Extending Type Systems in a Library

Introduction
Hi, my name is Yuriy, I’m a graduate student at Texas A&M University and I’m going to present our work on refining type systems as a library. This collaborative work was done by me, Dr. Jarvi and Esam Mlaih.

Motivation
Type systems proven to be useful in many programming languages and application domains. They are typically used to carry information about certain properties of values, their interaction with each other and possible uses. This information lets compiler perform better optimizations as well as makes sure certain types of bugs never appear in the program. Type systems are typically closely tightened to the language design and are a fixed part of a compiler as a result. There is no way to extend it to every possible domain, however there are many domains whose type systems require only slight modifications to the type system of an existing programming language. Such refined type systems could in principle guarantee important run-time invariants for their domain abstractions, but these abstractions are not modeled as part of the type system of the programming language used.

Use cases
Examples of such type systems were well studied in the literature and include tracking of physical units, semantic properties like sign, immutability, data races and structural properties similar to those of regular expression types.

Goal
Building a dedicated compiler or even a preprocessor is quite expensive and often remains a research project. Besides, when it is built for a particular domain, it often suffers problem of integration: an application may need to work with several domains and tools should be able to work together. Due to sensitivity to a particular syntax, this goal may be difficult to achieve.
In this paper, instead of a special purpose framework, we advocate a more lightweight mechanism for refining type systems with domain-specific abstractions: as software libraries. Our goal is to explore how far a pure library solution suffices to extending a type system. Ideally, this solution should allow interoperability of different abstractions, but in the worst case would at least allow using of different abstractions separately in the same program since only standard language features are used, so no problems on the level of syntax at least. It has some disadvantages though: less customized error messages and inability to express certain type systems like, for example, flow-sensitive type constructs.

Contributions
In this work, we tried to implement in C++ two distinct type systems and then extract useful commonalities into a dedicated framework, which one may use to create other
typing environments of interest. In particular, we implement general type qualifiers as well as limited form of regular expression types for working with XML data.

**Example**
In the following example function `current_height()` returns only positive values, but type system neither allows us to attach this property to the result type of a function, nor to carry it with variable that will accept the result. Due to the knowledge of domain we know that b will always be greater than a and as a result b-a should be positive unless there is a bug in program. Still, type system does not know anything about it and can neither warn us nor make sure there is no bug in subsequent call. For the same reason it cannot be sure that square root won’t be called on a negative value in the last case, even though in this case it would have been possible should the language provided support for tracking positiveness of a value.

**What do we need?**
Regardless of whether we are creating a new or extending an existing type system, we need some means to define typing rules, evaluation rules and subtyping relation. Typing rules show how to synthesize complex types out of simple ones, evaluation rules explain compiler how allowed constructs should be evaluated, while subtyping relation allows to build taxonomy among our abstractions.

**How do we achieve that in C++**
To achieve this in C++, we use class templates for type construction, function templates and overloading for evaluation and a set of dedicated meta-functions to deal with subtyping.

**Useful building blocks**
We note that several existing C++ libraries came in very handy in extending various type systems as they generalize in a library certain constructs, available in other languages only as a language feature. In particular, Fusion tuple greatly simplifies support of record types, Boost variant helps in defining union types, enable_if lets overload on arbitrary rather than structural properties of types, while MPL serves a great job in unifying compile-time computations. We point out that MPL, Fusion, Variant, and enable if, have all been developed using the generic programming methodology, building their interfaces to a large extent against common concepts. As a result, the above libraries are highly interoperable.

**Type qualifiers**
The first type system we looked at is type qualifiers. Type qualifiers allow attaching additional semantic properties to types without changing semantics of existing operations on those types. A theory of type qualifiers has been studied in the literature and many qualifiers of practical need have been developed. Among them qualifiers for tracking memory usage errors, detecting format string security vulnerabilities, keeping track of positive and negative values, ensuring that user pointers are never dereferenced in kernel space, preventing data races and deadlocks, and so forth.
**Example: Qualifiers’ Hello World**
The following example shows how to declare new qualifiers in our framework. Depending on subtyping relation of qualified and unqualified type qualifiers are divided into positive and negative type qualifiers. We then can define how operations carry those properties along. We then declare variables by attaching all necessary properties to the type. Note, that the order of qualifiers doesn’t matter and types that differ only in order are considered to be semantically the same. In some cases, like pos/neg qualifiers those properties may have to be verified at run-time. As can be seen, multiplication carries nonnull property as product of two nonnull values is nonnull as well as it derives that result is negative since one argument is positive and another is negative. In the following case we get an error because difference of two nonnull values can be zero, so attaching nonnull to the type of result is not safe. This happens because nonnull is negative qualifier. Tainted, which is a positive qualifier can be added to the type of variable even if it is not present in the RHS. But once added, it cannot be dropped.

**Example**
The following example demonstrates a function definition that only accepts positive values. We achieve this by restricting accepted types to be subtypes of pos<double>. We use subtype_cast to convert value of a subtype into a value of a supertype. We then have a function that returns positive untainted values (values that come from measurements for example). We declare a variable to hold all (or some of those properties) and then use it to safely pass the value into our function.

**Qualifiers summary**
As you can see type qualifiers can be very useful in tracking some properties, not trackable by the type system of the programming language. They are easy to define in our framework and easy to use, however our approach can not handle all possible types of qualifiers. For example it can not handle flow sensitive type qualifiers and reference qualifiers like for example those dealing with aliasing.

**Type system for XML**
Another type system we looked at employs types to describe XML elements with certain structure. These types can be mapped to XML Schema definitions of elements and can be used to guarantee that application performs only value preserving conversions as well as produce only those XML snippets that are conformant to their Schema definition.

**Example**
The following example demonstrates some XML Schema definitions and C++ types they are mapped to. C++ definitions are generated automatically from corresponding Schema definitions.

**Example**
As we’ll show in this example, generated definitions can be used out-of-the-box to parse and synthesize XML snippets that correspond to a particular type as well as perform structural conversion that preserve original values.
**XDuce**

Type system that we just glanced at was originally designed as a part of a programming language XDuce. It defines a type to be set of sequences over a certain domain and uses regular expressions to compose new types. They define a subtyping relation to be an inclusion relation on sets represented by types, which means that type A is a subtype of a type B if and only if set of sequences represented by type A is included into set of sequences described by type B.

**C++**

In our C++ representation of those regular expression types we represent concatenation of types with a tuple, alternative with a discriminated union and repetition with a vector of tuples. We represent type construction through a templated class, that accepts both: a tag-type and a data-type. Because of its significant complexity we currently do not provide support for general recursive types, however we believe that support of concatenation, alternative and repetition is functionally equivalent to restricting recursion to tail recursion only, which is an assumption used by XDuce.

**XTL**

We summarized our efforts of implementing the above type systems through a creation of a dedicated framework that can be used in extending C++ type system to other domains. It fixes an interface one may use to define a domain-specific subtyping relation as well as provides some ready definitions that can be used to extend this relation to subtyping of array and function types, commonly used in many type systems.

**Limitations**

In the course of our experiment, we noticed several limitations that library approach has. We currently require all new type constructs state reflexivity manually as providing a specialization of is_subtype for general reflexive case typically conflicts with other subtyping statements. There is also no implicit transitivity of subtyping relation and it has to be taken care of by specializing all desired cases. For the same reason our join meta-function that returns join of two types is not guaranteed to return supremum, but can rather return an arbitrary upper bound.

**Future work**

In the nearest future, we would like to try applying our approach to ownership types as well as look at alternative representations of type qualifiers. Our current implementation of subtyping on repetitions is rather basic, so we would like to close this gap as well.