is due to the Checker Framework not providing support for rewriting variable values during type checking, so it is not able to insert implicit conversions or run time checking code.

Another problem with the Checker Framework is that while there is some basic support for combining user-defined types, each combination must be atomically defined by the user. Figure 2.9 shows how programmers using the Checker Framework would define type combinations resulting from a multiplication operation. In lines 7-10, individual combinations of discrete `@Length` and `@Speed` dimensions are defined. For example, the statement in line 7 checks for a multiplication expression having a `s` (second) on one side and a `mPERs` (meter per second) on the other to validate the result as a `m` (meter). If a programmer forgets to define a pair, that pair would not type check. As we have discussed in Section 2.2.1, for a user-extensible type system supporting units of dimension this would lead to a polynomial order definition effort.

```
1  public AnnotationMirror multiplication(
2      AnnotatedTypeMirror lht, AnnotatedTypeMirror rht) {
3      if (UnitsRelationsTools.hasSpecificUnitIgnoringPrefix(lht, m)
4              && UnitsRelationsTools.hasSpecificUnitIgnoringPrefix(
                   rht, m)) {
5          ...
6
7      } else if (havePairOfUnitsIgnoringOrder(lht, s, rht, mPERs)) {

8          return m;
9      } else if (havePairOfUnitsIgnoringOrder(lht, h, rht, kmPERh))
           {
10         return km;
11     } else {
12         return null;
13     }
14 }
```

Figure 2.9. Definition of a type combination in Checker Framework, written in Java

2.3.3 F# Units of Measure Types

Some programming languages [5, 12, 13, 3] already come with built-in support for units of dimension in their type system, which contributes to show the need for such a type system. We take F# into consideration since by allowing users to define their own units of dimension, it effectively supports user-defined types and its type system is therefore to some extent user-extensible. The dimensional unit implementation in F# and other similar systems mostly borrow concepts from the CQual parser [14] which extends the C type system with user-defined qualifiers and provides type checking features for those qualifiers.

The F# implementation offers great flexibility as any user-defined type can be combined through division, multiplication, and power to automatically form new user-defined types. Figure 2.10 shows such an example where on lines 1-3 we define three arbitrary units of measurement `person`, `hour` and `foo`. The `person-hour` unit of measurement is automatically instantiated on line 6 as a composition of `person` and `hour` by the multiplication operator and the F# type checker supports type inference.

We however found the F# support for user-extensibility to be too liberal in applying type combinations. Indeed, since it accepts any combination previously defined or inferred units of measurement, it becomes possible to instantiate an invalid measurement unit and have it successfully pass type checking, as shown on line 13 of Figure 2.10. F# therefore suffers from the same inter-dimension problem we discussed in Section 2.2.1.

Another problem with the F# implementation which stems from the simplicity of the measurement unit definition is that there is no way to group semantically related units into the same dimension, for example having scaled units of `Length`. Similarly, implicit conversions between

```
1  [<Measure>] type person
2  [<Measure>] type hour
3  [<Measure>] type foo
4
5  // types can be combined
6  let workload = 40.0<person * hour>
7  let workers = 4.0<person>
8  let time = workload/workers
9  // val time : float<hour>
10
11 // invalid combinations are accepted
12 let foo_value = 1.0<foo>
13 let nonsense = workload/foo_value
14 // val nonsense : float<hour person/foo>
```

Figure 2.10. Defining and using units of dimension in the F# programming language

```
1  [<Measure>] type m
2  [<Measure>] type cm
3  let m_to_cm ( x : float<m> ) = x * 100.0<cm/m>
4  let cm_to_m ( x : float<cm> ) = x / 100.0<cm/m>
5
6  let m_value = 1.0<m>
7  let cm_value = m_to_cm m_value
8  // val cm_value : float<cm> = 100.0
```

Figure 2.11. Defining unit conversions in the F# programming language

scaled units is not supported, and each conversion pair must be explicitly defined by the programmer as shown on Figure 2.11. Programmers must first define conversions as shown on line 3, where measurement unit and value conversions are manually applied. The conversion function must then be explicitly called every time a conversion is needed, like on line 7. The use of explicit conversions not only add to the cognitive load on the programmer but also increases the chances of bugs when conversions are omitted.

2.3.4 Boost Units

The C++ Boost Units 1.1.0 library [4] offers a macro-based implementation of a type system supporting units of measurement. Their implementation is relevant to our discussion since it also provides user-extensibility. Similarly to the Checker Framework, Boost supports some form of grouping of units of measurement into dimensions such as `Length` or `Time`. Users can define their own units as shown in Figure 2.12 where we are simply defining a `worker` dimension which can

```
1  namespace worker {
2      struct worker_base_dimension :
3          boost::units::base_dimension<worker_base_dimension, -10>
4      {};
5      typedef worker_base_dimension::dimension_type worker_dimension
           ;
6      struct worker_base_unit :
7          boost::units::base_unit<worker_base_unit, worker_dimension
               , -10>
8      {
9          static std::string name()       { return "person"; }
10         static std::string symbol()     { return "person"; }
11     };
12     typedef boost::units::make_system<worker_base_unit>::type
           system;
13     typedef unit<worker_dimension,system>    worker;
14     static const worker person,persons;
15 }
```

Figure 2.12. Example of defining a new measurement unit in the C++ Boost library

take the concrete unit `person` or `persons`. It also supports composite dimensions by specifying a list of dimensions composing them and an integer indicating their power. Figure 2.13 shows the definition of two such dimensions. On the first line, an `Area` dimension is defined by being composed of the `Length` dimension with a power of 2, while on the second line a more complex dimension, `Workload`, is defined as being composed of a `Worker` dimension and a `Time` dimension, both with a positive power of one, yielding a composite unit of `person-hour`.

Arbitrary combinations are also accepted, as shown in Figure 2.14 where we use the `worker` dimension we previously defined and divide it with a `length` dimension. The result of that operation is improperly statically checked and accepted as a valid operation, which is the same problematic behavior presented in Section 2.2.1.

While it appears as if any scaled instances of units within a dimension are accepted here, in truth the Boost library deals with a default unit, the base unit for the dimension. For example, in the code sample shown in Figure 2.15 we define measurement units for `millimeter` on line 1 as `meter` multiplied by the `milli` scale, but the scaled unit is simply an alias as shown on line 2 where we use an arbitrary name to define the same `millimeter`.

As a result of scaled units being aliases of a single default unit, users have limited expressiveness in using scaled units within a dimension. For example, they cannot coerce the type of a variable to `millimeter` since the actual type `millimeter` does not exist. Any usage of the `millimeter` gets automatically converted to `meter` as shown on line 8 of Figure 2.15 where printing a variable that should hold `1000 mm` actually outputs `1 m`. Similarly, aliases are treated as equal if their value are equal since they are implicitly converted to their default measurement unit,

```
1 typedef derived_dimension<length_base_dimension,2>::type
    area_dimension;
2 typedef derived_dimension<worker_base_dimension,1,
3                           time_base_dimension,1>::type
                               workload_dimension;
```

Figure 2.13. Composite dimensions in the C++ library Boost

```
1 quantity<worker::worker> workers(4.0 * worker::person);
2 quantity<length> distance(1.0 * meter);
3 auto foo = workers / distance;
4 std::cout << foo << std::endl;
5 // 4 person m-1
```

Figure 2.14. Example where arbitrary combinations of dimension units yields invalid result, in C++ Boost

as shown on lines 11-12 where our user-defined `millimeter` and `unmeter` are deemed equivalent.

The Boost Units library does support implicit conversions, albeit only to the default unit, as shown on lines 15-16 where `millimeter` values are automatically converted to `meter`. It is possible to print values with automatic scaling applied, but this is only applied to the output display string and not to the internal representation of the variable. Since implicit conversions are applied greedily, the Boost Units library still exhibits the intra-dimension issues presented in Section 2.2.2.

2.3.5 AbleC

AbleC [15, 16] is an extensible language framework that supports user-defined type systems for the C programming language in a highly modular fashion. AbleC itself is built on top of the *Silver* and *Copper* language extensions provided by the same authors. The framework provides similar features to existing frameworks such as JavaCOP and Checker Framework, with the addition of analyzers aimed at ensuring that different user-defined type systems won't interfere with each other when used jointly. Users can define their own types by defining new C type qualifiers and their semantics and AST traversal behavior. They can enable implicit conversion on their type system via code insertion when overloading operators. For example, an overload on the addition operator can specify that if both sides of the binary expression are type check the right-hand side type should be scaled to match the units on the left hand side of the expression.

AbleC however still has limited support for flow-sensitive type (qualifier) inference, which the authors list as a current limitation. As well, similarly to the previously introduced frameworks, AbleC suffers from the issues outlined in Sections 2.2.1-2.2.2. Indeed, it does support automatically combining user-defined types, and while it is to some extent possible to provide implicit

```
1  static const auto millimeter = milli * meter;
2  static const auto unmeter = milli * meter;
3
4  quantity<length> m_val(1.0 * meter);
5  quantity<length> mm_val(1000.0 * millimeter);
6  quantity<length> unm_val(1000.0 * unmeter);
7
8  std::cout << mm_val << std::endl;
9  // 1 m
10
11 bool bar = mm_val == unm_val;
12 std::cout << bar << std::endl;
13 // true
14
15 quantity<length> hoge = m_val + mm_val;
16 std::cout << hoge << std::endl;
17 // 1.001 m
```

Figure 2.15. Example of user code using the Boost library in C++

conversions, those conversions would be applied greedily resulting in sub-optimal performance and possible precision issues when it comes to units of measurement.

2.3.6 Turnstile

Turnstile [17] is a macro-based metalanguage for Racket that aims to provide support for user-defined typed domain specific languages (DSLs). The idea of using the macro system is not entirely new. The difference with traditional methods is that Turnstile does not rely on the host language's type system and is therefore not confined by its semantics. Instead, it relies on the host system's macro system. By doing so, it can leverage the macro expansion step of compilation to perform type checking. The type rule definitions themselves are written in a syntax resembling judgement syntax, and Turnstile is designed to be modular; different sets of user-defined rules can be used concurrently, although the authors do not specify how that is ensured.

During the compilation process of a Racket program an AST is first formed, similarly to previously introduced works. This provides sufficient metadata such as symbol and context information. Host language type information may not be present, but this is not required by Turnstile. The following macro expansion phase starts expanding user-defined macros to perform type checking and rewriting on the syntax tree. This step may yield additional macro calls, so the macro expansion step runs in a loop until no further macro calls are found. If no errors were encountered the output now only contains regular Racket code without macros.

