Science of Computer Programming 76 (2011) 290–306

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Extending type systems in a library: Type-safe XML processing in C++

Yuriy Solodkyy *, Jaakko Järvi

Texas A&M University, College Station, TAMU 3112, TX 77843, USA

ARTICLE INFO

Article history:
Received 3 March 2007
Received in revised form 27 May 2010
Accepted 19 September 2010
Available online 20 October 2010

Keywords:
Type systems
XML
Type qualifiers
C++
Template metaprogramming
Active libraries

ABSTRACT

Type systems built directly into the compiler or interpreter of a programming language cannot be easily extended to keep track of run-time invariants of new abstractions. Yet, programming with domain-specific abstractions could benefit from additional static checking. This paper presents library techniques for extending the type system of C++ to support domain-specific abstractions. The main contribution is a programmable "subtype" relation. As a demonstration of the techniques, we implement a type system for defining type qualifiers in C++, as well as a type system for the XML processing language, capable of, e.g., statically guaranteeing that a program only produces valid XML documents according to a given XML schema.

1. Introduction

It is in general not possible to decide statically the exact set of all safe programs (programs whose behavior for all inputs is specified by the language semantics). Type systems of practical programming languages can only approximate this set, rejecting some safe programs, and accepting some unsafe ones. For example, the **if** statement below is rejected by a C++ compiler, even though the type-incorrect execution path would never be taken, and the initialization of j is accepted, even though i will always lead to a "division by zero" error:

```
int i = 1;
if (i == 1) i = 0; else i = "error";
int j = 1/i;
```

Replace i == 1 in the condition with an arbitrarily complex computation, and it is evident why practical type systems have this behavior: it is too inefficient, or impossible, to statically keep track of computations with certain abstractions to guarantee safety. There are, however, many abstractions for which ensuring their safe use with a type system would be neither inefficient nor impossible. Consider the following piece of code, accepted by a C++ compiler:

```
double w; // width in cm
double h; // height in m
  ...
double p = 2 * (w + h); // perimeter; oops, incompatible units!
```

The variables obviously correspond to physical quantities, but the *units* of those are outside of the type system, and the easy error goes undetected.

* Corresponding author. Tel.: +1 979 739 3587.
E-mail addresses: yuriys@cse.tamu.edu (Y. Solodkyy), jarvi@cse.tamu.edu (J. Järvi).
URLs: http://parasol.cs.tamu.edu/~yuriys/ (Y. Solodkyy), http://parasol.cs.tamu.edu/~jarvi/ (J. Järvi).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.09.004

There are numerous domain-specific abstractions for which type systems could in principle guarantee important run-time invariants—but the abstractions are not modeled as part of the type system of the programming language used. Of course, many type systems for domain-specific abstractions have been developed. For example, type systems for rejecting incorrect computations with physical quantities, such as the one in our example above, can be found [29]. As other examples, there are type systems for tracking memory usage errors with a *non-null* annotation [14,16,17], automatically detecting format-string security vulnerabilities [37], keeping track of positive and negative values [11], ensuring that user pointers are never dereferenced in kernel space [28], preventing data races and deadlocks [7], and so forth. All of the above type systems can be based on annotating types with different kinds of *type qualifiers*, and tracking their use in expressions.

We note that none of the above type systems have found their way to mainstream languages. Whether programmers can benefit from such type systems becomes a question of whether the abstractions involved are common enough and safety properties important enough to warrant complicating the specification of a general-purpose programming language and the implementation of its compilers and interpreters. It is clear that programming languages cannot be extended to support typing disciplines for every possible domain. Ideally, it would be possible to *extend* type systems to guarantee run-time invariants of new domain-specific abstractions.

Work towards extensible type systems exists. Chin et al. [11] share our view that language designers cannot anticipate all of the practical ways in which types may be refined in a particular type system in order to enforce a particular invariant. The proposed solution is a framework for user-defined *type refinements*, allowing programmers to augment a language's type system with new type annotations to ensure invariants of interest. The framework allows the generation of a type-checker based on declarative rules. Other work with similar goals includes that of optional, "pluggable" type systems [8]. While clearly beneficial, the above kind of framework has not yet found widespread use.

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. We show that most of the type refinements presented, for example, in [11,17,32], and available through dedicated frameworks can also be implemented as a library in a general-purpose programming language, namely C++. Our approach is therefore to refine the C++ type system with domain-specific abstractions via libraries. The underlying C++ type system cannot obviously be altered—by *refining* the type system we primarily mean defining the convertibility relations between data types of particular domains, and how these user-defined data types behave with respect to the built-in types of C++. The approach is constrained by what can be expressed in a library, and thus some capabilities of special purpose frameworks are not offered. For example, some frameworks [11] ensure the soundness of the generated type systems, which our library solution does not automatically guarantee.

Libraries taking the role of the type-checker have been proposed before. For example, C++ libraries for tracking physical units are presented in [4,9]. The introduction of several recent programming techniques and foundational C++ libraries, however, enables a more disciplined approach to defining such type system refinements—such that separately defined refinements compose. In this paper, we collect these techniques together, and show how to apply them for refining the C++ type system. Our contributions are:

- We identify the necessary library tools for extending the C++ type system for domain-specific types and abstractions.
- We identify the necessary language features of C++ that enable the definition of an arbitrary "subtyping" relation (in quotes, since a conversion may be necessary; we omit the quotes from now on).
- We provide a library of primitives for easy extension of the C++ type system with user-defined (sub)typing rules.
- We demonstrate with two extensive examples: a type system for building type-qualifiers and a type system for XML documents. The latter can, for example, guarantee statically that a program only produces XML documents that are valid according to a given XML schema.

The library and all examples are available for download [39].

To whet the appetite, consider the following example using our XML type-checking library:

```
typedef alt<seq<Name, Email>, seq<Name, Tel> > old_contact;
typedef seq<Name, alt<Email, Tel> > new_contact;
  ...
new_contact n = old_contact();
```

The alt and seq types represent, respectively, alternation and sequencing of XML elements while Name, Email, and Tel types represent particular XML elements. Thus, old_contact and new_contact are types of objects that represent fragments of XML data. We discuss these types in detail in Section 4.2. Our library statically assures that the two XML types are compatible and that the initialization of n with an object of type old_contact is safe, and generates the necessary code to conduct such a transformation. With our library, arbitrarily complex XML schemas can be represented as C++ types. These types provide static guarantees about dynamic content of their values, can aid in parsing conforming XML documents, and provide safe conversion operations between fragments of XML data.

2. Necessary building blocks for type system refinements

To be able to refine a type system in a library, several capabilities are required of the host language: first, the host language must support some form of metaprogramming, that is, the definition and evaluation of compile-time computations; second,

a representation of types needs to be accessible to metaprograms; and third, the host languages should support non-intrusively grouping types into different classes defined by metaprograms, and then defining operations and functions that work for types belonging to one or more of such classes. The toolbox of a C++ programmer has grown significantly during recent years, and can support these key capabilities. Below, we identify several techniques and libraries that are invaluable for defining type refinements.

2.1. Language for metaprogramming

The ability to express interesting typing rules necessitates that one can encode computations in a library. C++ templates are a Turing-complete language [43] allowing arbitrary computations on types and constants to be performed at compile time. Such *template metaprograms* [42] have found uses in various C++ libraries (see e.g. Boost.type_traits [31], and numerous other libraries in the Boost library collection [6]). Template metaprogramming, however, remained a relatively ad-hoc activity until the introduction of the Boost Metaprogramming Library (MPL) [1,21]. MPL provides a solid foundation for metaprogramming in C++, defining essentially a little programming language and a supporting library for defining *metafunctions*; in MPL metafunctions are functions from types to types. MPL allows one to define higher-order metafunctions, lambda metafunctions, etc., and provides a host of data structures and algorithms for storing and manipulating types.

