Advantages of Programmable Compile Time with Metaprogramming: the Case of ASN.1 and Perl 6

Nataliya Osipova [1][0000-0002-9929-5974], Oleksandr Kyriukhin [2]

[1,2] Kherson State University, 27, 40 Rokiv Zhovtnya St., Kherson 73000, Ukraine
natalie@ksu.ks.ua
alexander.kiryuhin@gmail.com

Abstract. With the ever-growing complexity of software systems already built and, more importantly, needing to be built in future, the need for research toward better methodologies, approaches and architectures is hard to overestimate. Ideas of novel software writing techniques are often implemented in tools in order to test out the pros and cons of the new technique, which may become generally accepted in the case of its successful application. This paper demonstrates a real world application of a recently introduced approach to the development of software libraries using advanced metaprogramming features that are present in some programming languages, notably in a recently released language called Perl 6. This research takes as a subject a well-developed problem, ASN.1 (The Abstract Syntax Notation One) implementation, with two approaches to writing a solution being common, and depicts resulting disadvantages of those approaches. Next, it describes how the use of metaprogramming in Perl 6 allows mitigating such problems, along with obstacles encountered during the development process. A number of software libraries were designed and implemented utilizing this approach, and it is used as a part of ASN.1 support for the Perl 6 language. A brief explanation of the solution's internal architecture and ideas about possible future improvements are provided.

Keywords: compile time computation, metaobject protocol, Perl 6.

1 Introduction

Various techniques can be used to reduce the complexity of implementing large-scale software systems while maintaining correctness: some are based on type-driven verification [1], some on model-driven verification [2] and some on other less popular techniques. The reason behind this is that, almost regardless of the level of abstraction provided by a common general purpose programming language, often beyond complex yet concise code pieces, a certain amount of so-called "boilerplate" code has to be written manually or generated. Manual coding done by a human is prone to errors, is known to be tedious and costly, has low ability to adapt to changes, so the idea of entrusting the process of generating the necessary code to the computer, guided by a set of rules, is not new and dates back to the first assemblers written.

ASN.1 is a part of a group of telecom standards, a standard that describes a language used for describing data structures, and it is widely used in communications and networks [3]. Thus, a number of implementations have been created over the past decades, already proven to be sound, and highly optimized. However, at the same time, selecting this particular topic renders us an opportunity to look at mature implementations' architecture, and to contrast it with the approach this research examines.

When it comes to cases of working with data structures and generating code for those structures, based on a formalized specification (which is easier to reason about compared to actual software code), two widely used implementation techniques come to mind. As the ASN.1 standard provides formal means to describe data structures and rules for its encoding-decoding process, its implementations serve us as a particular cases of the approaches:

- Writing a compiler that takes a specification as an input, parses it, forms an AST (Abstract Syntax Tree) and, using it as a guide, prepares a textual representation of the code in the target language that describes the specification in terms of both language's native type structures and specialized additional code (in the ASN.1 case it is code related to encoding and decoding data according to various encoding formats). As a next step, the generated textual code representation has to be compiled and linked to other modules of the end-user application in order to be used. This approach is typically, but not necessary, used for compiled languages including C. It was successfully applied for years and a number of commercial ASN.1 compilers were developed [4, 5].
- Writing a library that offers a set of predefined types, with which the end-user can manually derive the necessary specialized types required by the schema using the target language. Inheritance or instantiation of such types gives the user the ability to use generalized additional code "out-of-the-box" by means of code reuse. This technique can be seen as a subset of the former, with the key difference is that the end-user has to manually translate abstract types given in the specification into native programming language types offered by the library (it can be seen as the parsing and AST constructing phases of a compiler). Such an approach can be found in [6, 7].

However, both described approaches have particular disadvantages that arise from the nature of either compilation into high level language's textual representation (we will refer to it as a "two-pass compilation approach") or explicit AST-like structures usage (we will refer to it as an "AST-driven approach").

This paper explores an application of another technique, which is based on ability to execute programming language code at compile time (which takes a different angle on metaprogramming compared to Lisp-like macro support [8] and other derived AST transformation systems), that can naturally address those known disadvantages, using as an example a case of translation of an ASN.1 specification into native programming language types. We will use the Perl 6 language, which is a modern, multi-paradigm language with a rich set of features, built-in support for executing code parts at compile time [9] and an object system built upon a metaobject protocol [10].

Perl 6 is a brand new language released in late 2015 that originated from the well-known Perl programming language development process and gradually became an independent language with numerous unique features and fresh approaches to known problems in language design. As well as other languages originally created for industry purposes and not inside of the scientific community, it is yet to become widely recognized.