Since it allows the creation of user-defined (and extensible) type systems, Turnstile should be able to allow users to create a type system based on dimensional units with user-provided type checks. There however seems to be no obvious support for automatic type creation, and

Turnstile does not appear to do anything in particular to address the issues presented in Sections 2.2.1-2.2.2.

Chapter 3

Improving Composability Support

In Chapter 2, we provided background information on user-extensible type systems and outlined some issues pertaining to their support of user-defined units of measurement and how existing systems do not address them in a satisfying matter. In this chapter, we introduce our proposed user-extensible type system, called composite type labels, designed to solve those issues. Namely, our system aims to provide better support for application semantics-aware error detection, while keeping the definition of such semantics down to a manageable size. Our implementation has three main features: composition rules, implicit conversion rules and a novel type normalization algorithm applied to binary expressions.

3.1 Composite Type Labels

At the base of user-extensible type systems are user-defined types, which exist on top of the host language's built-in types. The exact terminology for those user-defined types vary across user-extensible type system implementations and there is no common term, but for our system we chose to call them type labels.

Type labels describe the most atomic types in our implementation; they are used to describe concrete types. In a type system for units of measurement, type labels would be used to define units of `sec` (second) or `m` (meter) for instance. Similarly to the related works such as the Checker Framework, we implement type labels in the form of annotations attached to primitive types in the host programming language, which is C# in the case of our implementation.

On top of regular type labels, we introduce composite type labels, which are special type labels that are defined by the composition of regular type labels or other composite type labels. Composite type labels act like an abstract type label that can be an instance of any of its members or composed labels. The concrete instance is selected automatically during the type checking process based on static type inference.

Figure 3.1 illustrates the relationship between regular type labels and composite type labels. Composite type labels are shown with a capitalized first letter while regular type labels are shown entirely in lower capitalization. At the root of each graph are the composite type labels. The child nodes are regular type labels which have been defined as being members of the composite type label. So for instance, `@cm`, `@m` and `@km` are dimensional units of length and belong to the

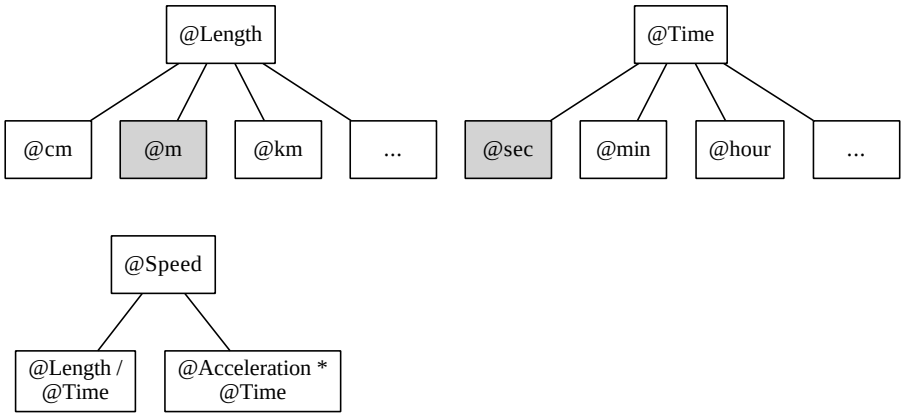

Figure 3.1. Composition example of composite type labels

`@Length` composite type label and any variable declared with a `@Length` type label can take any instance of its members. On the other hand, the `@Speed` composite label illustrates an example where a composite type label is defined as the composition of other composite type labels; its child nodes show composite type labels joined by a binary operation.

Similarly to previous work, we also select one type label as a default or canonical representation for a composite type label, illustrated by the filled boxes. Existing systems use canonical labels as the internal representation of the entire dimension and greedily convert other type labels, for instance scaled instances of canonical labels, to the canonical label. Our approach differs in that we use canonical labels as a way to resolve ambiguities during type checking and normalization, and as a way to automatically find implicit conversions between scaled units of measurement. Both type labels and composite type labels can be defined by type system designers through a simple DSL we designed for this purpose.

3.1.1 Composition Rules

It is desirable for a user-extensible type system for units of measurement to be able to automatically combine existing measurement unit types to define new ones since doing so improves usability by reducing the definition burden on type system designers. For this reason, composite type labels also support automatic composition. To address the problem outlined in Section 2.2.1 where undesirable combinations may occur, we introduce composition rules in our definition mechanism for composite type labels.

Composition rules limit the automatic composition behavior by specifying which combinations of composite type labels should be accepted by the type checker as valid combinations, as well as the operator joining the left-hand side and right-hand side composite type labels. Figure 3.2 shows an example where we define two composite type labels, `@Speed` and `@Length`. In the case of `@Length`, the concrete instances it can take are expressed as any instance of the regular type labels `@m`, `@cm`, etc. as well as any instance which is a binary expression taking a `@Speed` dimension

```
1 CompositeLabel Speed {
2     CompositionRules: Acceleration * Time | Length / Time | ...
3 }
4 CompositeLabel Length {
5     MemberLabels: m | cm | km | ...
6     CompositionRules: Speed * Time | ...
7 }
```

Figure 3.2. Definition of a composite type label

```
1 float@m m_val = 1.0f;
2
3 float@sec sec_val = 60.0f;
4 float@min min_val = 1.0f;
5
6 float@Speed speed_val1 = m_val/sec_val; // m/sec
7 float@Speed speed_val2 = m_val/min_val; // m/min
```

Figure 3.3. Example where composite type labels receive different concrete type labels

on the left-hand side and a `@Time` dimension on the right-hand side joined by the multiplication operator.

On the other hand, we can define `@Speed` with composition rules exclusively; it is defined either as a binary multiplication expression taking an `@Acceleration` value on the left-hand side and a `@Time` value on the right-hand side, or as a division of `@Length` and `@Time`. It is worth noting that programmers do not need to define the `CompositionRules: Acceleration * Time * Time` rule on the `@Length` composite type label since that rule is automatically handled by the type checker through the substitution of `@Speed` by its own composition rules.

During compilation, the type checker will automatically accept concrete type label instances that adhere to those rules. Figure 3.3 shows an example where a combination of different type labels yields different concrete type labels. In line 6 the resulting type becomes `m/sec` because the variable declared as `@Speed` receives a `@m` and a `@sec` and adheres to the `Length / Time` rule. In the case of line 7, the resulting type becomes `m/min` because a `@min` was received instead. The type checker refines those two `@Speed` variables with their respective concrete types and any subsequent usage of those variables take the refined type into account by virtue of data flow analysis.

Since our type checker operates on the basis of conversion rules, it predictably becomes able to reject invalid type label combinations, for example if the user tries to add a `@Speed` value to a `@Length` value or divide a `@Time` value by a `@Length` value.

```
1  Label m {
2      AllowedTypes: uint ulong double float
3      Canonical
4  }
5  Label km {
6      AllowedTypes: uint ulong double float
7      ConvertFromCanonical: Divide(1000)
8      ConvertToCanonical: Multiply(1000)
9  }
10 Label cm {
11     AllowedTypes: uint ulong double float
12     ConvertFromCanonical: Multiply(100)
13     ConvertToCanonical: Divide(100)
14 }
```

Figure 3.4. Definition of a type label including implicit conversion rules

3.1.2 Implicit Conversion Rules

Since we allow automatic type label combinations, special consideration must also be given to the merging of data flow containing mixed type labels. For such a type system to be useful, it becomes necessary to support implicit conversion.

Implicit conversions are useful to reduce the annotation burden on programmers. However, this is made possible because the implicit conversion behavior has been defined by the type system designer beforehand. Traditionally, type system designers must explicitly define every implicit conversion, and in a type system that supports automatic type label combinations not all type labels are known beforehand and defining every conversions can become an unrealistically large task. It therefore becomes necessary to have a mechanism to automatically derive additional implicit conversions. This necessity was less apparent in existing systems since the only conversion allowed was often conversions from scaled units within the same dimension to a canonical unit, for instance from *centi-*, *milli-*, or *kilo-* to *meter*. It is possible in some existing systems [3] to use a shorthand way to chain previously defined implicit conversions, but those systems work on the basis of every user-defined types being known beforehand.

To solve this, we introduce implicit conversion rules to the definition of type labels. Rather than having type system designers define every conversion pair, we have them declare conversions *to* and *from* the canonical type label which serves as the default type label. For instance, in Figure 3.4 three regular type labels are defined, with `m` (meter) being selected as the canonical label with the `Canonical` clause in the label definition. The `AllowedTypes` clause specifies the host language's base types the type label can be applied to. The `ConvertFromCanonical` and `ConvertToCanonical` clauses in the definition of `km` and `cm` define implicit conversions from and to the canonical `m` label, respectively.

Based on those definitions, deriving additional implicit conversions become a path-finding effort. We say that it is possible to implicitly convert between two type labels if there exist

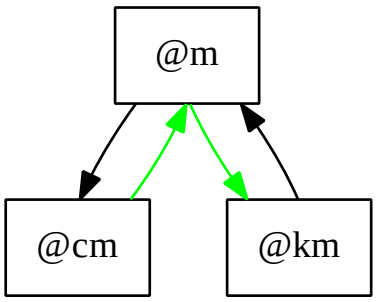

Figure 3.5. Example of implicit conversion path between scaled units of the same dimension

```
1 float@m m_val = 1.0f;
2 float@mm cm_val = 1000.0f;
3
4 float@sec sec_val = 60.0f;
5 float@min min_val = 1.0f;
6
7 float@Speed speed_val = m_val/sec_val + cm_val/min_val;
```

Figure 3.6. Example of data flow merging of mixed type labels and composite type labels

a conversion path between them that goes through the canonical label; the conversion is the concatenation of all path segments. Figure 3.5 illustrates the case for a conversion path between the `@cm` and `@km` type labels, with the green path showing that we can implicitly convert from `@cm` to `@km`. If programmers wish to forbid the reciprocal path of `@km` to `@cm`, it is sufficient to remove an edge by not inserting the `ConvertToCanonical` or `ConvertFromCanonical` clause in one of the type labels.

Composite type labels do not contain implicit conversion rules, as they inherit their rules from any regular type label that is encountered during their composition. This also means that the path-finding exercise must be done for all different measurement dimensions encountered. For instance, a `@Speed` instance using the `@Length / @Time` composition rule as shown in Figure 3.6 must be able to implicitly convert within `@Length` instances (`@m` and `@cm`) and within `@Time` instances (`@sec` and `@min`) on line 7 or type checking will fail.