For complex typing rules, a framework like MPL is essential; we use MPL extensively to define relations between types, in particular in the user-defined subtyping relation. For example, the following application of the is_subtype metafunction determines whether the types old_contact and new_contact shown above are in a subtyping relation:

```
is_subtype<old_contact, new_contact>::type;
```

Following the conventions of MPL, the result of the is_subtype metafunction is not a Boolean constant but rather a type, either mpl::true_ or mpl::false_.

2.2. Constraining and specializing functions based on arbitrary properties of types

Type systems typically define in which context the use of objects of certain types is allowed, what operators between objects of different types are allowed, and so forth. With metafunctions, it is possible to define arbitrary sets of types and relations between types. The ability to *enable* or *disable* functions based on conditions defined by arbitrary metafunctions then allows one to define the contexts where the specified sets of types are valid. This ability is offered with the enable_if templates [26,27]. We use these templates to enable certain operations, such as assignment, only when its parameters are in a subtyping relation. For example, the following assignment operation is defined only if the right-hand side of the assignment is an *arithmetic* type:

```
class A {
  ...
  template <typename T>
  typename enable_if<is_arithmetic<T>, A&>::type
  operator=(const T&);
};
```

The first argument to enable_if is a condition, a metafunction that has to evaluate to true for the assignment operator to be considered as a candidate for overload resolution; the is_arithmetic metafunction is defined in [31] and also in the current draft specification of the C++ Standard Library [25]. The second argument is the type of the entire enable_if<...>::type type expression in the case where the condition is true. Thus, in the definition above, the return type of the assignment operation is A&.

In addition to function overloads, the enable_if templates can be applied to enable and disable class template specializations based on arbitrary conditions.

2.3. Access to structure of types

To be able to define typing rules and conversion operators based on structural properties of types, a representation of the structure of types must be accessible to template metaprograms. The **class** construct of C++ is not useful in this regard—apart from modest (and inadequate for our purposes) compile-time reflection obtained by clever uses of templates, C++ offers no language support for inspecting the structure of classes at compile time. Instead of classes, we thus use the *tuple* types from the Boost Fusion Library [12]. When types are represented as nested instantiations of tuples, their structure becomes accessible to metaprograms; we can inspect and manipulate tuples with Boost Fusion's algorithms at compile time.

Fusion draws its design from that of MPL, but where, say, an MPL vector only contains types, a Fusion tuple contains types and values. Similar to MPL, we can define metafunctions in Fusion, but Fusion's metafunctions can also have a run-time component: Fusion's metafunctions map types to types and also values to values.

As a simple demonstration of the functionality offered by the Fusion library, below we first create a tuple type, populate its elements with values, define a function object that prints out its argument, filter out all non-arithmetic types from the tuple, and then print out the values that remain:

```
typedef fusion::tuple<std::string, int, char> grading_record;
grading_record rec = fusion::make_tuple("Humpty Dumpty", 89, 'B');

struct print {
  template <typename T> void operator()(const T& x) const { std::cout << x; }
}

fusion::for_each(filter_if<is_arithmetic<_>>(rec), print());
```

Both MPL metafunctions (such as is_arithmetic) and "traditional" function objects (such as print()) can be given as inputs to Fusion algorithms. Some Fusion algorithms, for example transform, require a *hybrid* of a metafunction class and a function object. This algorithm transforms a tuple to another tuple, potentially transforming both the types and the values of the elements. We use Fusion tuples to represent XML types and Fusion algorithms in defining implicit conversions between XML types.

2.4. Variants and other powerful abstractions

During the past few years, several new C++ Libraries that implement new "language constructs" have been introduced. The Boost Variant [18] and the Boost Optional [10] libraries, both of which provide notable new functionality to C++, are very useful in expressing complex types. For example, in the code below, Contact is a discriminated union type that can hold an object of any of the three types Email, Tel, or ICQ; MiddleName is a type that possibly contains a string:

```
typedef boost::variant<Email, Tel, ICQ> Contact;
typedef boost::optional<std::string> MiddleName;
```

Such *alternation* and *optionality* are central in XML typing.

We point out that Boost MPL, Fusion, Variant, and Optional, even enable_if, have all been developed using the *generic programming* methodology (see e.g. [33,20]), building their interfaces to a large extent against common *concepts* (in the technical sense of Stepanov, as established with the Standard Template Library [40]). As a result, the above libraries are highly interoperable. For example, the list of the element types of a variant type can be viewed as an MPL sequence, and thus manipulated either with the MPL or Fusion algorithms; enable_if expects MPL metafunctions as its condition argument, and so forth. Though mere libraries, the above set extends the C++ language in a significant way.

Though we have not attempted to implement XTL in any other language besides C++, we note that, e.g., Haskell supports the key capabilities that we identified as necessary for implementing type refinements. Template Haskell [38] offers powerful metaprogramming capabilities, as well as access to the representations of data types. The type class system of Haskell supports non-intrusive classification of types, and also a form of metaprogramming; for example, the approach of *strongly typed heterogeneous collections* [30] is very similar to that of Boost Fusion.

3. XTL: an eXtensible Typing Library

In this section, we demonstrate how the library techniques from the previous section enable extending the C++ type system. We present the rationale and design of the library components that support this task. We refer to these components and the accompanying conventions collectively as the *eXtensible Typing Library* (XTL). The core of XTL is very small, consisting of only a handful of template definitions, providing hooks for extension. A particular domain-specific type system refinement is achieved by extending the core according to XTL's conventions, which provide a uniform interface for each refinement, and interoperability between them.

We start with a simple example, presented in [11], that extends the integer type with qualifiers pos and neg to track statically when a value is positive or negative. A straightforward wrapping of a type with a template provides a simple but incomplete solution, shown in Fig. 1. The constructors and the assignment and conversion operators are supposed to capture the relationship of the new type pos<T> and the original type T. The technique of capturing such a relationship between types can be traced back to the early days of C++ [41, Section 6.3.2]. Objects of the underlying numeric type (T) can be used to initialize objects of pos<T> and neg<T> types. This is an unsafe operation and thus equipped with a run-time assertion. Conversions back to the underlying numeric type are safe, and provided with the user-defined conversion operators to T. How the pos and neg qualifiers behave with various operators is encoded by overloading those operators; here we show the overloads of **operator+**.

This straightforward solution is fairly limited. When using solely the types pos<T> and neg<T>, the behavior is well-defined, but the interaction of these types with other types, either built-in or user-defined, or with other possible qualifiers, is not. We can identify several questions, the answers to which are not clear in this simple approach. What is the relationship between the element type T and type pos<T>? The provided constructor and conversion operator make them convertible to

```
template <typename T> class pos {
  T m_t;
public:
  explicit pos(const T& t) : m_t(t)  { assert(t > 0); }
  operator T() const { return m_t; }
};

template <typename T> class neg;

template <typename T> pos<T> operator+(const pos<T>& a, const pos<T>& b);
template <typename T> T      operator+(const pos<T>& a, const neg<T>& b);
template <typename T> T      operator+(const neg<T>& a, const pos<T>& b);
template <typename T> neg<T> operator+(const neg<T>& a, const neg<T>& b);
```

Fig. 1. A straightforward implementation of type qualifiers pos and neg. We only show the definition of the class pos and the definition of **operator**+; class neg and other operators are defined analogously.

one another, but does this conversion lose any semantic information? Can values of one type always be implicitly converted to and used in place of the other? Are these types in a subtyping relation? How about the relationship of instantiations of pos and neg with different element types? What should, e.g., be the relationship between pos<**int**> and pos<**double**>? The straightforward approach is lacking in many respects.