However, while actual popularity of a language is known to be hard to measure, steadily increasing attention to the language can be observed: new books were written over last two years [11], open source, proprietary and scientific [12] projects are written in Perl 6 along with an increasing of number of dedicated development tools [13, 14] and a growing ecosystem [15].

The Perl 6 language proves to be worth working on for its features: a lot of them are rather powerful, compared to currently popular languages' features, and a number of design solutions, including compile time code evaluation, which are unique to the language, makes it very expressive when it comes to building abstractions.

2 Related Work

The idea of transforming programming code at compile time is not new: it dates back to the first Lisp implementations with its macro system that allowed the programmer to transform program elements themselves, in effect enabling extension of the language without the need to change the language implementation. In the decades following this invention, programming language implementations were predominantly compiler-based, derived from the ALGOL family of programming language features, but starting in the late 80s and early 90s script languages have gained popularity, notable examples being Perl, Python, PHP and Ruby [16].

However, many of the currently popular programming languages support only the most basic compile time transformations such as constant folding, which are usually done by the compiler itself and are related to performance optimization. Therefore, the programmer has very little control over the compilation process. Even though they know the language well enough to do what is required at runtime, that knowledge cannot be applied at compile time.

Other than modern Lisp implementations like Common Lisp, that include a macro system, one can note a few languages that give the user the ability to do compile time calculations to a greater extent than constant folding-level computations:

- Haskell has TemplateHaskell, a GHC compiler extension, that allows the user to utilize compile time metaprogramming based on AST transformations [17].
- Perl 5, although unlike Perl 6, Perl 5 is an interpreted language, so BEGIN code blocks that run during program parse stage will be re-calculated on every script start. While it cannot be considered to be exactly compile time, it was an origin of sorts.
- C++ with its template system can be used for metaprogramming [18].

While the C++ template system is Turing-complete and can be used to implement a similar approach as this paper shows, it has to be recognized that C++ templates use a distinct “meta-language” to express its metaprogramming features, and this internal language is only a subset of C++.

On the contrary, Perl 6 presents a powerful mix between static and dynamic behavior to user, and allows execution of normal Perl 6 code at compilation time in a similar way to normal code execution at program running time.

Perl 6 implements advanced Object-Oriented Programming techniques including a Metaobject Protocol, which, among other possible applications, allows the programmer to define and patch types (OOP classes, but also roles [19], enumeration objects etc.) using so called “meta classes”. This way one can, using normal Perl 6 constructions, generate classes with attributes (data) and methods (operations), routines and more of the language’s first-class citizen values depending on currently accessible context. Generated symbols (a type or a callable object with a name) can be stored in the generated bytecode after compilation and used afterwards.

A number of popular dynamic programming languages do support metaprogramming to varying degrees, often providing ways to create and change types (even if they do not offer a complete metaobject protocol) as well as generating code at runtime. Runtime code generation can be used successfully for performance optimizations and the results can be cached [20], however it is rarely used to generate types for persistent use, mainly due to inconveniences during saving and exporting those types as they would be with normal compilation units.

The key approach to exploiting Perl 6 features that allow overcoming it was presented by Worthington [21], who stressed its usefulness for library and framework writers. This paper presents an example of real-world application of the idea along with an analysis of the benefits it brings compared to well-known approaches.

3 Overview of Common Approaches to Code Generation

As this paper overviews a number of design solutions made in mature ASN.1 implementations, it has to be understood that the aim of this research is to evaluate a different approach to designing specification-driven implementations and the developed software does not aim to compete with the mentioned implementations in terms of efficiency, ASN.1 standard coverage completeness or security (although, without doubt, those characteristics are seen as important ones and can serve as directions for further research work in this area).

Instead, it uses noted implementations as particular cases to compare with to analyze the benefits gained from a module written to take advantage of compile time evaluation with the full language available, as such module architecture can be generalized for a wider scope of cases.

Among existing free and open source ASN.1 compilers, two widely used ones were chosen as examples: asn1c[6], a compiler written mainly in C that targets C++-compatible C code, and pyasn1, set of libraries written in the Python programming language.

3.1 A Two-Pass Compilation Approach

A two-pass compilation approach schema is depicted in Fig.1. The first phase consists of using an ASN.1 compiler to produce source code in the target language. The second phase consists of using the target language compiler to process both generated source code and application code to later be linked and executed.