3.1.3 Type Normalization

Type inference alone does not however solve the intra-dimension problem we outlined in Section 2.2.2. Indeed, the resulting type label on line 7 of Figure 3.6 is still unclear at this point. During type checking, we also perform type normalization to solve such ambiguous cases. In simple cases the canonical label can be selected. In the case of composite type labels, the canonical label of each dimension involved in the type composition can be used as a canonical.

This approach is however not optimal in all situations. For instance, if `@m` is the canonical

```
1 float@m m_val = 1.0f;
2 float@cm cm_val = 1000.0f;
3
4 float@cm cm_sum = cm_val + m_val + cm_val;
5 float@Length length_sum = cm_val + cm_val + cm_val + cm_val;
```

Figure 3.7. Example of inefficiency in the traditional type normalization algorithm

label for the `@Length` composite type label, the traditional type normalization algorithm would enforce conversions to `@m` in all sub-expressions. While this works for the type systems regularly found built into the host programming language, it is not the best approach for type systems that deal with units of dimension.

Figure 3.7 illustrates two such cases. On line 4, although the expected type label is `@cm` on the left-hand side of the assignment expression, all variables in the binary expression on the right-hand side of the assignment expression are first converted to `@m` before the entire binary expression gets converted to the expected `@cm` type label. Similarly, on line 5 no concrete type label is specified on the left-hand side of the assignment expression, and while the b̈est fitẅould be `@cm` since the right-hand side is comprised entirely of that type label, all individual variables would get converted to `@m` and the left-hand side of the assignment expression is resolved to a `@m` type label.

We developed a novel algorithm that aims to address this normalization issue. Our proposed algorithm reduces the use of implicit conversions, which as a result improves compilation time. For the time being, we focus on optimizing the normalization of binary expressions and haven't yet applied our algorithm to other types of expressions such as method calls.

Our algorithm, which is detailed in Figure 3.8, begins similarly to the existing algorithm, by operating on the leaves of the tree formed by the binary expression and working its way up. In our case, we only insert implicit conversions if a type label is found to occur more often in the partial expression than any other type label. If no such label is found, we defer the insertion of implicit conversions to the processing of the node's parent. This process is non-trivial because it must be done separately for each composite type label in use in the expression. As a result, the type checker will be working with partially normalized nodes due to the multiple deferrals that can occur and will have to revisit nodes once the desired type label for them is known at a later time. Furthermore, an answer might not be found once all nodes are consumed, for instance if no majority label is ever found. It is therefore necessary to have a final step that resolves such cases by converting all remaining nodes to a canonical type label. Lastly, the entire binary expression itself may have an expected type given to it through type coercion by the user. If that type is a concrete type label, we insert one final implicit conversion to convert the binary expression to the expected type label. If instead a composite type label has been specified by the user, this means type inference is left to the type checker, and we can refine that composite type label to the type of the binary expression.

To illustrate how our algorithm works, we retake the example presented in Figure 3.7 and examine the binary expression on line 4. The tree in Figure 3.9 (a) represents the binary expres-

```
1  if LHS is BES:
2      recurse
3  if RHS is BES:
4      recurse
5
6  let compositeLabels <- get type labels in use, group by composite
       label
7  for each compositeLabel in compositeLabels:
8          if all type labels are the same:
9              merge
10         else:
11             select major label
12             if found:
13                 convert all other type labels to major label
14             else:
15                 return the set containing all used type labels
16 finally:
17     if return type label is defined:
18         convert entire expression to expected type
19     else:
20         refine return type label
21     if any ambiguities remain:
22         convert ambiguous type labels to canonical label
```

Figure 3.8. Proposed normalization algorithm on binary expressions

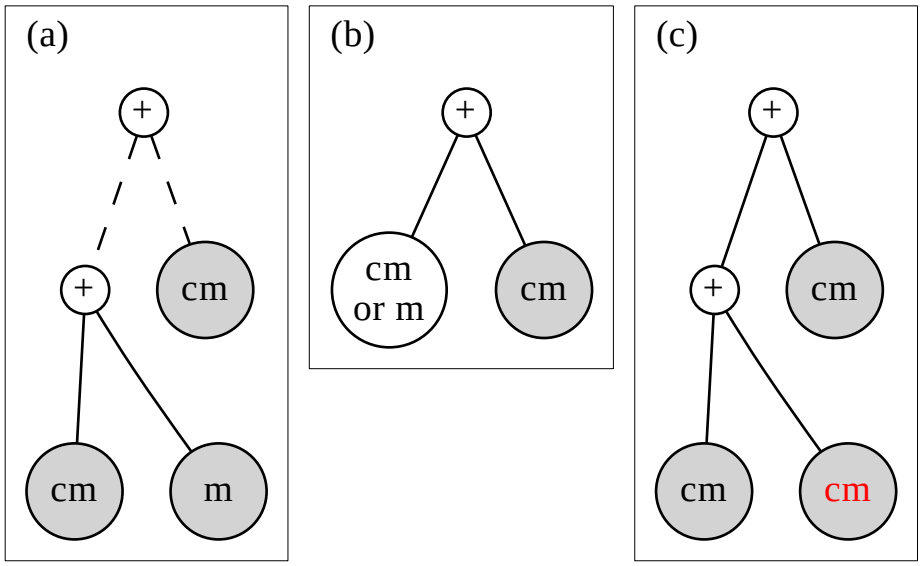

Figure 3.9. Normalization of a binary expression using a single composite type label

sion at the beginning of normalization. Since the sub-expression doesn't yield a clear major label as `cm` and `m` occur both exactly once, we defer any further processing on those nodes and return both type labels in (b). At the root of the expression we find one more occurrence of `cm` and can therefore a smaller number of implicit conversion is found to be a single conversion of `m` into `cm`, illustrated by the red-colored node in (c). Where the existing algorithm would have inserted three conversions ours only required one.

A more complex case is presented in Figure 3.10 where the binary expression involves different composite type labels: `Length`, `Time` and `Speed` which can be decomposed into `Length` and `Time` for simplicity. The initial tree shown in (a) corresponds to the binary expression `(m/sec + km /hour)* sec`. The sub-expression `m / sec` contains two type labels, but they belong to two different composite type label so the node can be merged as seen in (b). In the sub-expression in (c) however, two units of `Length`, `m` and `km` are in conflict without any clear resolution. Similarly, two units of `Time`, `sec` and `hour` are also conflicting. This sub-expression therefore returns all four type labels and defer their processing. In (d), the right-hand side type label is `sec` and solves the deferred `Time` sub-expressions, but leave the sub-expressions of `Length` untouched. If no further parent node exists, the `Length` sub-expressions will be converted to the canonical label.

We claim that with our algorithm we can achieve a smaller number of conversions in binary expressions. This can be demonstrated as follows. For a sub-expression comprised of n nodes and having a major label appearing n_m times, with m being at most n, we know that the number of conversions with the proposed algorithm c_p is $c_p = n - n_m$. In contrast, the legacy algorithm relies on converting all non-canonical nodes to the canonical label. With n_c being the number of times the canonical label appears in a sub-expression, the number of conversion c_l in the legacy case is $c_l = n - n_c$. It becomes obvious that if $c_l < c_p$, n_c is necessarily larger than n_m, which

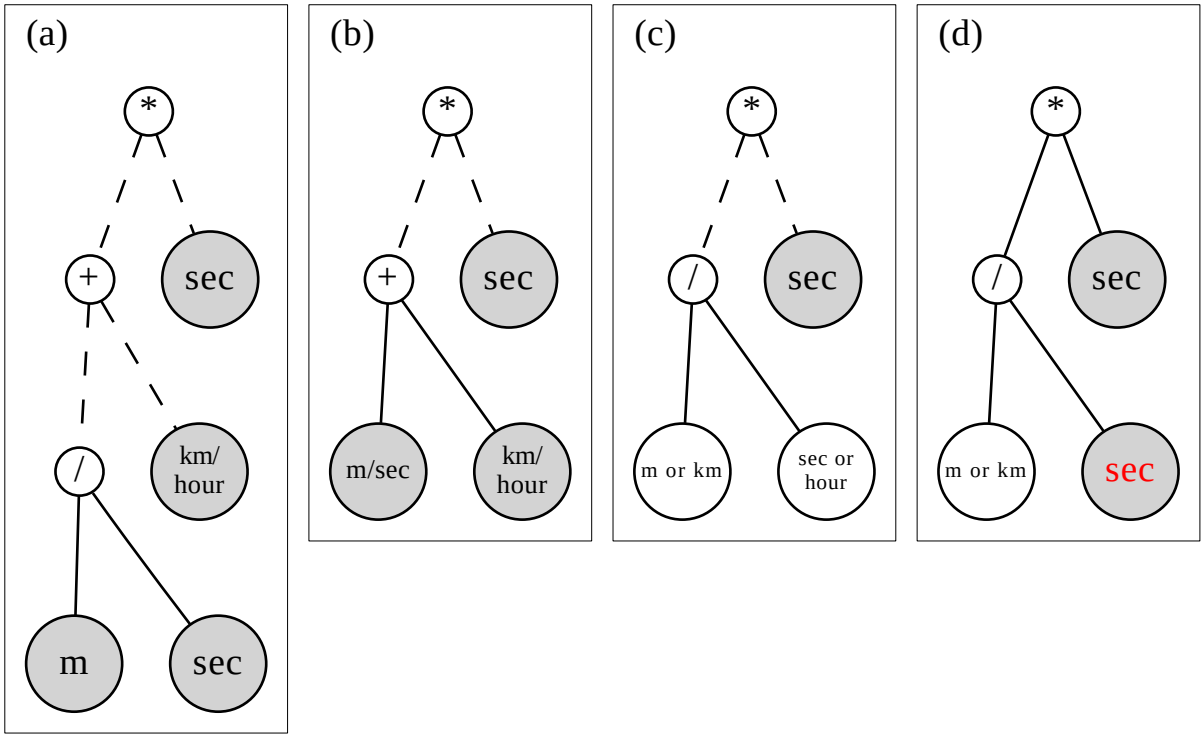

Figure 3.10. Normalization of a binary expression involving mixed composite type labels

```
1 public float@Length GetLength()
2 {
3     return 1.0f;
4 }
5 float@m m_val = 1.0f;
6 float@cm cm_val = 1000.0f;
7
8 float@cm cm_sum = cm_val + m_val + GetLength();
9 float@Length length_sum = cm_val + GetLength() + m_val + cm_val;
```

Figure 3.11. Example of binary expression involving method invocations with unclear return type

contradicts the definition of a major label. Hence, we know that in the worst case $n_c = n_m$ which results in the same number of conversions with both algorithms.