3.1. XTL subtyping

The central notion in XTL is a user-definable subtyping relation (not based on inheritance). XTL sets the policies of how the subtyping relation is extended for new user-defined types, and provides the general building blocks to make the task effortless. In particular, when a user defines a type to be a subtype of another type, the rest of the framework assures that objects of the first type can be used in contexts where objects of the second type are expected. Note that even though we use the term *subtyping*, a conversion may in some cases be involved, e.g., in the case of XML types, described in Section 4.2. As part of defining the XTL subtyping relation for data types of a domain, the programmer defines the necessary conversion operators as well. In practice, when an object of a subtype is used in the context where supertype is expected, a user-defined conversion is often implicitly performed.

XTL's user-extensible subtyping relation is defined by the is_subtype metafunction: if the metafunction invocation is_subtype<S, T>::type evaluates to mpl::true_, XTL considers the type S to be a subtype of type T. The metafunction's implementation consists of a primary template and a set of template specializations. The primary template is defined as follows:

```
template <class T, class U, class Cond=void> struct is_subtype : is_same<T, U> {};
```

The is_same metafunction, defined in the (draft) standard library [25, Section 20.5.5], compares types for equality; XTL's subtyping relation is thus reflexive. The third template parameter Cond is a hook that allows (with the help of the enable_if template, see Section 2.2) one to attach an arbitrary type predicate to a template specialization, to determine when the specialization is enabled.

New pairs of types are added to the subtyping relation by partially or explicitly specializing the is_subtype template. To refine a type system for a particular domain, it generally suffices to specialize is_subtype for the types specific to that domain. Once these basic relations have been established, XTL provides an elaborate set of ready to use subtyping algorithms for compound types, including support for subtyping of function types (see Section 3.2), standard container types, type sequences and discriminated unions (see Section 4.1), and types refined with type qualifiers (see Section 3.4) similar to those described in [16].

The XTL subtyping relation is fully under the control of the programmer. The classes and types involved do not have to be altered when extending the is_subtype metafunction. For example, if deemed useful, the C++ type **char** can be defined to be a subtype of **int**, **int** a subtype of **double**, **double** a subtype of complex<**double**>, and so forth. In fact, such safe conversions are commonly useful, so they are available through inclusion of a dedicated XTL header file. The following assignments demonstrate the effect of these rules:

```
neg<int> ni(−20);
neg<double> nd = ni;    // OK
ni = nd;                // error
```

3.2. Subtype casting

Since physical representations of values in different types may vary, our definition of subtyping relation implies existence of a unified conversion mechanism capable of transforming a physical representation of a subtype into a physical

representation of a supertype. Such a conversion in XTL is accomplished with the subtype_cast function template, invoked as subtype_cast<T>(val), where T represents a supertype of val's type. This function deduces the type of val and, if it is a subtype of T, converts val to an object of type T. Otherwise the subtype_cast function template is disabled with the enable_if mechanism, and a call to it results in a compile-time error.

We demonstrate the use of the XTL subtyping relation and subtype_cast with subtyping of function types, instances of the std::function template from the (draft) standard library [25, Section 20.7.16.2]. Consider the following types A and B that are in a subtyping relationship B <: A; the tainted type qualifier is explained in Section 3.4:

```
typedef tainted<int> A;
typedef     pos<int> B;
```

We further define two function types B_to_A and A_to_B. The function types are instances of the std::function template, which is the standard library's generic wrapper for different kinds of function types, giving a uniform interface to function pointers, pointers to member functions, and function objects.

```
typedef std::function<A(B)> B_to_A; // function type B → A
typedef std::function<B(A)> A_to_B; // function type A → B
```

According to the usual subtyping rules between function types (covariant on return types, contravariant on parameter types), a function type A → B is a subtype of B → A, and we thus would like to be able to use functions of type A_to_B everywhere where functions of type B_to_A are expected, but not vice versa. For example:

```
A f(B);
B g(A);
B_to_A b2a = &f;  // Wrap f into std::function
A_to_B a2b = &g;  // Wrap g into std::function
b2a = a2b;        // OK, but rejected!
a2b = b2a;        // Error
```

The first assignment above, even though safe, is rejected. If implicit conversions apply between corresponding argument types and return types of two functions, std::function defines an implicit conversion between those two function types. This is of no help, since there is no implicit conversion from B to A. XTL, however, recognizes the subtyping relation between A_to_B and B_to_A, and with a subtype_cast the safe assignment is accepted:

```
b2a = subtype_cast<B_to_A>(a2b); // OK
a2b = subtype_cast<A_to_B>(b2a); // Error
```

XTL derives the subtyping fact A_to_B <: B_to_A from a general subtyping rule for function types, and the rules for type qualifiers that establish the fact pos<int> <: tainted<int>. All these facts are expressed by extending the is_subtype metafunction, and all conversions are performed using subtype_cast. As a consequence, types recognized by the XTL compose. For example, instead of types A and B above, the parameter and return types of the above functions could be other function types, or any other types recognized by the XTL, and the framework would recursively check whether the appropriate subtyping relations between those types hold. The framework thus allows extension with many domain-specific types independently, resulting in the expected behavior when the types are used together.

Before we proceed to revising the example in Fig. 1, we note that explicit casting is not very elegant. In practice, we can often avoid it by providing implicit conversions, either as constructors in the supertype or conversion operators in the subtype, that perform the call to subtype_cast. Such implicit conversions are not, however, possible when both types involved in the conversion are types that the developer of a type system cannot alter, such as built-in or standard types. In generic code, it is advisable not to rely on implicit conversions between types encoded as part of the XTL framework, but rather use subtype_cast explicitly if conversions are necessary. This will ensure that the code works with all applicable types. We follow this rule consistently in XTL.

3.3. Revisiting pos and neg

To demonstrate the use of the XTL's subtyping relation, we rewrite our naïve implementation of the pos and neg class templates, extend the is_subtype metafunction appropriately, and define the subtype casts. The pos class template is shown in Fig. 2. The definition of neg is similar.

We can observe that there is a new constructor in the pos class. Though taking two arguments, the second one has a default value, and thus the constructor implements an implicit conversion. The first argument seemingly matches any type, but in reality, only types that are subtypes of pos<T> will be considered. This is made possible by the second parameter that acts as a guard: the constructor is only enabled if U is defined to be a subtype of pos<T> by the is_subtype metafunction. The rather complex type expression boils down to the type **void**∗ when the function is enabled, thus the parameter can accept the default value 0. This is an idiomatic use of the enable_if template when placing a constraint to a constructor. The body of the constructor performs a conversion between the representations using the subtype_cast function.

```
template <typename T>
class pos {
  T m_t;
public:
  explicit pos(const T& t) : m_t(t) { assert(t > 0); }

  template <typename U>
  pos(const U& u, typename enable_if<is_subtype<U, pos<T> >, void>::type* = 0)
    : m_t(subtype_cast<T>(u)) {}

  template <typename U>
  typename enable_if<is_subtype<U, pos<T> >, pos&>::type
  operator=(const U& u) {
    m_t = subtype_cast<T>(u);
    return *this;
  }

  operator T() const { return subtype_cast<T>(*this); }
};
```

Fig. 2. Revisited definition of the pos class template.

The assignment operation has the same guard as the converting constructor described above. The condition is now, however, expressed as part of the return type of the operator. Again, this is idiomatic use of enable_if. The effect is that an object of type U can be assigned to a variable of type pos<T> exactly when U is a subtype of pos<T>.

