Outlook on Composite Type Labels in User-Defined Type Systems

Antoine Tu　Shigeru Chiba

This paper describes an envisioned implementation of user-defined type systems that relies on a feature we call composite type labels for type-checking. Desiging a type system is a complex task, and while some user-defined type systems exist to address the need for use-case specific types, those systems usually have a high level of verbosity and compromise on expressiveness to ensure safety. Our preliminary approach addresses these problems through automating the instantiation of additional type labels, the derivation of implicit conversion rules and the resolution of ambiguities during type-cheking as necessary. To achieve that we rely on composite type labels, that are defined by composition rules. We first begin by identifying some key problems with current approaches, after which we demonstrate how our planned system solves them.

1 Introduction

In this paper, we present a system for type-checking user-defined type labels that relies on composition rules to reduce the cognitive load on programmers while ensuring a high level of expressiveness in declaring new type labels. Our approach uses those rules to automatically instantiate additional type labels when appropriate. Similarly, our system automatically derives implicit conversions between instantiated type labels and performs type normalization where necessary.

A programming language's built-in type system only provides a small set of primitive types that are suitable for general use-cases. In some applications however, use-case specific types are useful to prevent type-related errors that are not handled by the general-purpose built-in type system. The idea of allowing programmers to bring in their own type system to enhance the existing one has already been proposed by Bracha [1] and is called user-defined or pluggable type systems. Current implementations of this idea only implement it partially and in a restricted fashion, as it is challenging to open up the type system while maintaining the safety promises of the overall system.

We believe that for a type system to be useful, it must provide users with a high level of expressiveness while minimally increasing the cognitive load required to define and use such type system. To achieve this, we extend the idea of user-defined type systems by proposing composite type labels. A composite type label is a type label that is defined by the composition of traditional type labels or other composite type labels through user-defined composition rules. Our contribution is the automatic instantiation of additional type labels using type label composition rules, implicit conversion support for pluggable type systems using path finding, and the automatic resolution of flow merging ambiguities through the automatic application of type normalization. We believe these contributions enhance existing pluggable type systems by reducing the cognitive load on programmers while maintaining an equal or higher level of expressiveness compared to existing systems.

2 Motivation

User-defined or pluggable type systems [1], as defined by Bracha, have been introduced to address the need to support use-case specific type systems in addition to a host language's built-

コンポジットタイプレーベルによるユーザ定義型システム
ツ アントアン　千葉滋, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

in type system. Their implementations do so by providing a framework that allows users to define additional attributes, which we will refer to as type labels, that can be attached to primitive types to provide additional information to the type checker. Type-checking is then performed against those user-defined type labels on top of the regular type-checking that occurs in the host language. This allows users to bring their own type system to the host language. For instance, the Checker Framework [4] described by Papi et al. accomplishes this by providing a compiler extension to the Java programming language that parses user-defined type labels implemented through Java type annotations.

2.1 Reducing definition burden

Those systems however are subject to several constraints to ensure that users do not introduce unexpected behavior to the host language. Indeed, they often demand a high level of verbosity on the part of the programmer, and provide low expressiveness. The previously introduced Checker Framework, for instance, requires every type label to be explicitly defined and doesn't provide support for combining type labels.

As an example, if we wanted to implement a type system to express units of measurement, current systems would require us to explicitly define all units of length (*m*, *km*, *cm*, etc.) and time (*sec*, *min*, *hour*, etc.) as well as all possible combinations of length and time (*m/sec*, *m/hour*, *km/hour*, etc.). Furthermore, current systems do not provide a way to relate composite types such as *km/hour* with the types labels they are composed of (*km* and *hour*). For all intents and purposes, current systems treat them as completely unrelated types.

The problem with this approach is that due to its high verbosity it increases the cognitive load on the programmers who want to define type labels. To support more complex user-defined type systems, we therefore need to provide a way of reducing definition burden.