Now that we have proven a reduced solution is achieved for sub-expressions, we can extend that proof to the entire binary expression. For that, we observe that once a sub-expression has been converted, it will not be touched again for the same composite type label; if a conversion needs to be inserted, it is done on its parent. We can therefore assert that the total number of conversion in the binary expression is equivalent to the sum of conversions performed on its sub-expressions, expressed as $C = \sum_i c_i$. Again, if a smaller sum was found, this implies that for a c_j within the summation there exists a c_j^* such that $c_j^* < c_j$. By virtue of the previous demonstration for sub-expressions, if such a c_j^* were to be found this means that a major label has not been selected for the sub-expression as we have previously demonstrated that the sub-expression is reduced if a major label is selected. Any c_j^* is therefore necessarily larger than c_j and our binary expression can also be shown to contain a smaller number of conversions. We also note that in the worst case, all type labels are distinct, and no major label can be found in the entire binary expression. In such cases our algorithm can end up with the same number of implicit conversions as the existing normalization algorithm.

So far, we have only implemented our algorithm in binary expressions. While we have shown that this results in a number of implicit conversions that is a local improvement to the binary expression, we do not have a definite proof that this solution is the best for the whole program. This becomes more complex when we consider the fact that under our system method calls can have different return types depending on context. For instance, the `GetLength()` method in Figure 3.11 returns a `float@Length`. Suppose we refine the return type label to `float@cm` based on applying our algorithm on line 8. This refinement may cause other parts of the code to have to insert additional implicit conversions if they expect a different return type from `GetLength()` and while we have improved implicit conversions in one binary expression we may not have improved implicit conversions on the whole program.

3.2 Implementation Details

Our composite type label implementation is built on top of the C# programming language using the open source Roslyn [18] compiler platform. Since the C# syntax does not provide any support for type annotations, we first modified the open source C# compiler itself to recognize type label expressions in the form of `PredefinedTypeSyntax@TypeLabelSyntax` where `PredefinedTypeSyntax` is any built-in C# primitive type such as `int` or `float` and where `@TypeLabelSyntax` is the `@` symbol followed by a type label name. We are not however looking for a full compiler modification; similarly to some previous work, we perform type checking on type labels during compilation and remove any type label information before the intermediate language or machine code emitting phase. Hence, not all components of the compiler pipeline are relevant to our implementation and some shortcuts can be made; the purpose of our compiler modification is to provide enough metadata to our type checker by making use of the Roslyn syntax analyzer and semantic analyzer.

The Roslyn compilation pipeline has four phases: parsing, symbol generation, binding, and emitting. The parsing phase reads the source code into tokens and generate a syntax tree according to the C# language grammar. The symbol generation phase creates a symbol table from the declared variables containing various symbol metadata. The binding phase binds declared variables from the source code to the generated symbols. The emit phase compiles the source code into an assembly-like language called the Intermediate Language (IL). We will not be touching the emit phase.

3.2.1 Compiler Modifications

Our modifications are branched off of commit *10cde35e7feb54471941bc52fa7f3cba27c089e1* of the *dev16* branch in the official Roslyn repository. Syntax metadata for the compiler are found in `Syntax.xml` where we can define our new syntax. This is show in Figure 3.12 where we add two new syntax nodes: `LabelTypeSyntax` and `LabelNameSyntax`. The `LabelNameSyntax` node allows us to extract an identifier for the type label or composite type label for type checking while the `LabelTypeSyntax` node is defined as an extension to the regular `TypeSyntax` containing both the built-in `PredefinedTypeSyntax` and our type label as a `LabelNameSyntax`, which will later allow us to isolate expressions containing type label or composite type label.

A provided script, `generate-compiler-code.cmd` generates the relevant syntax for the compiler. Since we are not looking for full compiler support of our syntax it is necessary to manually edit the generated code to inhibit some errors that the compiler will throw due to new syntax being unrecognized further along the compiler pipeline. It is also necessary to assign a key to the new syntax token we are introducing, namely `LabelType` and `LabelName` which will be used later by the lexical analyzer to generate tokens.

Another manual step required is to expose a public API to our new syntax for the compiler so that we can write compiler extensions that consume it, for instance our composite type label type checker. Included in the public API are methods that allow AST manipulation of type label nodes. Since all AST nodes are immutable, any method that manipulates them return a new instance of the node to replace the original one. If for instance we want to modify the name of a type label, called `Identifier` here for some consistency with naming conventions in the Roslyn

```
1  <Node Name="LabelTypeSyntax" Base="TypeSyntax">
2      <Kind Name="LabelType"/>
3      <Field Name="ElementType" Type="PredefinedTypeSyntax">
4      </Field>
5      <Field Name="LabelToken" Type="SyntaxToken">
6          <Kind Name="LabelToken"/>
7      </Field>
8      <Field Name="LabelTypeName" Type="LabelNameSyntax">
9      </Field>
10 </Node>
11 <Node Name="LabelNameSyntax" Base="SimpleNameSyntax">
12     <Kind Name="LabelName"/>
13     <Field Name="Label" Type="SyntaxToken">
14         <Kind Name="IdentifierToken"/>
15     </Field>
16 </Node>
```

Figure 3.12. Syntax definition of a type label

compiler platform, we have to expose a `WithIdentifier` method that returns a new instance with the updated identifier. Syntax token key mappings and factory methods for programmatically generating new `LabelName` nodes are also included in the public API, as well as AST traversal methods which we can later leverage during type checking.

After completing those steps, the compiler is able to generate symbols for our new syntax, but still needs to bind those symbols to their internal representation. This is handled by the *Binder* component in the compiler pipeline, which ensures that the various symbols are used in the right places in the source code. In Figure 3.13, we first bind `LabelName` to `BindIdentifier()` in line 7 as part of the `BindExpressionInternal()` method since we chose to treat type labels as identifier keywords. Our `LabelTypeSyntax` should be treated by the compiler as a type, so be bind it to `BindType()` in line 19 in the `BindNamespaceOrTypeOrAliasSymbol()` method to do so. At this point, the C# compiler can parse source code containing type labels and generate an AST from it but attempting to compile the source code to machine code will result in an error since we did not touch the Emitter portion of the pipeline. As well, we had to modify some other portions of the Binder to inhibit errors related to type label binding. This is fine for our purposes since our type checker will later remove all type label information.

3.2.2 Type Checker and Normalizer

In addition to our custom C# compiler, the core of our implementation is in our composite type label type checking tool, which provides compiler functionality related to composite type labels, namely the type checking and normalization of user-defined type labels, as well as the transpilation into regular C# source code of successfully type-checked programs. Our type checking tool is a regular program that calls upon the compiler API provided by Roslyn, including the

```
1  // Binder_Expressions.cs:
2  private BoundExpression BindExpressionInternal(ExpressionSyntax
       node, DiagnosticBag diagnostics, bool invoked, bool indexed)
3  { // ...
4      switch (node.Kind())
5      { // ...
6          case SyntaxKind.LabelName:
7              return BindIdentifier((LabelNameSyntax)node, invoked,
                   diagnostics);
8          // ...
9      }
10 }
11 // Binder_Symbols.cs:
12 internal Symbol BindNamespaceOrTypeOrAliasSymbol(ExpressionSyntax
       syntax, DiagnosticBag diagnostics, ConsList<Symbol>
       basesBeingResolved, bool suppressUseSiteDiagnostics)
13 { //...
14     switch (syntax.Kind())
15     { // ...
16         case SyntaxKind.LabelType:
17         {
18             var node = (LabelTypeSyntax)syntax;
19             var type = BindType(node.ElementType, diagnostics,
                   basesBeingResolved);
20             type = LabelTypeSymbol.CreateLabelTypeSymbol(this.
                   Compilation.Assembly, type, node.LabelTypeName.
                   Identifier.Text);
21             return type;
22         }
23         // ...
24     }
25 }
```

Figure 3.13. Compiler binding code for type labels

new API we exposed as part of our compiler modifications outlined previously. Our type checker design operates in two phases. In the first phase, we perform high-level type checking of type labels to validate composition rules. In the second phase, we normalize variables by refining composite type labels to their concrete type and applying implicit conversions where necessary and possible. This two-phase approach simplifies the implementation and allows us to abort type checking earlier if it fails at a high level, saving us the processing cost of normalization.

We are only interested in type checking user-defined type labels since the regular C# type checker can provide type checking for regular C# symbols. To that end, we extend the `CSharpSyntaxWalker` class provided by Roslyn to create a syntax walker, a method that traverses the AST and collects nodes based on user-defined criteria. Our syntax walker collects any method, the highest-level node we examine for our type checking process, for any usage of type labels.

We then examine all the sub-nodes within those collected method nodes to validate user-defined type label rules. The Roslyn compiler platform provides a `SemanticModel` class that can be queried for type information and additional metadata for any variable via the `SemanticModel.GetTypeInfo(node)` or `SemanticModel.GetSymbolInfo(node).Symbol.OriginalDefinition` method calls. This allows us to extract the type label information that got embedded into the node through our compiler modification, giving us both the built-in C# type and the type label name the variable was defined with. At this point, no modification is performed on the examined syntax tree nodes; we are validating expressions based on their adherence to user-defined composition rules.

The normalization step is more complicated since not only do we need to rewrite nodes, but we also need to keep track of nodes with refined type labels and nodes for which we have deferred processing as part of our normalization algorithm. Additionally, not all concrete type labels are known for composite type labels since they exist on a rule-based basis. For instance, a `@Speed` type label using the `@Length / @Time` rule may have a concrete type label of `@km/hour` which isn't explicitly defined by the user. Instead of exhaustively instantiating all such type labels and incur a memory cost, we are storing such refined types in a symbol dictionary as they are encountered.

Rewriting nodes in the Roslyn platform is done using syntax visitors. Since the AST nodes are immutable, a syntax visitor allows programmers to visit the current node and return a modified instance. Figure 3.14 shows an example of such syntax visitor where we want to apply normalization to a local declaration. The `NormalizeVariableDeclarators` method in turn examines the child nodes of a `LocalDeclarationStatementSyntax`, which are `VariableDeclaratorSyntax` nodes, and returns a list of modified nodes that we use to replace the original variable declarators before returning the modified instance of the local declaration statement node.