The subtyping rules for pos are defined outside the pos class, by specializing the is_subtype metafunction:

```
template <typename S>
struct is_subtype<pos<S>, S> : mpl::true_ {};

template <typename S, typename T>
struct is_subtype<pos<S>, pos<T> > : is_subtype<S, T> {};
```

Here, the first specialization states that a pos type is a subtype of its element type, and the second that two pos types are in a subtyping relation when their element types are.

Calls to the subtype_cast function express the target type as an explicitly specified template parameter. This function is disabled when the source type is not a subtype of the target type, but otherwise it performs the cast by delegating the task to the subtype_cast_impl function. The target type is carried in the type of the first argument to subtype_cast_impl. This arrangement makes extending XTL with new types easier. First, the disabling condition in subtype_cast remains the same for all types, and does not need to be repeated for each extension. Second, some extensions require partially specializing the target type. We can rely on the function overloading mechanism for this with the subtype_cast_impl functions, where the target type is deducible. In contrast, the target type template parameter in subtype_cast is explicitly specified; partially specializing such parameters is not supported in C++.

The subtype_cast_impl function is thus the function overloaded when extending XTL with new types. According to the above subtyping rules, we overload subtype_cast_impl to specify how to convert between a subtype and a supertype in the case of pos-qualified types:

```
template <typename T>
T subtype_cast_impl(target<T>, const pos<T>& p) { return p.m_t; }

template <typename T, typename S>
pos<T> subtype_cast_impl(target<pos<T> >, const pos<S>& p) {
  return pos<T>(subtype_cast<T>(p.m_t));
}
```

Definitions of operators now also change slightly to take subtyping into account:

```
template <typename T, typename U>
pos<typename add_op_result_type<T, U>::type>
operator+(const pos<T>& a, const pos<U>& b) {
  typedef typename add_op_result_type<T, U>::type result_type;
  return pos<result_type>(a.m_t + b.m_t);
}
```

The metafunction add_op_result_type<T, U> computes the resulting type of the addition operation between objects of types T and U. We omit the definition of add_op_result_type; in the forthcoming revision of standard C++ [25], this metafunction invocation can be replaced with the built-in decltype operator.

3.4. Type qualifiers

The pos and neg qualifiers presented above are a simple example of an important direction for enriching type systems: refining a type with qualifiers. Type qualifiers modify existing types to capture additional semantic properties of the values flowing through the program. A well-known example of a type qualifier is the **const** qualifier of C++, used for tracking immutability of values at different program points. Other examples include type qualifiers for distinguishing between user and kernel level pointers [28], safe handling of format strings [37], and tracking of values with certain mathematical properties [11].

Instead of implementing different type qualifiers to type systems in an ad-hoc manner, several systems, based on a general theory of type qualifiers, have been described [16,17,15]. These systems allow an economical definition of behavior of domain-specific qualifiers.

In this section, we review common properties of type qualifiers, and show how to implement type qualifiers as a C++ template library using the XTL framework. To give a brief example, we use *taintedness* analysis [37] that uses the qualifiers untainted and tainted to tag data coming from trustworthy and potentially untrustworthy sources, respectively. The requirement is that tainted data may never flow where untainted data is expected. We may want to ensure that, say, data originating from measurements, considered as trustworthy (untainted) data, is not mixed with tainted data from untrustworthy sources (e.g. assumptions, values obtained from modeling, etc.) to produce untrustworthy results. Note that type qualifiers can be composed—besides trustworthiness, values may have other properties we want to track: positiveness, constness, measurement units, etc. The following pseudo-code involves multiple type qualifiers applied to the same type:

```
extern untainted kg double get_weight();
const kg double a = get_weight(); // OK, untainted dropped
kg untainted double b = a;        // Error, no untainted in the right—hand side
b = get_weight();                 // OK, qualifiers are preserved
```

We discuss later in this section how to verify type safety of an assignment that involves multiple type qualifiers; here we just note that the order of application of type qualifiers to a type should not matter—it does not in our framework—and types that differ only in the order of qualifiers should be semantically equivalent. In what follows, by *qualified type* we mean a type that is obtained through applying one or more type qualifiers to an *unqualified type*.

As with pos and neg, we represent a type qualifier as a template class with a single parameter that represents the type being qualified. By taking advantage of the common properties of all type qualifiers, we can reduce the work that is necessary for defining a new qualifier. The developer of a type qualifier explicitly marks his template class as a type qualifier through specialization of a traits-like class is_qualifier. It is not necessary to provide new specializations for the is_subtype metafunction or overloads for the subtype_cast_impl functions. The behavior follows according to whether the qualifier is *positive* or *negative* [16], which the programmer states in the definition of the qualifier class. An example definition is shown in Fig. 3.

Definition 1.
A type qualifier q is *positive* if T <: q T for all types T for which q T is defined.
A type qualifier q is *negative* if q T <: T for all types T for which q T is defined.

The C++ qualifier **const**, type qualifier tainted [37], and optional [10] are examples of positive type qualifiers because T <: **const** T, T <: tainted<T>, and T <: optional<T>, respectively. Qualifiers pos, nonzero, and untainted are examples of negative type qualifiers because pos<T> <: T, nonzero<T> <: T, and untainted<T> <: T.

As mentioned above, the order of applying type qualifiers to a type should not affect the resulting type's behavior. Thus, definitions of operations on type qualifiers must be made ignorant of the particular order of qualifiers. Directly overloading an operator for a particular qualifier, as in the addition operator in Section 3.3, is insufficient. We note that type qualifiers do not change the underlying operation, only the type of the result. For example, when we apply the pos qualifier to type **int** we still use the addition operation defined on **int**s, but ask pos to be applied to the result type whenever both argument types are qualified with pos. Consequently, XTL defines generic operations that match all qualified types and ignore the order of qualifiers. Using metafunctions, these operators can then be customized with the typing rules for particular qualifiers. For example, the rules for how the untainted and tainted type qualifiers are propagated in the addition operation are as follows:

```
untainted + untainted → untainted
untainted + tainted   → tainted
tainted   + untainted → tainted
tainted   + tainted   → tainted
```

The encoding of typing rules for the addition operator is done by specializing XTL's add_op template:

```
template <template<typename> typename A, template<typename> typename B>
struct add_op { typedef mpl::identity<mpl::_1> type; };
```

```
template <typename T>
struct untainted : negative_qualifier<typename unqualified_type<T>::type> {
  typedef negative_qualifier<typename unqualified_type<T>::type> base;

  untainted() : base() {}
  explicit untainted(const typename base::unqualified_type& t) : base(t) {}

  template <typename U>
  explicit untainted(
    const U& u, typename enable_if<is_subtype<U, untainted<T> >, void>::type* = 0)
      : base(subtype_cast<T>(u)) {}

  template <typename U>
  typename enable_if<is_subtype<U, untainted<T> >, untainted&>::type
  operator=(const U& u) {
    base::operator=(subtype_cast<T>(u));
    return *this;
  }

  operator T() const { return subtype_cast<T>(m_t); }

};
```

Fig. 3. The definition of the untainted type qualifier class using the XTL framework. The negative_qualifier class is defined in XTL, as well as the subtyping rules common to all negative qualifiers. To access the underlying unqualified type, XTL provides the unqualified_type metafunction.

For the tainted and untainted rules above, the specializations are as follows:

```
template <> struct add_op<untainted, untainted> { typedef add_qualifier<untainted> type; };
template <> struct add_op<untainted,  tainted>  { typedef add_qualifier<tainted>   type; };
template <> struct add_op<tainted,  untainted>  { typedef add_qualifier<tainted>   type; };
template <> struct add_op<tainted,   tainted>   { typedef add_qualifier<tainted>   type; };
```

The class template add_op takes two qualifier templates as template template parameters, and defines a metafunction that will be applied to compute what qualifiers should be present in the result type of **operator+**. The primary template sets the default to mpl::identity: qualifiers are neither added nor removed from the result type. The metafunction add_qualifier, defined by XTL, applies a given qualifier template to the result type. XTL's generic implementation of a particular operation will loop through all possible combinations of qualifiers in the arguments' types and apply the corresponding metafunctions to compute the qualifiers that should be applied to the result type.