2.2 Implicit conversions

Implicit conversions are useful in a type system as they reduce the need for programmers to explicitly annotate conversions, but they are usually avoided by current user-defined type system approaches due to the additional level of complexity its support introduces. Indeed, the Checker Framework does not support rewriting values and therefore has limited support for implicit conversions. Defining all possible implicit conversions for a type system consisting of n type labels would be an effort of the order $O(n^2)$. We therefore want to have a mechanism for defining implicit conversions with a low definition burden.

```
1  double@m foo = 5;
2  double@mm bar = 59;
3  double@Length baz = foo + bar;
```

Fig. 1 Flow merging in Checker Framework presented using C# syntax

2.3 Resolving ambiguity

Current systems like the Checker Framework allow the grouping of type labels into something similar to a subtyping or subclassing relationship with a parent or abstract type label. For instance, a programmer can define type labels m (meter) and mm (millimiter) as being concrete incarnations of the *Length* abstract type label. A variable declared as *Length* may take on a label m or mm later on through a process the authors call type refinement which assigns to a variable a more restrictive type label if such label is found to exist [4]. However, this process only applies from the abstract type label to a concrete one, and doesn't allow for interactions between concrete type labels.

Indeed, merging two data flows of different type labels results in a compiler error even if both type labels are related. For instance, in Figure 2.3 we present a simple Checker Framework example, that we have rewritten in C# syntax for consistency with the language used in this paper. In this example, if we try to add two units of *Length*, namely m and mm, the compiler throws an error since it is unclear which type label this operation should result in. However, it is necessary in our composite type label implementation to handle such ambiguities. Indeed, since we automatically generate additional type labels, we cannot rely on the users to explicitly specify how to solve every ambiguous case.

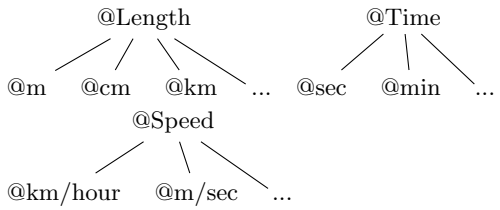

Fig. 2 Composite type labels

```
1  Label m {
2      AllowedTypes: uint ulong
           double
3      Canonical
4  }
5  Label cm {
6   AllowedTypes: uint ulong double
7   ConvertFromCanonical: Multiply
       (100)
8   ConvertToCanonical: Divide(100)
9  }
10 CompositeLabel Length {
11     MemberLabels: m | km | cm
12     CompositionRules: Speed *
           Time
13 }
```

Fig. 3 Definition of labels and composite type labels

3 Composite Type Labels

A composite type label is a type label that is defined by the composition of traditional type labels or other composite type labels through user-defined composition rules. It can be seen as similar to an abstract type which can take the instance of any type label in its definition. The concrete instance is selected automatically during type-checking. Figure 3 shows the *Length*, *Time* and *Speed* composite type labels and their members. New type labels and composite type labels are defined using a provided external DSL for this purpose, as illustrated in Figure 3.

```
1  int@m foo = 5;
2  int@sec bar = 2;
3  int@Speed foobar = foo/bar;
4  int@Length baz = foobar * bar;
```

Fig. 4 Sample code using composite type labels

3.1 Composition rules

Composition rules define the legal combinations of type labels or composite labels, providing hints to the type-checker. They are defined by using the *CompositionRules* keyword as shown on line 12 of Figure 3 where we declare that a *Length* composite type label can be composed of the multiplication of a *Speed* and *Time*. Figure 3.1 shows sample C# code that uses those type labels. In lines 3 and 4, composite type labels are used. During type-checking, line 3 will resolve to m/sec while line 4 will resolve to m, according to the respective composition rules of the declared type labels.

3.2 Implicit conversion rules

Implicit conversions are implemented by resolving conversion paths. Indeed, a programmer only needs to select a type label to be a canonical representation, and provide conversion rules to and from the canonial type label. In Figure 3, we declare m to be a canonical type label, and declare conversions for *cm* in lines 7 and 8. Based on those rules, we define that there is an implicit conversion between a type label A and B if there exists a conversion path from A to B that goes through their respective canonical type labels and can automatically resolve additional implicit conversions from the minimal set provided.

This approach is inspired from the Checker Framework, which provides an interface to relate sub-units like mm and km to the base unit m. However, it does not support modifying values during type-checking, and the sample code in Figure 3.2 will type-check, but the result will be erroneous as the run-time semantics are different than those during compile-time. In this example, line 3 will yield a value of 10.0 instead of the correct value of 5.05.