Figure 3.15 shows our implementation of our proposed normalization algorithm for binary expression nodes. The `NormalizeBinaryExpressionProposedCore` method is recursively called in lines 3-10 on the child nodes of the binary expression until we have reached its leaves. Currently known normalization data is embedded in the nodes in the form of `SyntaxAnnotation` which is a class provided by Roslyn for the purpose of adding transient annotations to syntax nodes; those annotations are not visible anywhere in the source code and annotation information is automatically discarded. We gather the type labels in use in the currently examined nodes and attempt to find a major label in line 23 for each composite type label in use.

If a major label has been found, we gather the nodes that may require replacement and

```
1 public override SyntaxNode VisitLocalDeclarationStatement(
     LocalDeclarationStatementSyntax node)
2 {
3     if (node.Declaration.Type.Kind() != SyntaxKind.LabelType)
4     {
5         return node;
6     }
7     var rewrittenVariableDeclarators =
        NormalizeVariableDeclarators(node.Declaration.Variables);
8     return node.WithDeclaration(node.Declaration.WithVariables(
        rewrittenVariableDeclarators));
9 }
```

Figure 3.14. Syntax node visitor for local declaration statements

compute the replacement nodes in lines 27-29. Replacement nodes are computes by applying implicit conversion rules explained in previous sections. If an implicit conversion has been applied, we remove the relevant `SyntaxAnnotation` to indicate that the node has been processed and no further rewriting is expected. That `SyntaxAnnotation` is given to the current node, which has the rewritten nodes as children, since if any further implicit conversions are needed they will affect the current node. If the node is composed of several composite type labels, for example a node composed of `@Length` and `@Time`, removing a single `SyntaxAnnotation` may not lock the node for further processing involving type labels of untouched composite type labels. We are however ensuring that for each composite type label attached to a node, there will be at most one implicit conversion applied.

```
1  private ExpressionSyntax NormalizeBinaryExpressionProposedCore(
       BinaryExpressionSyntax node)
2  {
3      var leftPart = node.Left is BinaryExpressionSyntax leftNode
4          ? NormalizeBinaryExpressionProposedCore(leftNode)
5          : NormalizeBinaryExpressionProposedSegment(node.Left);
6      var rightPart = node.Right is BinaryExpressionSyntax rightNode
7          ? NormalizeBinaryExpressionProposedCore(rightNode)
8          : NormalizeBinaryExpressionProposedSegment(node.Right);
9
10     node = node.WithLeft(leftPart).WithRight(rightPart);
11     // ...
12     foreach (var compositeLabel in compositeLabelsInUse)
13     {
14         var concreteLabels = node.DescendantNodesAndSelf().Where(d
               => d.HasAnnotations(compositeLabel))
15             .Select(d =>
16             {
17                 var label = d.GetAnnotations(compositeLabel).
                       Select(a => a.Data).Single();
18                 var score = d.GetAnnotations(label).Select(a =>
                       int.Parse(a.Data)).Single();
19                 return new KeyValuePair<string, int>(label, score)
                       ;
20             })
21             .GroupBy(p => p.Key, p => p.Value)
22             .ToList();
23         var majorLabel = GetMajorLabel(concreteLabels);
24         if (majorLabel == null) continue;
25         var nodeCount = node.DescendantNodesAndSelf().Count(d => d
               .HasAnnotations(compositeLabel));
26         // ...
27         var nodesForReplacement = node.DescendantNodes().Where(d
               => d.HasAnnotations(compositeLabel));
28         node = node.ReplaceNodes(nodesForReplacement, (oldNode,
               ignore) => GetConvertedNode(oldNode, compositeLabel,
               majorLabel))
29             .WithAdditionalAnnotations(liftedAnnotations);
30     }
31     return node;
32 }
```

Figure 3.15. Application of normalization algorithm to binary syntax nodes

Chapter 4

Evaluation

We start the evaluation of our proposed system by qualitatively comparing it to some of the related works presented in Chapter 2, of which we selected the most representative ones. We demonstrate the ease of use of our system by showing how to implement the same kind of user-defined type system using our proposed system and select related systems. In some cases, our proposed system offers features that cannot be reproduced by related works, as we will demonstrate.

Later in this same chapter we present micro-benchmarks to demonstrate quantitative comparisons of the proposed normalization algorithm that we presented with our proposed system against an implementation of the traditional normalization algorithm. We performed a micro-benchmark to show that the use of our proposed normalization algorithm indeed results in a lower number of implicit conversions inserted. As well, we present micro-benchmarks to examine the execution times of different components of the compilation pipeline, as well as the execution time of the resulting programs. Lastly, we present micro-benchmarks where we examine the impact of the selection of the normalization algorithm on the precision of floating point values to measure whether there is any reduction of precision loss.

4.1 Qualitative Comparison

First, we compare our implementation with the Checker Framework to which it shares a number of conceptual elements. One of the important contributions of our system is the flexibility of definition of user-defined type labels, as well as the bounding of their combinatory behavior. As shown in Listings 4.1-4.2, our system has a lower definition burden; with the Checker Framework, the definition of `Time` and its component units `@sec` (second), `@min` (minute) and `@hour` requires more than 200 lines of code which includes about 7-13 lines per type label definition and additional code for explicit conversion definitions and helper classes. For our system only about 3-5 lines of code per type label definition are required, for a total definition size of about 45 lines.

Listing 4.1. Composite type labels

```
1  Label sec {
2  AllowedTypes: uint ulong
       double
3  Canonical
4  }
5  Label min {
6  AllowedTypes: uint ulong
       double
7  ConvertFromCanonical: Divide
       (60)
8  ConvertToCanonical: Multiply
       (60)
9  }

41 CompositeLabel Time {
42 MemberLabels: sec | min |
       hour | msec | nsec | usec
43 CompositionRules: Length /
       Speed
44 }
```

Listing 4.2. Checker Framework

```
1  // class imports

7  @SubtypeOf(UnknownUnits.
       class)
8  @Documented
9  @Retention(RetentionPolicy.
       RUNTIME)
10 @Target({ElementType.
       TYPE_USE, ElementType.
       TYPE_PARAMETER})
11 public @interface Time {}

84 @Documented
85 @Retention(RetentionPolicy.
       RUNTIME)
86 @Target({ElementType.
       TYPE_USE, ElementType.
       TYPE_PARAMETER})
87 @SubtypeOf(Time.class)
88 @UnitsMultiple(quantity=s.
       class, factor=3600)
89 public @interface hour {}

200 // helper classes and
       explicit conversions
```

Furthermore, as discussed in Section 2.3, the Checker Framework does not support implicit conversions or type combinations. Listings 4.3-4.4 show two identical user programs written in our system and in the Checker Framework. The Checker Framework code will result in value and type errors whereas in our system the type labels are correctly converted.

Listing 4.3. Composite type labels

```
1  double@sec sec_value = 5.0;
2  double@min min_value = 5.0;
3  Console.WriteLine(min_value
       + sec_value); //305.0
4  double@sec sum = sec_value +
        min_value; //305.0
```

Listing 4.4. Checker Framework

```
1  @sec double sec_value = 5.0
       * sec;
2  @min double min_value = 5.0
       * min;
3  System.out.println(min_value
        + sec_value); //10 (
       wrong)
4  @sec double sum = sec_value
       + min_value; // type
       error
```

Next, we compare our system with F#, which allows automatic type combinations. As shown in Listings 4.5-4.6, the definition burden is greater with our system, as we must define the `@Workload` composite type label and its combination rules, whereas F# automatically instantiates a new measurement type when we combine a `person` and an `hour` measurement in line 7 of Listing 4.6. However, an invalid combination is also created in line 8. In contrast, that operation correctly results in a type error in line 28 of Listing 4.5 as there is no composite type label or composition rule that defines that behavior. We also use this opportunity to bring back the C++ example presented in Chapter 2, presented here in Listing 4.7, which exhibits similar behavior.

Listing 4.5. Composite type labels

```
1  // People.ls
2  Label person {
3      AllowedTypes: uint ulong
           double
4      Canonical
5  }
6  CompositeLabel People {
7      MemberLabels: person
8  }
9  // Time.ls
10 Label hour {
11     AllowedTypes: uint long
           double
12     Canonical
13 }
14 CompositeLabel Time {
15     MemberLabels: hour
16 }
17 // (other definitions)
18 // Workload.ls
19 CompositeLabel Workload {
20     CompositionRules: People
           * Time
21 }
22 // Program.csl
23 double@person workers = 4.0;
24 double@hour time = 10.0;
25 double@cm distance = 1.0;
26
27 double@Workload workload =
       workers * time;
28 var invalid = workload /
       distance; // type error
```

Listing 4.6. FSharp

```
1  [<Measure>] type person
2  [<Measure>] type hour
3  [<Measure>] type cm
4
5  let workers = 4.0 <person>
6  let time = 10.0 <hour>
7  let workload = workers *
       time
8  let nonsense = workload /
       1.0<cm>
```

Listing 4.7. C++

```
1  // definitions abridged
2  quantity<worker::worker>
       workers(4.0 * worker::
       person);
3  quantity<length> distance
       (1.0 * meter);
4  auto foo = workers /
       distance;
```

We also see the benefits of our system over the F# implementation when it comes to value conversions. In the F# implementation, we cannot implicitly convert between measurement units, and all conversions must be defined. As shown in Listing 4.9, we must explicitly define every conversion pair or explicitly chain conversions in F# to obtain the same output as using our system in Listing 4.8 where implicit conversions are automatically applied, and extra conversion paths are automatically found. Furthermore, similarly to the Checker Framework code in Listing 4.4, data flows of different measurement units cannot be merged; line 12 of Listing 4.9 results in a type error whereas our system properly merges and converts the values to their expected output.