To arrange that a particular operator is not dependent of the order of qualifiers in its argument types (for example, untainted<nonzero<optional<**int**>>> should have the same overloading behavior as optional<untainted<nonzero<**int**>>>), the XTL uses enable_if in the overloaded operators for qualified types. Focusing a particular operator, say, **operator+**, no separate overloads for optional<T>, untainted<T>, etc., are provided. Instead, a single overload matches any composition of qualifiers. This is arranged by guarding the overload with enable_if and the metafunction is_qualifier_type<T>. Similarly, the is_subtype metafunction, that determines when an object of one qualified type can be assigned to that of another type, is agnostic of the exact order of the qualifiers. This metafunction inspects the set of type qualifiers, and bases the subtyping relation on the negativeness or positiveness of the qualifiers. Omitting some details, to preserve subtyping, a positive type qualifier can only be added to the right-hand side, and a negative type qualifier can only be removed from the left-hand side. For example: nonzero<optional<untainted<T>>> is a subtype of tainted<optional<nonzero<U>>> whenever T <: U. Here the negative type qualifier untainted was dropped from the left while the positive type qualifier tainted was added to the right. Other qualifiers were preserved. Dropping the optional qualifier in the right-hand side would have made the subtyping fail.

To give a feel of working with type qualifiers built with XTL, Fig. 4 shows code using some of the type qualifiers mentioned above.

Even with the subtyping and casting functionality provided by the XTL, the definition of an individual type qualifier class is still fairly elaborate, but mostly boilerplate code. For cases where no special run-time checks are needed, the XTL provides two macros for taking care of this boilerplate. For example, the two macro invocations below generate the type qualifier classes for the tainted and untainted qualifiers:

```
DECLARE_POSITIVE_QUALIFIER(tainted);
DECLARE_NEGATIVE_QUALIFIER(untainted);
```

The metafunctions add_op, sub_op etc. that describe how different operations carry the qualifiers must still be defined.

```
struct SomeType {};

void foo(pos<int>)          { /*...*/}
void foo(pos<SomeType>) { /*...*/}

int main()
{
  nonzero<neg<int> >              a(-44);
  untainted<pos<nonzero<int> > > b(2);

  neg<nonzero<long> >     m = a * b; // OK
  //nonzero<pos<double> > e = b - a; // Error: difference of two non-zeros can be zero
  pos<double>             d = b - a; // OK: no nonzero

  foo(b);   // OK, picks the right overload
  //foo(d); // Error: double is not a subtype of int

  nonzero<tainted<double> > bc = subtype_cast<nonzero<tainted<double> > >(b);
  nonzero<tainted<double> > bi = b; // same as above
  bi = b;

  pos<nonzero<int> > c(3);
  std::string             ci = subtype_cast<std::string>(c); // user defined that int <: std::string
  tainted<std::string>  cc = subtype_cast<std::string>(c); // OK to add a positive qualifier to the left
}
```

Fig. 4. Example of working with XTL's qualifiers, as well as demonstration of how subtyping relation can be established non-intrusively between types. Above, the programmer has defined **int** to be a subtype of the std::string type.

4. Typing XML in C++

In this section, we describe how the XTL, with the help from several C++ template libraries, allows an elaborate extension to the C++'s type system: static typing of XML.

4.1. Background: regular expression types

Type systems that understand XML data have gained considerable interest. The central idea is to harness the type system to guarantee statically that a particular program cannot manipulate or produce XML documents that do not conform to a particular DTD [45] or schema [35]. The insight is that XML data corresponds directly to *regular expression types*, which then can be given a representation in the type systems of various languages. Some of the recent efforts in this direction include the XDuce language [22], specifically designed for XML processing, that has a direct representation for regular expression types; the C_ω [5] and Xtatic [19] languages that extend C# with regular expression types; and the HaXml [44] toolset, that uses Haskell's algebraic data types to represent XML data.

Regular expression types, e.g., as defined in XDuce, are sets of sequences over certain domains. Values from those domains denote singleton and composite sequences. Composite sequences are formed with the regular expression operators "," (concatenation), "|" (alternation), "*" (repetition), and "?" (optionality), together with type constructors of the form "$l[\cdots]$". If S and T are types, then S, T denotes all the sequences formed by concatenating a sequence from S and a sequence from T. $S|T$ denotes a type that is a union of sequences from S and sequences from T. Type $l[T]$, where T is a type and l ranges over a set of labels, defines a set of labeled sequences where each sequence from T becomes classified with the label l. Type $T*$ denotes a set of sequences obtained by concatenating a finite number of elements from T. The empty sequence is denoted with (), and $T?$ denotes any sequence from T or an empty sequence.

Consider for example the following XML snippet describing a contact:

```
<contact>
  <name>Humpty Dumpty</name>
  <tel>555-4321</tel>
  <email>humpty.dumpty@tamu.edu</email>
</contact>
```

This snippet conforms to the XML schema in Fig. 5. Labels classify types similarly to how XML tags classify their content; For example, the type email[...] corresponds to <email>...</email> in XML parlance. Using XDuce syntax, the regular expression type

```
contact[name[string], tel[string]*, email[string]*]
```

corresponds to the schema in Fig. 5; both define the set of XML snippets with "contact" as the root element containing single "name" element, followed by zero or more "tel", followed by zero or more "email" elements.

```
<?xml version="1.0" encoding="ISO−8859−1" ?>
<xsd:schema xmlns:xsd ="http://www.w3.org/2001/XMLSchema">
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="tel" type="xsd:string"/>
<xsd:element name="email" type="xsd:string"/>
<xsd:element name="contact">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="name"/>
      <xsd:element ref="tel" maxOccurs="unbounded"/>
      <xsd:element ref="email" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Fig. 5. An example XML schema.

An interesting feature of the XDuce language is the *semantic subtyping* relation between regular expression types, defined as the subset relation between languages generated by two tree automata [22]. For example, the following subtyping relationships hold:

```
T∗, U∗ <: (T | U)∗
T, (T)∗ <: T∗
```

Subtyping between two regular expression types corresponds to safe convertibility between XML fragments. Subtyping between XML fragments is useful, for example, in providing backward compatibility of documents that correspond to an older schema: code written against a newer schema should work for older schemas, as long as the type defined by the newer schema is a supertype of the type defined by the older one.

The decision problem for subtyping between regular expression types is EXPTIME-hard [23,36] in the worst case, but the cases that lead to this worst case complexity are reported to be rarely seen in practice [2].

4.2. Regular expression types in C++

We define an encoding of regular expression types in C++. Regular expression types are represented as nested template instantiations, consisting of sequence types, variants, and lists. We represent XML elements in our system as a struct parameterized with two types, the first of which represents the element's tag and second the element's data:

```
template <typename Tag, typename T = detail::empty> struct element { T data; };
```

The Tag type denotes the name of the XML element, or the label in XDuce's regular expression types. Empty XML elements can be represented by an element instantiated with nothing but a tag type. Complex XML elements may have several levels of instantiations of element as their data type. Consider for example the following XML snippet:

```
<contact>
  <name>Humpty Dumpty</name>
</contact>
```

It can be given the following type in our library:

```
struct contact { /*... */ };
struct name    { /*... */ };

typedef element<contact, element<name, string> > Contact;
```

Tag-classes are also used to keep additional information about the tag: the name as a character array, XML node type, additional restrictions etc. For example, the full definition of the contact tag is:

```
struct contact {
  static const char* tag_name() { return "contact"; }
  typedef attribute node_type;
};
```

Sequencing of XML elements is represented with the seq template. Here is an example of using sequencing of elements:

```
template <typename TTag, typename T = detail::empty>
class element {
  ...
  template <typename UTag, typename U>
  element(const element<UTag, U>&,
          typename enable_if<
                    is_subtype<element<UTag, U>, element<TTag, T> >
                  >::type* = 0) { ... }
  ...
};
```

Fig. 6. The converting constructor of the element class template. The enable_if guard allows the constructor to match exactly when the argument's type is a subtype of the class to be constructed, according to the library's type system.