In more complex cases such as composite type la-

```
1  double@m foo = 5;
2  double@cm bar = 5;
3  Console.WriteLine(foo + bar);
```

Fig. 5 An example where implicit conversion is needed

bels composed of two or more type labels, we need to perform that conversion path finding exercise on all of the type labels that compose the type. For instance, to convert from a $km/hour$ composite type label to a m/sec type label we will need to find conversion paths joining km to m as well as $hour$ to sec. This might lead us into edge cases such as where the conversion rules defined by the programmer yield a path from km to m but not $hour$ to sec. In this case, a conversion from $km/hour$ to m/sec will not be possible. As we cannot make a judgement on whether this is intended behavior or a definition omission on the part of the programmer, we will reject this code and generate a compiler warning.

In allowing programmers to define conversion rules, we need to prevent them from introducing unintended behavior. Some general-purpose languages such as C# that allow programmers to define their own implicit conversions and put that responsibility on the them. In our case, since our system automatically resolves conversion paths it would be counter-intuitive to rely on programmers to predict inconsistent behavior. For this reason, we chose to restrict the rule definition by only allowing programmers to define conversion rules by calling built-in conversion methods in our provided DSL, as illustrated in lines 7-8 of Figure 3. These methods can be chained, allowing more complex conversion rules to be composed.

3.3 Type normalization

We perform type normalization automatically to solve the ambiguity problem presented in Section 2.3. Such normalization is also needed for our type-checker to validate a variable's type label against composition rules. During type checking we examine the AST (Abstract Syntax Tree) for nodes where type labels are being used. We validate expressions by traversing their left and right subtrees and comparing them to type label composition rules, which we store internally as a tree structure. In more complex expressions where the left or right subtrees are deeply nested, we perform such comparisons on the smaller subtrees and normalize them.

In order to resolve ambiguous cases like the one presented in Figure 2.3, we use the canonical type label as a default type label, and will fall back to it if no better candidates exist. In this case the *bar* variable on line 3 will be implicitly converted from mm to m and the resulting type label for *baz* will be m. In more complex cases where a simple fallback may not resolve all ambiguities, we rely on the order of definition of the type labels.

4 Related Works

Papi et al. have introduced the Checker Framework, which is a pluggable or user-defined type system implementation in Java which relies on Java annotations to provide additional information to the type-checker [4]. However, it doesn't support automatically instantiated type labels and thus comes with a high level of verbosity. As well, it cannot rewrite or change the value of a type during type-checking and therefore does not support implicitly converting variables during type-checking to resolve ambiguities when merging data flows.

The JavaCOP framework introduced by Markstrum et al. behaves similarly to the Checker Framework in that programmers write additional checkers whose goal is to enhance the built-in type system [3]. One improvement JavaCOP had over the Checker Framwork is an enhanced capability for dataflow analysis. It however suffers from the same downside of not supporting type label combinations.

It is also worth mentioning that the F# programming language provides support for user-defined measurement units, albeit only for numerical types [2]. Those act like type labels in the sense that using them adds some label metadata to the variable, allowing the type-checker to perform some additional consistency checks. The interesting part of F#'s implementation is that it provides some elementary support for combining labels and instantiating new ones. However, such label combination is performed liberally, without any regard to user

intent. While we also feature measurement units in our code examples throughout this paper, our system aims to support arbitrary user-defined type labels and therefore an unbounded combinatory behavior is not appropriate for general use.

5 Conclusion

In this paper we introduced composite type labels, which is a new class of type labels that is defined recursively by traditional type labels and composition rules. We further elaborated how we use those composition rules to automatically instantiate additional type labels as necessary, infer implicit conversion rules based on path finding and resolve ambiguities in user code by automatically performing type normalization. In doing so, we reduce the cognitive load imposed on programmers by existing user-defined or pluggable type systems while providing them with a higher level of expressiveness in the definition of type labels.

References

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

[2] Kennedy, A.: *Types for Units-of-Measure: Theory and Practice*, Springer Berlin Heidelberg, 2010, pp. 268–305.

[3] Markstrum, S., Marino, D., Esquivel, M., and Millstein, T.: T.: Practical enforcement and testing of pluggable type systems, Technical report.

[4] Papi, M. M., Ali, M., Correa, Jr., T. L., Perkins, J. H., and Ernst, M. D.: Practical Pluggable Types for Java, *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, New York, NY, USA, ACM, 2008, pp. 201–212.