Listing 4.8. Composite type labels

```
1  // Time.ls
2  Label sec {
3  AllowedTypes: uint ulong
       double
4  Canonical
5  }
6  Label min {
7  AllowedTypes: uint ulong
       double
8  ConvertFromCanonical: Divide
       (60)
9  ConvertToCanonical: Multiply
       (60)
10 }

41 CompositeLabel Time {
42 MemberLabels: sec | min |
       hour | msec | nsec | usec
43 CompositionRules: Length /
       Speed
44 }
45 // Program.csl (user code)
46 double@min min_value = 1.0;
47 double@hour hour_value =
       hour_value;
48 double@sec sum = min_value +
        hour_value;
```

Listing 4.9. FSharp

```
1  [<Measure>] type sec
2  [<Measure>] type min
3  [<Measure>] type hour
4  let sec_to_min ( x : float<
       sec> ) = x / 60.0<min/sec
       >
5  let min_to_sec ( x : float<
       min> ) = x * 60.0<min/sec
       >
6  let sec_to_hour ( x : float<
       sec> ) = x / 3600.0<hour/
       sec>
7  let hour_to_sec ( x : float<
       hour> ) = x * 3600.0<hour
       /sec>
8
9  // user code
10 let min_value = 1.0 <min>
11 let hour_value = sec_to_hour
        min_to_sec min_value
12 let sum = min_value +
       hour_value //error
```

To complete our qualitative case study, we now examine the impact of adding a new type label to an existing definition. In this case, we add a new unit of time that Facebook defined called a `flick` [19]. A flick is defined as being 1/705600000 of a second and serves the purpose of expressing a single frame duration for various hertz values used in cinematography. Listings 4.10-4.12 show the definitions of a flick in various systems. In our system, it is sufficient to

define a flick as shown in Listing 4.10 and adding `flick` to the `MemberLabels` clause. In F#, while the definition may seem more concise, due to the lack of implicit conversion support the user code instead becomes more verbose, as previously demonstrated. In the case of the Checker Framework, additional scaffolding is needed in the form of helper classes. As well, user code remains verbose as the Checker Framework does not provide support for implicit conversions.

Listing 4.10. Composite type labels

```
1 Label flick {
2     AllowedTypes: int uint
        double float long
3     ConvertFromCanonical:
        Multiply(705600000)
4     ConvertToCanonical:
        Divide(705600000)
5 }
```

Listing 4.11. FSharp

```
1 [<Measure>] type flick
2 let sec_to_flick ( x : float
    <sec> ) = x *
    705600000.0<flick/sec>
3 let flick_to_sec ( x : float
    <flick> ) = x /
    705600000.0<flick/sec>
```

Listing 4.12. Java

```
1  package org.checkerframework
     .checker.units.qual;
2  import java.lang.annotation.
     Documented;
3  import java.lang.annotation.
     ElementType;
4  import java.lang.annotation.
     Retention;
5  import java.lang.annotation.
     RetentionPolicy;
6  import java.lang.annotation.
     Target;
7  import org.checkerframework.
     framework.qual.SubtypeOf;
8  @Documented
9  @Retention(RetentionPolicy.
     RUNTIME)
10 @Target({ElementType.
     TYPE_USE, ElementType.
     TYPE_PARAMETER})
11 @SubtypeOf(Time.class)
12 @UnitsMultiple(quantity=s.
     class, factor=705600000)
13 public @interface flick {}
```

4.2 Micro-benchmarking Methodology

To quantitatively evaluate our normalization algorithm, we created a micro-benchmark to compare the performance of our algorithm with an implementation of the legacy normalization algorithm, since our normalization algorithm is unique to our system. We wrote a script to generate a large number of source code files of different sizes using composite type label to compile against both implementations. We determine the size by the number of type label variables used in the source code. The source files are generated through a template that first sets an initial value, then performs a varyingly large number of computations on that initial value and finally performs reciprocal operations to revert the variable back to its initial value, as shown below. The end result is then compared with the initial value to measure skew. Figure 4.1 shows the pseudocode

```
let variableCount = {10, 20, 30, 40, 50, 100}
foreach COUNT in variableCount
        set initial value IV to random value
        generate random type labels up to COUNT
        perform up to COUNT number of random operations on IV
        append reciprocal operations to IV to revert it back
        print source file
```

Figure 4.1. Pseudocode for the source code generating template

of the template used.

```
double@min initialValue = 881.084734984247;
double@min foo0 = initialValue;
foo0 = foo0 + foo5 - foo2 * foo3 + foo8 - foo8 + foo2 * foo3 - foo5;
return initialValue - foo0;
```

We then compare the compilation and execution times between the proposed and legacy normalization methods and also provide an evaluation of their impact on the precision of double values. The compilation time is comprised of the time spent by the entire compiler pipeline converting raw source code to machine code. Namely, this includes parsing and lexing, type checking, normalization and translation to machine code. Since up to the normalization step both algorithms share the same pipeline, we are not measuring those components and are focusing on measuring the difference in the time spent for the normalization and machine code translation portions of the compiler pipeline for our micro-benchmarks.

Our micro-benchmark program ran on a Windows 10 Pro 64-bit version 15063.674 environment on a computer with an Intel Core i7-6700T 2.80GHz CPU and 16.0GB of memory. Our micro-benchmark ran iterations on each of 600 generated source code files using up to 100 variable declarations containing type labels.

4.3 Micro-benchmark Results

The blue ticks show the results obtained for the legacy normalization algorithm while the orange ticks show the results obtained for the proposed normalization algorithm. Error bars show the standard deviation.

First, we take a look at the extent of the conversion to validate that our algorithm does result in a reduced number of implicit conversions inserted. As shown in Figure 4.2, the selection of our proposed algorithm does indeed result in a significantly lower number of implicit conversions being inserted on average by the normalization process.

Figure 4.3 shows the results for time spent on the normalization component of our compilation pipeline, measured in millions of ticks. As illustrated by the graph, we observe a clear reduction in normalization time when using our proposed algorithm. The effect of the number of variables using type labels on the normalization time is increasing linearly with our proposed algorithm,

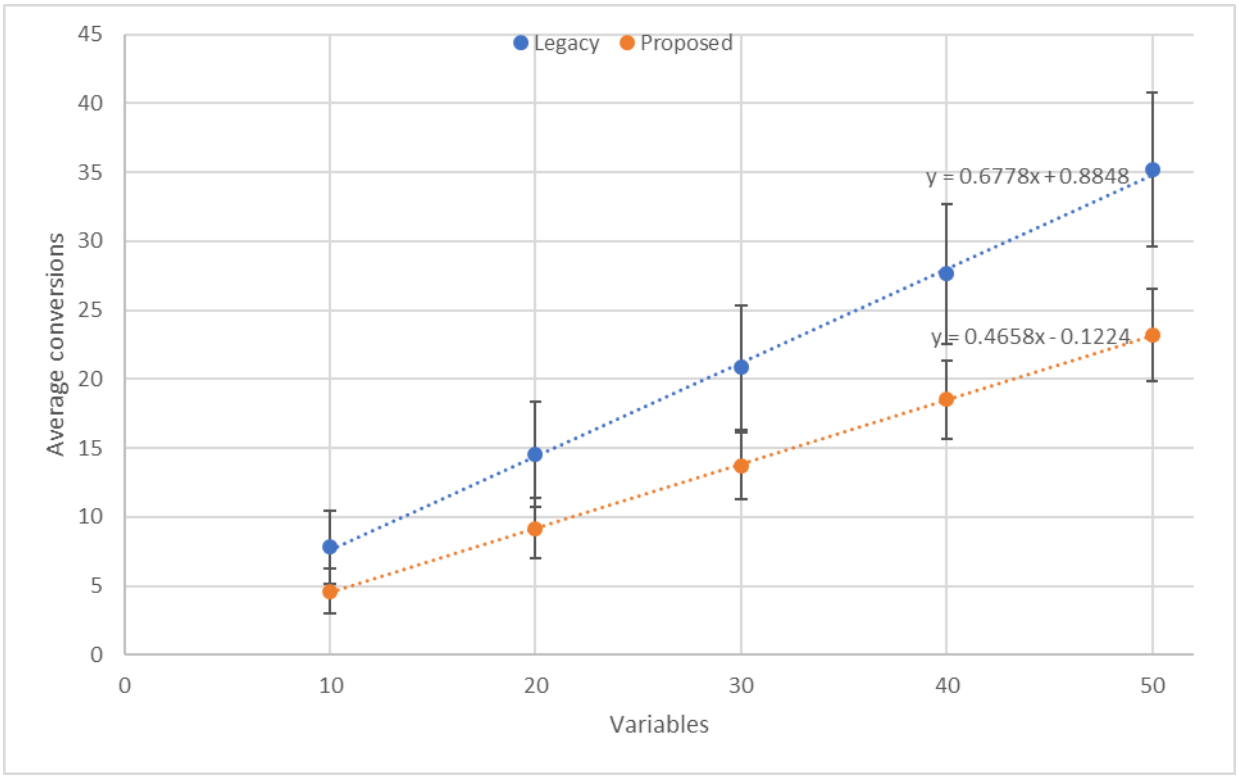

Figure 4.2. Total conversions inserted based on the chosen algorithm

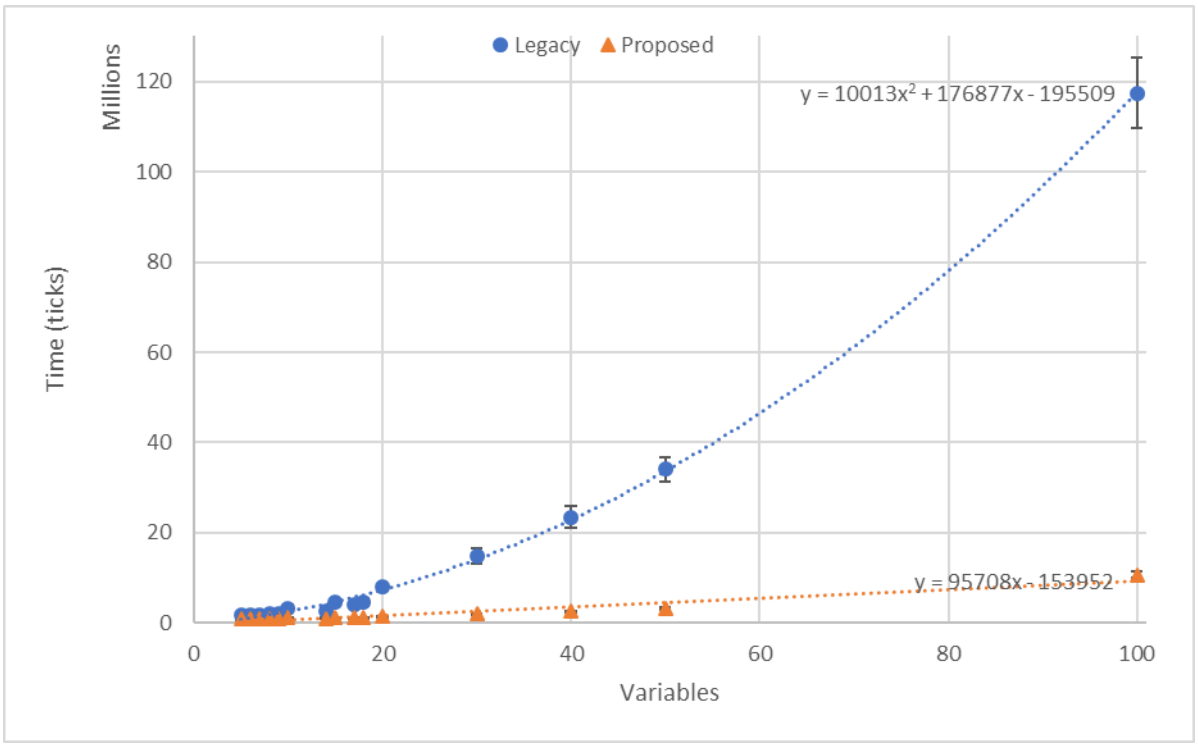

Figure 4.3. Time spent on the normalization component of compilation based on variable count