```
typedef element<name,    string> Name;
typedef element<tel,     string> Tel;
typedef element<email,   string> Email;
typedef element<contact, seq<Name, Tel, Email> > Contact;
```

The empty sequence () is represented by seq<>.

The seq template is a simple wrapper around Fusion's tuple class [12]. We use the wrapper to change the behavior of certain operations, e.g., to perform preprocessing on the tuple types. For example, we define the I/O operators for seqs to perform a *flattening* of sequences prior to delegating the call to Fusion's tuple I/O. For example, the sequence (A, B, (C, D), E) is flattened to (A, B, C, D, E). Fusion tuples are MPL-compliant sequences [1,21], and in our subtyping algorithm we operate on tuples using MPL algorithms.

Alternation of XML elements is represented with the alt class template that is again just a simple wrapper, now around Boost's variant template [18]. In the case of alternation, the wrapping is done to allow the redefinition of the I/O routines. Here is a small example of using alternation:

```
typedef alt<Tel, Email> ContactInfo;
typedef element<contact, seq<Name, ContactInfo, ContactInfo> > AlternativeContact;
```

The empty union is represented by alt<>.

A repetition of XML elements is represented with the rep template, a wrapper over a std::vector of Fusion tuples. Using repetition, the contact definition from Section 4.1 can be expressed as follows:

```
typedef element<contact, seq<Name, rep<Tel>, rep<Email> > > FlexibleContact;
```

4.2.1. Subtyping relation

We utilize the static metaprogramming capabilities of C++ to establish a subtyping relation between two regular expression types. Again, the is_subtype metafunction is harnessed for this purpose. We note one restriction. XDuce allows the definition of recursive data types, and can decide subtyping between right/tail recursive data types (the decision problem of subtyping between general recursive data types is undecidable). An implementation of subtyping between recursively defined data types, analogous to that in XDuce, has so far eluded us. However, we support *repetition*, which is functionally equivalent, but possibly syntactically more cumbersome, to right/tail recursion (analogously to the equivalence between right-linear grammars and regular expressions). The restriction is that subtyping of regular expression types with repetition, is weaker than in XDuce—there are cases where two regular expression types are in the semantic subtyping relation, but the is_subtype metafunction does not agree.

The implementation of the is_subtype metafunction for XML types is lengthy and we do not show it here. It amounts to implementing the subtyping rules of XDuce (really a limited form of them per the restrictions mentioned above) using MPL. Once the is_subtype metafunction has been defined to recognize our XML types, we can exploit it to implement guards similar to those in the constructors of the qualified types—Fig. 6 demonstrates with the converting constructor of the element class template.

We do not overload subtype_cast_impl to define conversions on element types because we define such conversions on the element class itself. Calling subtype_cast on the element type then calls the most general implementation of subtype_cast_impl, which simply tries to apply either a standard or a user-defined conversion on the type, which is exactly what we need.

4.3. Working with the library

In addition to the core type system, we have implemented some supporting functionality as part of our XML processing library. This includes I/O and automatic generation of the C++ types from an XML schema. For I/O, we provide direct streaming operations. To automate generation of the C++ types from XML types, we provide an XSL transformation from an XML schema to C++ source code. To give a general feel for the use of our XML framework, Fig. 7 presents a complete example

```
using namespace xml;
using namespace std;

// ——— definitions generated by XSLT transformation —→
struct name { /*...*/}; struct email   { /*...*/};
struct tel    { /*...*/}; struct contact { /*...*/};

typedef element<name,  string>           Name;
typedef element<email,  string>          Email;
typedef element<tel,    alt<string, int> > Tel;
typedef element<contact, seq<Name, Tel, Email> > Contact;
typedef element<contact, seq<Name, rep<Tel>, rep<Email> > > ContactEx;
// ←— definitions generated by XSLT transformation ———