whereas with the legacy normalization algorithm it is increasing in a polynomial fashion. The reduction in normalization time using our proposed algorithm is expected, as by deferring the insertion of implicit conversions and minimizing them, we are effectively reducing the number of required AST manipulations, which can be quite expensive. Indeed, since the AST is immutable, every time a node has to be rewritten a new instance must be returned.

Figure 4.4 shows the results for time spent on the machine code translation component of the compilation pipeline, measured in millions of ticks. As illustrated by the graph, results obtained for the proposed and legacy normalization algorithms are all within one standard deviation of each other. This indicates that while the choice of normalization algorithm has an effect on the number of implicit conversions inserted into the C# source code during transpilation, this has no significant effect on the time spent to translate to machine code. While our proposed algorithm does not improve machine code emitting time, it also does not negatively impact it.

Figure 4.5 shows the results for execution times of the resulting programs, measured in thousands of ticks. As illustrated by the graph, results obtained for the proposed and legacy normalization algorithms are all within one standard deviation of each other. This again indicates that the choice of normalization algorithm has no significant effect on the execution time once transformed into machine code. This also means that while our proposed algorithm does not provide any improvement when it comes to execution time of resulting programs, it similarly

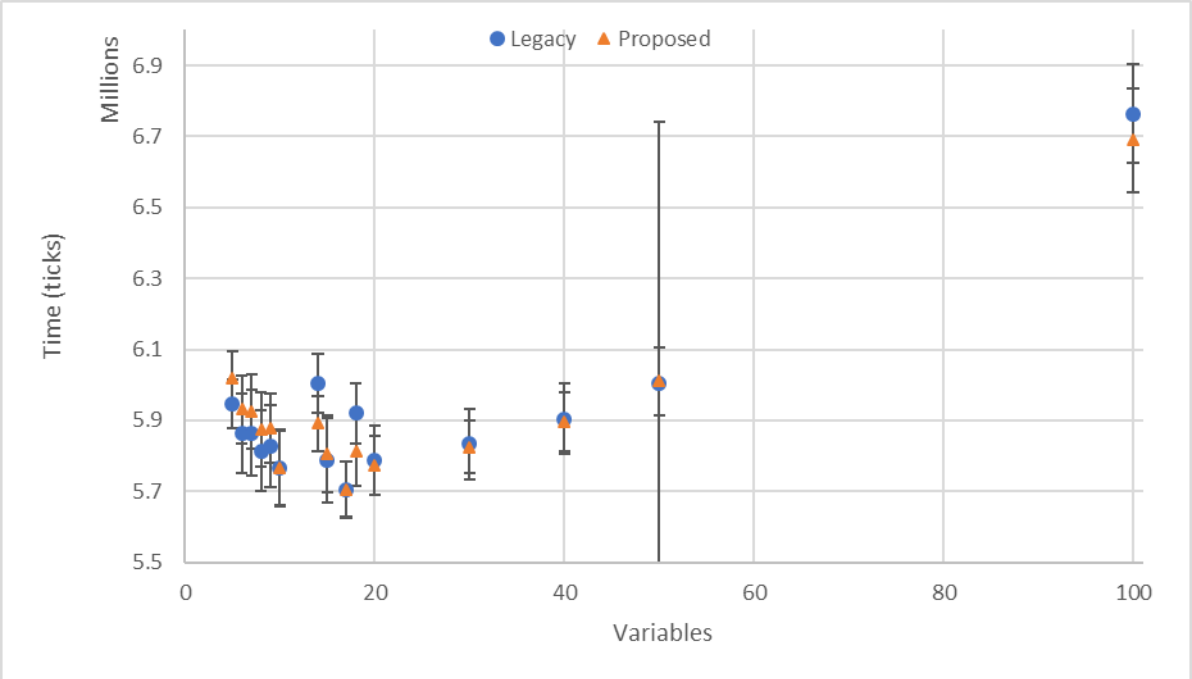

Figure 4.4. Time spent on the machine code translation component of compilation based on variable count

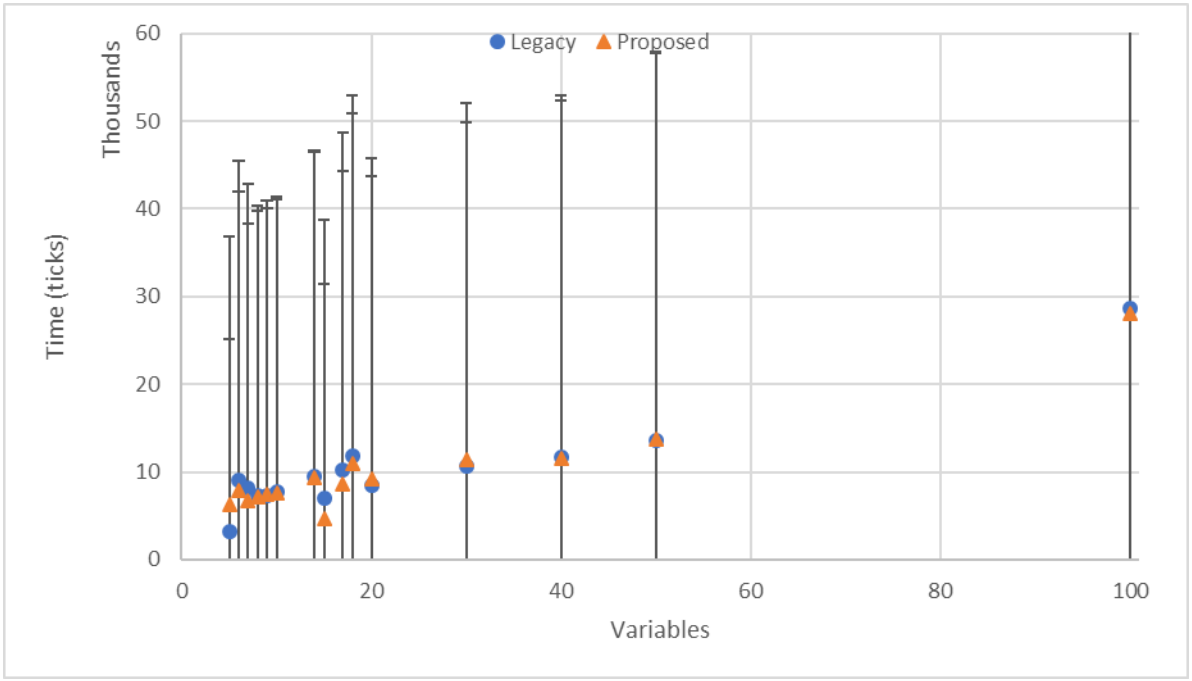

Figure 4.5. Execution time of resulting program based on variable count

does not negatively impact execution times.

4.4 Precision Micro-benchmark Results

For precision micro-benchmarks, we measured the precision loss on a large number of operations involving double values with type labels attached. In the template pseudocode in Figure 4.1, the initial value has a number of operations performed on it that later get reversed. The reason we do that is so that we can compare the final value with the initial value to measure and compare the difference. We can then compare the results between both algorithms to see how often ours performs better.

Here, we made some changes to the template used. Namely, we reduced the number of variables since the legacy algorithm took too long to compile with 100 variables. Furthermore, from a few initial test runs we noticed that the precision gain was not always consistent for purely random operations. We hypothesized that our proposed algorithm suffers less precision loss in programs where normalization yields less repeated divisions or less large divisions.

However, this behavior is not entirely consistent. For example, for the code below repeated divisions in the second line result in a loss of precision compared to using a single division in the third line. This relationship is inverted in the last two lines. This likely has to do with how values are rounded by the compiler.

```
double foo0 = 1980.83299304397;
(foo0 )/1000000000/1000; //1.9808329930439696e-09
(foo0 )/1000000000000; //1.98083299304397e-09

(foo0 )/1000000/1000; //1.98083299304397e-06
(foo0 )/1000000000; //1.9808329930439698e-06
```

We modified our template creation algorithm to attempt to minimize large-scale divisions. This is done by removing very small-scale type labels like `@nm` from the list of available type labels for the algorithm to choose from. Figure 4.6 shows the results we obtain when we run our micro-benchmark against a corpus generated under the above conditions, where we loosely attempt to minimize conversions between distant scales. As the results show there is a significant, within one standard error, improvement from using our proposed algorithm in around 50% of the run cases overall. As the complexity of the source program increases, our algorithm tends to fare less well. We found that our algorithm tends to perform better when the number of large divisions or consecutive divisions can be kept relatively low. This tends to be true for programs with less type labels, where the precision gain is significantly greater using our proposed algorithm. As the number of type label variables increases however, our algorithm performs increasingly worse due to the accumulation of precision error. Since the legacy algorithm greedily converts all type labels to the canonical label, the scale of the conversions tends to be smaller than our proposed algorithm. In the case of our algorithm, since it often defers conversions, the resulting implicit conversions tend to be of larger scales, impacting precision.

We also ran the same micro-benchmarks with varying choices of canonical type labels to evaluate their impact on precision for both algorithms on random input. Source code were generated using the `@Length` composite type label and its member type labels. The generated source code was kept the same in all runs this time, but instead we modified the type label definition file to select a different canonical label for each run. Namely, `@nm` (nanometer), `@m` (meter), and `@km` (kilometer) were selected as a canonical label. Unlike the previous micro-benchmark where we simplified the code samples to minimize large-scale conversions, this micro-benchmark has conversions of randomly varying scales, which is more consistent with real-life scenarios. The results we obtain, shown on Figure 4.7, indicate that the choice of a canonical label doesn't have a significant impact, and that the precision gain obtained from using our algorithm is less pronounced. Indeed, merely changing the canonical label does not completely avoid the occurrence of large-scale conversions. For instance, changing the canonical label to `@km` will still cause a large-scale conversion if we need to convert from a `@nm` to `@um` which result in a $@nm \rightarrow @km$ conversion followed by a $@km \rightarrow @um$ conversion.

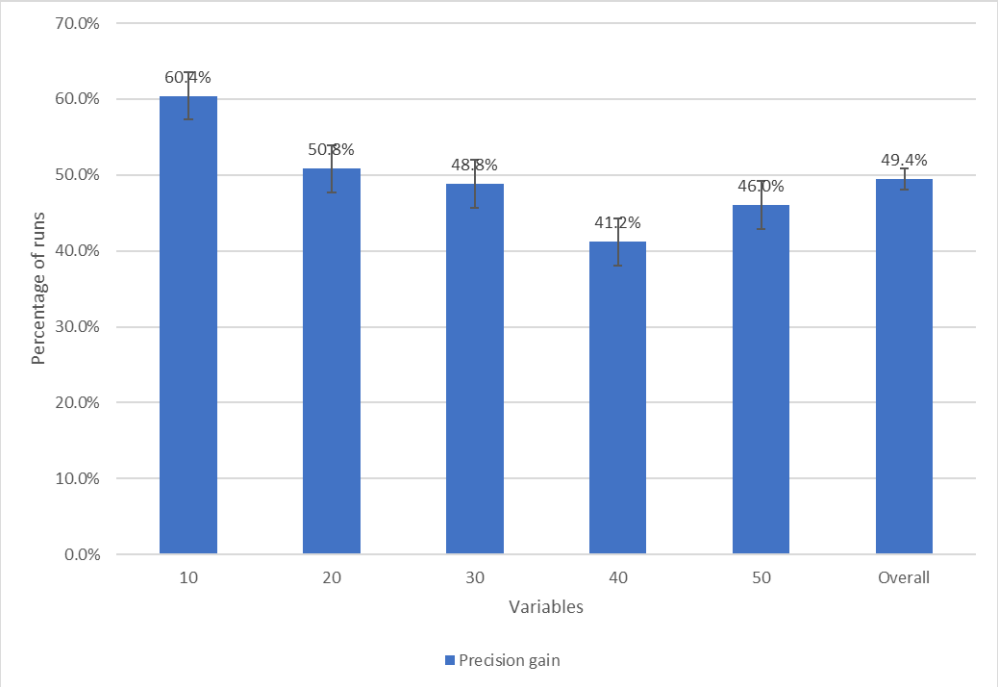

Figure 4.6. Precision gain incurred from using the proposed normalization algorithm

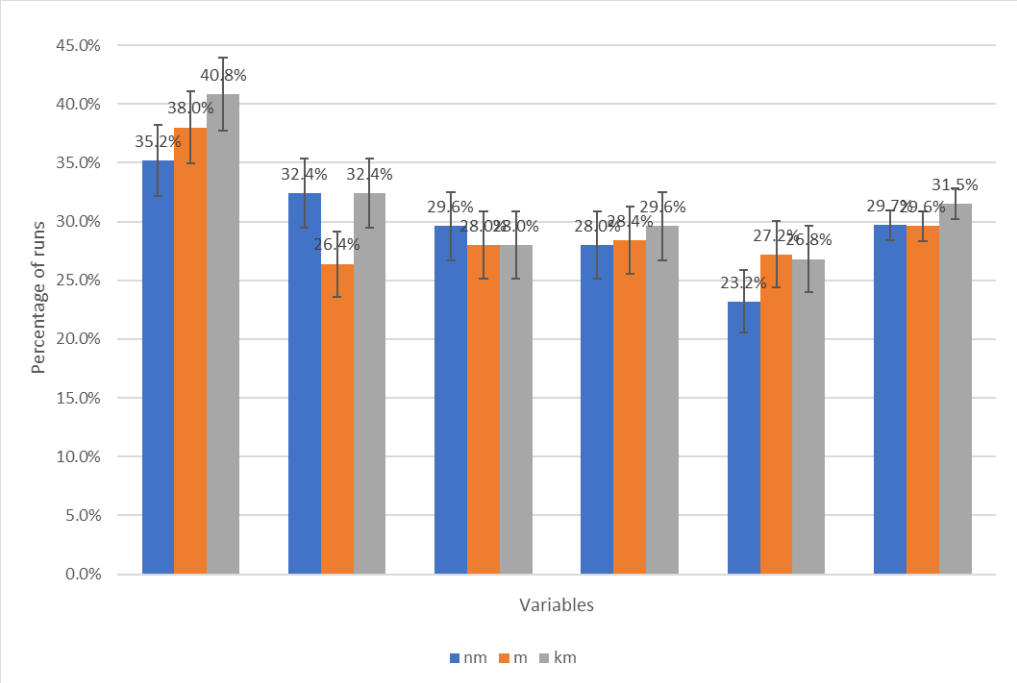

Figure 4.7. Impact of canonical type label selection on precision gain

Chapter 5

Conclusion and Future Work

5.1 Summary

We presented our proposed system, called composite type labels, which addresses some of the issues we found with composability in user-extensible type systems affecting support for handling application specific semantics. In particular, we focused on systems that aim to support units of dimension, one prime example of use-case for composability. Unlike other systems that focus on SI units of dimension, ours provides true support for user-defined dimensions by allowing users to define any units of dimension relevant to their workflows.

Most of the surveyed existing systems exhibited at least one of two categories of problems we have isolated: inter-dimensionality related problems and intra-dimensionality related problems. Inter-dimensionality refers to the support for handling user-defined types belonging to different units of dimensions while intra-dimensionality refers to problems relating to units within the same dimensions.

We presented the three main features of our system, namely composition rules, implicit conversion rules, and type normalization, and describe how their inclusion addressed both inter-dimensionality and intra-dimensionality problems while providing a flexible user-extensible type system and maintaining the safety promises of static type checking by preventing the creation of unpredictably arbitrary combinations. As a result, our system provides better support for user-defined application semantics with minimal additional definition burden imposed on type system designers. Our novel normalization algorithm, tailored to composable user-defined type systems, was shown to improve normalization speed and reduce unneeded implicit conversions.

In the evaluation of our proposed system, we compared our proposed system with some existing ones and demonstrated that ours offer more concise definitions in most cases, while addressing issues we outlined pertaining to the handling of application-specific semantics. We also presented a case study showcasing the ease of extensibility of our system when defining a new unit. In our micro-benchmarks, we found that our proposed normalization algorithm does indeed significantly improve overall compilation time by greatly reducing the amount of time required to perform normalization during type checking. We also found our proposed normalization algorithm to provide less precision loss than the legacy algorithm in some limited scenarios, namely when the number of large-scale conversions can be kept low. As this condition gets harder to maintain

when the number of variables using type labels increase, we see a reduction in precision gain as programs use more type label variables.

5.2 Future Work

We presented in Section 3.1.3 our novel algorithm to improve type normalization in composite type labels and have shown that our proposed algorithm can reduce the number of implicit conversions in binary expressions. Adapting the algorithm to the whole program requires additional work to ensure that the number of implicit conversions remains small in the presence of any other syntax nodes.

As well, in Chapter 4 we found that our algorithm has less precision loss compared to the legacy algorithm only in specific cases, namely situations that avoid implicit conversions having very large divisions or several consecutive division operations. We verified the effect off those factors by running a micro-benchmark against source code generated in a way that aims to reduce their occurrence, showing that our algorithm may need further optimizations to also handle general or truly random cases.

Publications and Research Activities

(1) Antoine Tu, Shigeru Chiba. Outlook on Composite Type Labels in User-Defined Type Systems. 日本ソフトウェア科学会第 34 回大会, 2017.9.19-21.

References

[1] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *Computer Science Handbook, Second Edition*, chapter 97. Chapman & Hall/CRC, 2004.

[2] Gilad Bracha. Pluggable type systems. In *In OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[3] KarolS. https://github.com/karols/units, 2016.

[4] Matthias C. Schabel and Steven Watanabe. Boost.units, 2008.

[5] Andrew Kennedy. *Types for Units-of-Measure: Theory and Practice*, pages 268–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[7] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *In Scheme and Functional Programming Wworkshop*, pages 81–92, 2006.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Pldi 2002: Extended static checking for java. *SIGPLAN Not.*, 48(4S):22–33, July 2013.

[9] Dana N. Xu. Extended static checking for haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA, 2006. ACM.

[10] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, October 2006.

[11] Shane Markstrum, Daniel Marino, Matthew Esquivel, and Todd Millstein. T.: Practical enforcement and testing of pluggable type systems. Technical report.

[12] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., March 2008. Version 1.0.

[13] Lingxiao Jiang and Zhendong Su. Osprey: A practical type system for validating dimensional unit correctness of c programs. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 262–271, New York, NY, USA, 2006. ACM.

[14] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.

[15] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to c: The ablec extensible language framework. *Proc. ACM Program. Lang.*, 1(OOPSLA):98:1–98:29, October 2017.

[16] Travis Carlson and Eric Van Wyk. Type qualifiers as composable language extensions. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 91–103, New York, NY, USA, 2017. ACM.

[17] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 694–705, New York, NY, USA, 2017. ACM.

[18] Microsoft Corporation. https://github.com/dotnet/roslyn.

[19] Facebook Inc. https://github.com/oculusvr/flicks.

Acknowledgements

I wish to express my sincere gratitude to my supervisor, Professor Shigeru Chiba, whose guidance, encouragement, patience and insight were indispensable to the completion of my master's research.

I also want to extend my gratitude to my colleagues at the Core Software Group, in particular Kazuhiro Ichikawa and Tetsuro Yamazaki, who imparted on me some of their vast knowledge and broadened my horizons in the field of computer science. I also want to thank Daniel Perez, who helped me conquer the monster that is LaTeX.

Last but not least, I want to thank my mother and late father, without whose support I would not have been able to embark on what turned out to be the greatest adventure of my life.