int main() {
 try {
  ifstream xml("contact.xml");
  Contact contact;
  xml >> contact;                    // Parse file
  ContactEx contact_ex = contact; // OK, implicit subtype_cast<ContactEx>(contact)
  cout << contact_ex;                // Output XML
  // contact = contact_ex;           // Error, not in subtyping relation
 }
 catch(invalid_input& x) {
  cerr << "Error parsing " << x.what();
  return —1;
 }
 return 0;
}
```

Fig. 7. An example of working with our XML library. Type Contact is a subtype of ContactEx, which is why the assignment contact_ex = contact is allowed while contact = contact_ex is not.

of turning an XML schema into the corresponding C++ encodings of XML types, and working with them. The comments in the code point out the parts of the code that were generated from an `.xsd` file (the XML schema description), where our support functionality (input and output) is invoked, and where the subtyping checks are performed.

To demonstrate the practicality of the library, as another, larger example, we generated XML types for two established Internet standards for syndication, the RSS [34] and Atom [3], and modeled a subtyping relation between documents of these standards, allowing thus a type safe conversion from one to another. The standards themselves do not provide a ready to use XML schema to validate the documents. Several slightly different schema exist; our XML types are generated from the schemas in [46].

To establish a subtyping relation between the types representing the two different syndication formats, call these types T_{RSS} and T_{Atom}, we defined a mapping, presented in Fig. 8, between tags from one schema (RSS) to another (Atom), making T_{RSS} a subtype of T_{Atom}. In many cases a tag in one schema had a natural "semantic" match in the other. For example an RSS tag *pubDate* that indicates when an item was published can be matched to Atom's tag *published* with the same meaning. Similarly, RSS's notion of *author* can be represented with Atom's broader notion of *contributor*. With "subtagging" relations defined, the subtyping relation between XML types is taken care of by the XTL. Both feeds accept unrecognized XML tags (for future extensibility) via an "extension element" any[any]. Elements in RSS that do not have a match in Atom are thus matched to a wildcard element any[any]. Our implementation of any_element is a supertype of all XML element types. Objects of the any_element type simply store the XML source code of an element cast to them, and thus do not lose information. For our experiment, we ignored some of the differences in representations of data stored as strings within certain XML elements. For example, we did not account for differences in the date formats. We also dropped attributes from XML elements, as our current implementation does not support them. Atom's feed element consists of a single repetition of a large alternation, which we unrolled to a sequence of three such repetitions (which is semantically equivalent), to match the structure of the RSS's channel element.

Fig. 8 shows the regular expression types representing the schema of the RSS and Atom feeds. We further zoom in to the types representing a single news story within a feed: item in RSS and entry in Atom. These are the most interesting types; the elements whose definitions we omit are much simpler.

4.4. Impact on compile times

Heavy use of template metaprogramming is known to increase compile times of C++ programs, often significantly. We conducted experiments to estimate the impact that library-defined type systems written using the XTL have on compile

RSS	Atom
item[([1]title[string]? | [2]description[string]? | [3]link[anyURI]? | [4]author[tEmailAddress]? | [5]category[tCategory]? | [6]guid[tGuid]? | [7]pubDate[tRfc822FormatDate]? | [8]source[tSource]? | [0]comments[anyURI]? | [0]enclosure[tEnclosure]? | [0]any[any]∗)+]	entry[([1]title[textType] | [2]content[contentType]? | [3]link[linkType]∗ | [4]author[personType]∗ | [5]category[categoryType]∗ | [6]id[idType] | [7]published[dateTimeType]? | [8]source[textType]? | contributor[personType]∗ | rights[textType]? | summary[textType]? | updated[dateTimeType] | issued[dateTimeType] | modified[dateTimeType] | [0]any[any]∗)∗]
channel[([1]title[string] | [2]link[anyURI] | [3]category[tCategory]? | [4]copyright[string]? | [5]managingEditor[tEmailAddress]? | [5]webMaster[tEmailAddress]? | [6]lastBuildDate[tRfc822FormatDate]? | [6]pubDate[tRfc822FormatDate]? | [7]generator[string]? | [8]image[tImage] | [0]description[string] | [0]language[language]? | [0]docs[anyURI]? | [0]cloud[tCloud]? | [0]ttl[nonNegativeInteger]? | [0]textInput[tTextInput]? | [0]skipHours[tSkipHoursList]? | [0]skipDays[tSkipDaysList]? | [0]any[any]∗)+, [9]item[tRssItem]+, [0]any[any]∗]	feed[([1]title[textType] | [2]link[linkType]∗ | [3]category[categoryType]∗ | [4]rights[textType]? | [5]contributor[personType]∗ | [6]updated[dateTimeType] | [7]generator[generatorType]? | [8]icon[iconType]? | [9]entry[entryType]∗ | author[personType]∗ | id[idType] | logo[logoType]? | subtitle[textType]? | [0]any[any]∗)+]

Fig. 8. RSS and Atom types for news item and feed. Tags with the same numbers in both columns represent our semantic matching between tags.

times. We tested both the uses of type qualifiers and the use of the XML framework. We used the GCC 3.4.4 compiler for all tests, on an Intel Pentium M processor running at 2 GHz with 512 MB of RAM.

Table 1
Compilation times, in seconds, of the type qualifiers test programs.

N	0	1	2	3	4	5	6	7	8	9	10
Time	1.12	1.72	2.61	2.81	3.18	3.94	4.97	5.55	6.28	15.40	19.22

Table 2
Compilation times, in seconds, of the test programs for the XML type system. The compilation time of an "empty" program, containing the same include files but no template instantiations, was 1.31 s.

n/k	1	2	3	4	5	6	7	8	9
1	1.48	1.66	1.73	1.81	2.03	2.14	2.35	2.54	2.79
2	1.52	1.77	2.29	3.67	8.28	27.38			
3	2.43	2.00	2.66	6.00	31.43				
4	2.62	2.03	3.63	25.57					
5	2.32	2.28	7.15						
6	2.73	4.13	19.30						
7	2.48	2.86	56.78						
8	1.74	3.65							
9	1.76	4.93							

Our test-suite for type qualifiers consisted of 11 versions of the same program, each working with types qualified with a different number of qualifiers: the program with index $n \in \{0, \ldots, 10\}$ used types qualified with n qualifiers. Each program defined 20 pairs of qualified types where first type of the pair was always made to be a subtype of the second type of the pair. Pairs used various combinations of positive and negative qualifiers that preserved the subtyping relation. For all type pairs (A, B), the compiler was forced to verify subtyping as is_subtype<A, B>::type::value;. Additionally, each program contained 20 functions, each of which instantiated values of two types from the pair and then performed a multiplication of them to trigger inference of qualifiers from an operation. Each applied qualifier had up to 4 rules defined on it. The full test suit can be obtained from XTL's website [39]. Compilation times (in seconds) for these tests are given in Table 1. The compile time should be compared against the value in row 0, which defines the baseline: the equivalent program without any qualifiers. Our current implementation of the subtyping algorithm for qualifiers has (compile-time) complexity $O(n^2)$ where n is the number of qualifiers applied. Though there is noticeable increase in compilation times of the last two cases, for a reasonable number of qualifiers slowdowns are moderate—ten different qualifiers applied to the same type seems unlikely. Note also that the qualifier use in the test programs is proportionally very high—the programs contain practically no other code than code that triggers the is_subtype test with different inputs. Though not supported in XTL, a "release mode" that sidesteps subtype tests for faster compile times would be possible for the type qualifier library. In particular, with *template aliases*, a qualified type, such as pos<T> for any T, could be defined to be a type alias for the unqualified type T. Template aliases are a forthcoming feature of the next revision of standard C++ [13,25, Section 14.5.7].

The subtyping relation of the XML types is computationally more expensive than that of type qualifiers. As mentioned earlier, in the general case deciding subtyping of regular expression types is EXPTIME-hard. The computationally expensive cases are subtyping relations of the following form:

$$l(A_1, \ldots, A_n) <: l(B_{11}, \ldots, B_{1n}) | \cdots | l(B_{k1}, \ldots, B_{kn}).$$

Verification of such a relation in the general case involves a number of steps that is proportional to the number of ways a k-element set can be split into n disjoint sets. Detailed discussion can be found in [23].

We wanted to test the effect of this worst-case scenario on compile times. Our test suite for the XML type system consisted of 81 tests—one for each combination of n and k (ranging from 1 to 9) from the above relation. In each of the tests, we invoked the is_subtype metafunction using XML types that trigger the exponential case with (essentially) the following code:

```
is_subtype<
  element<a, seq<Ak0, . . . , Akn> >,
  alt<element<a, seq<A00, . . . , A0n> >,
      element<a, seq<A10, . . . , A1n> >,
      . . . ,
      element<a, seq<Ak0, . . . , Akn> >
  >
>::type::value
```

Table 2 represents compilation times in seconds for different values of n and k. Empty entries correspond to tests that did not finish within 10 min or exceeded the compiler's limit on the number of nested template instantiations (500). With small n and k, effect to compile times is small. With larger values, they become infeasible as expected. Other implementations of the

subtyping algorithm have been reported to behave satisfactorily on practical examples [23], suggesting that the exponential case with large n and k does not manifest often in practice.

Summarizing the test results, using our approach to refining type systems can have a notable negative impact on compile times. In the case of the type qualifiers, the slowdown is quite reasonable, and typical for libraries relying on template metaprogramming. The XML case behaves similarly, except that the pathological cases that lead to exponential growth in the cost of deciding subtyping also obviously increase the compilation times exponentially.

5. Conclusions and future work

Type systems are traditionally closely tied to the implementation of a compiler or an interpreter, and typically are not extensible. We present a library solution for extending the type system of a general-purpose programming language with typing of domain-specific abstractions. This is a very economical and lightweight approach to building type systems. The presented solution does not require any compiler support and can be fully implemented in standard C++ [24]. We demonstrated that it is feasible to implement elaborate typing behavior purely as a library. We used our framework to build two extensions to the C++ type system: type qualifiers and regular expression types. The library of type qualifiers allows effortless definition of new type qualifiers and their order-independent composition. Regular expression types can directly describe the structure of XML documents. A subtyping relation between regular expression types can express safe conversions between different structures of XML data. For example, with our library we can write programs that are statically guaranteed to produce only XML data that conforms to a particular schema. For more convenient use of the library, we additionally provide machinery to map XML schema definitions to corresponding library abstractions.

In the future, we plan to explore the limits of the approach, implementing different kinds of type system extensions in terms of the XTL tools we described. We are currently looking into extending XTL with the possibility to define co- and contravariancy of template arguments on multi-argument class templates. If proven successful, the current implementation of the type qualifiers will become a partial case of this more general subtyping algorithm. Within the XML domain, we are currently working on support for attributes and exploring alternatives for providing support for pattern matching. In addition, currently we only support a basic subset of primitive XML data types. We plan to extend this support to other built-in types as well as possibly providing a compile- and run-time support of facets.

Acknowledgements

We are grateful to Esam Mlaih for contributing to XTL's XML I/O implementation.

References

[1] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, 2004.
[2] A. Aiken, B.R. Murphy, Implementing regular tree expressions, in: Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Springer-Verlag New York, Inc., New York, NY, USA, 1991.
[3] The atom syndication format, 2009. http://tools.ietf.org/html/rfc4287.
[4] J. Barton, L. Nackman, Scientific and Engineering C++, Addison-Wesley, 1994.
[5] G.M. Bierman, E. Meijer, W. Schulte, The essence of data access in Cω, in: A.P. Black (Ed.), ECOOP 2005—Object-Oriented Programming, 19th European Conference, in: Lecture Notes in Computer Science, vol. 3586, Springer, 2005.
[6] Boost, Boost C++ Libraries, Boost is a peer-reviewed collection of portable C++ libraries. http://www.boost.org/.
[7] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: preventing data races and deadlocks, in: OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2002.
[8] G. Bracha, Pluggable type systems, in: OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004. URL: http://pico.vub.ac.be/~wdmeuter/RDL04/papers/Bracha.pdf.
[9] W.E. Brown, Applied template metaprogramming in SIUNITS: the library of unit-based computation, in: Second Workshop on C++ Template Programming, 2001, in Conjunction with OOPSLA'01. http://www.oonumerics.org/tmpw01/brown.pdf.
[10] F. Cacciola, The Boost Optional Library. http://www.boost.org/libs/optional/.
[11] B. Chin, S. Markstrum, T. Millstein, Semantic type qualifiers, in: PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2005.
[12] J. de Guzman, D. Marsden, T. Schwinger, The Boost Fusion Library. http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html/index.html (Sep 2008). URL: http://www.boost.org/libs/fusion/.
[13] G. Dos Reis, B. Stroustrup, Templates aliases (revision 3), Tech. Rep. WG21/N2258=07-0118, JTC1/SC22/WG21 C++ Standards Committee, 2007. URL: http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2007/n2258.pdf.
[14] D. Evans, Static detection of dynamic memory errors, in: PLDI '96: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, Philadelphia, PA, USA, 1996.
[15] J.S. Foster, Type qualifiers: lightweight specifications to improve software quality, Ph.D. thesis, University of California, Berkeley, 2002.
[16] J.S. Foster, M. Fähndrich, A. Aiken, A theory of type qualifiers, in: PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1999.
[17] J.S. Foster, T. Terauchi, A. Aiken, Flow-sensitive type qualifiers, in: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2002.
[18] E. Friedman, I. Maman, The Boost Variant Library, 2002. http://www.boost.org/doc/html/variant.html.
[19] V. Gapeyev, M.Y. Levin, B.C. Pierce, A. Schmitt, The Xtatic experience, in: Workshop on Programming Language Technologies for XML (PLAN-X), 2005, University of Pennsylvania Technical Report MS-CIS-04-24, 2004.
[20] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, J. Willcock, A comparative study of language support for generic programming, in: OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2003.
[21] A. Gurtovoy, The Boost MPL library, July 2002. http://www.boost.org/libs/mpl/doc/index.html.

[22] H. Hosoya, B.C. Pierce, XDuce: a typed XML processing language (preliminary report), in: D. Suciu, G. Vossen (Eds.), International Workshop on the Web and Databases (WebDB), 2000, in: LNCS, vol. 1997, Springer, 2001. Reprinted in The Web and Databases, Selected Papers.
[23] H. Hosoya, J. Vouillon, B.C. Pierce, Regular expression types for XML, ACM Trans. Program. Lang. Syst. 27 (1) (2005) 46–90.
[24] International Organization for Standardization, ISO/IEC 14882:2003: Programming Languages: C++, 2nd ed., Geneva, Switzerland, 2003. URL: http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110.
[25] Working draft, standard for programming language C++, Tech. Rep. N2723=08-0233, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2008. URL: http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2008/n2723.pdf.
[26] J. Järvi, J. Willcock, H. Hinnant, A. Lumsdaine, Function overloading based on arbitrary properties of types, C/C++ Users Journal 21 (6) (2003) 25–32.
[27] J. Järvi, J. Willcock, A. Lumsdaine, The Boost Enable_if Library, Boost, 2003. http://www.boost.org/libs/utility/enable_if.html.
[28] R. Johnson, D. Wagner, Finding user/kernel pointer bugs with type inference, in: USENIX Security Symposium, 2004.
[29] A. Kennedy, Dimension types, in: Proceedings of the 5th European Symposium on Programming, ESOP, in: Lecture Notes in Computer Science, vol. 788, Springer-Verlag, 1994.
[30] O. Kiselyov, R. Lämmel, K. Schupke, Strongly typed heterogeneous collections, in: Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, ACM Press, New York, NY, USA, 2004.
[31] J. Maddock, S. Cleary, et al. The Boost type_traits library, 2002. www.boost.org/libs/type_traits.
[32] Y. Mandelbaum, D. Walker, R. Harper, An effective theory of type refinements, in: ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ACM Press, New York, NY, USA, 2003.
[33] D.A. Musser, A.A. Stepanov, Generic programming, in: Proceedings of International Symposium on Symbolic and Algebraic Computation, in: Lecture Notes in Computer Science, vol. 358, Rome, Italy, 1988.
[34] RSS 2.0 specification, 2009. http://www.rssboard.org/rss-specification.
[35] XML Schema, 2005. http://www.w3.org/XML/Schema.
[36] H. Seidl, Deciding equivalence of finite tree automata, SIAM J. Comput. 19 (3) (1990) 424–437.
[37] U. Shankar, K. Talwar, J. Foster, D. Wagner, Detecting format string vulnerabilities with type qualifiers, in: Proceedings of the 10th USENIX Security Symposium, 2001. URL: http://portal.acm.org/citation.cfm?id=1267628.
[38] T. Sheard, S. Peyton Jones, Template metaprogramming for Haskell, in: M.M.T. Chakravarty (Ed.), ACM SIGPLAN Haskell Workshop 02, ACM Press, 2002.
[39] Y. Solodkyy, J. Järvi, E. Mlaih, eXtensible Typing Library, 2007. URL: http://parasol.tamu.edu/xtl.
[40] A. Stepanov, M. Lee, The Standard Template Library, Tech. Rep. HPL-94-34(R.1), Hewlett–Packard Laboratories, 1994. http://www.hpl.hp.com/techreports.
[41] B. Stroustrup, The C++ Programming Language, Addison-Wesley, Reading, MA, USA, 1986.
[42] T.L. Veldhuizen, Using C++ template metaprograms, C++ Report 7 (4) (1995) 36–43. Reprinted in *C++ Gems*, ed. Stanley Lippman. URL: http://extreme.indiana.edu/~tveldhui/papers/.
[43] T.L. Veldhuizen, C++ templates are Turing complete, 2003. www.osl.iu.edu/~tveldhui/papers/2003/turing.pdf.
[44] M. Wallace, C. Runciman, Haskell and XML: generic combinators or type-based translation?, in: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ICFP'99, vol. 34–39, ACM Press, NY, 1999, URL: http://portal.acm.org/citation.cfm?id=317765.317794.
[45] Extensible markup language (XML™), 2005. http://www.w3.org/XML.
[46] CS/INFO 431/631: Web information systems, 2009. http://www.cs.cornell.edu/courses/cs431/2008sp/assignments.htm.