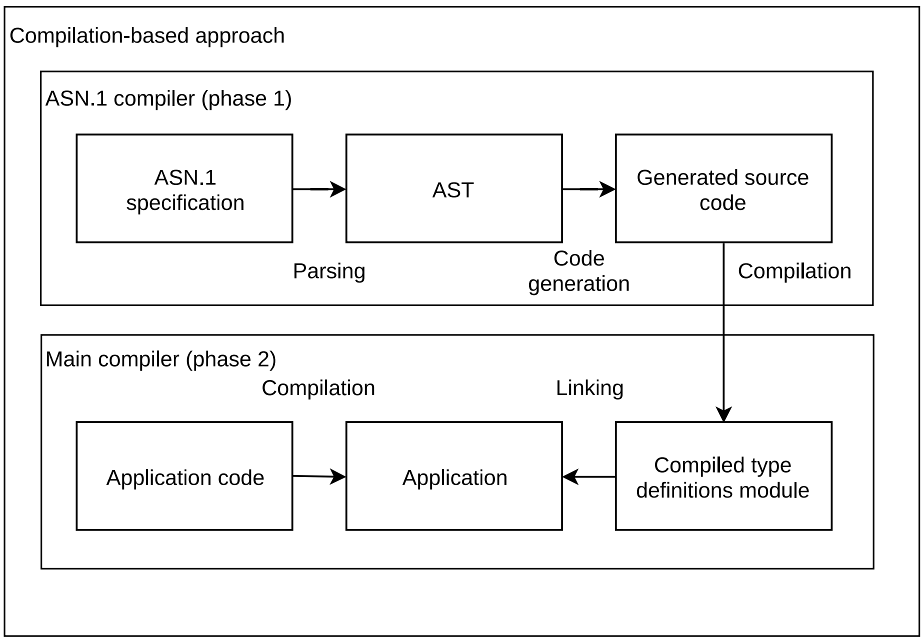

Fig. 1. Typical schema of the compilation-based approach

This approach has disadvantages both maintenance-wise and architecture-wise:

- A certain amount of additional textual sources are generated just as an intermediate form, for its later compilation into a compiled module to link an application with. In case of a gradual protocol development process, where specification and end-user code are being developed simultaneously, each change requires full generation of new sources to be compiled.
- Generated source code has to be immutable in a sense that its manual extension becomes, while possible, a maintenance burden. Changes introduced manually have to be re-applied every time the specification is changed and its re-compilation occurs as changes made will not be preserved in generated sources. Thus, in case that the end-user wants to process an ASN.1 type differently, an extension system has to be developed. With it in place, code generation can still be difficult from the point of view of typical programmer without experience in compiler writing, making extensions code more error prone.

The former issue is caused by a number of factors: lack of compiler APIs that allow for something more robust than text, but higher-level than the binary output of a compiler, and underestimation of the usefulness of metaprogramming.

The latter issue is more complex to work with, although it may be less difficult from a technical point of view. It originates from the fact that writing a particular code part most often is easier compared to writing or improving code that can generate such code parts for non-trivial cases. This issue requires additional measures to be taken to make the system extensible enough to provide general means for handling special cases, while minimizing the complexity of work with more common, simple cases.

Looking from the general point of view on this code translation approach, it still remains useful in number of situations:

- Code translation, be it immediate compilation or interpretation, from high-level language code into a target architecture is a fundamental aspect of software development.
- Various development tools, such as linters or tools for minimization of source code, do not possess the described issues by design because of their purposes: firstly, generated code is meant to be used directly (to be passed over the network, to be used as a new version of old source code, etc.), which satisfies idea of code duplication avoidance, secondly, code immutability can be either desirable or not applicable.

3.2 AST-Driven Approach

While a set of Python libraries, pyasn1 [7], offers to the user an ASN.1 compiler with Python as a target language, it also provides predefined types that can be used to manually describe an ASN.1 intermediate representation in order to manipulate it later using generic code for encoding and decoding data. Thus it can be seen as corresponding to the second technique mentioned, an AST-driven approach, depicted in Fig. 2.

While this approach is intended for rare cases when the included compiler does not support a given specification (which may happen due to fact that ASN.1 set of standards is complex and hard to implement completely) it may be also helpful when a high degree of extensibility is required for data types (classes in the OOP sense in this case). However, this approach has its own cons:

- The translation process is not automated and demands workload from the end-user.
- Such an implementation is more error prone.
- In case of upstream specification updates, manual implementation updates are necessary.
- In case of major specification changes, it is harder to reason about correctness of the update done.

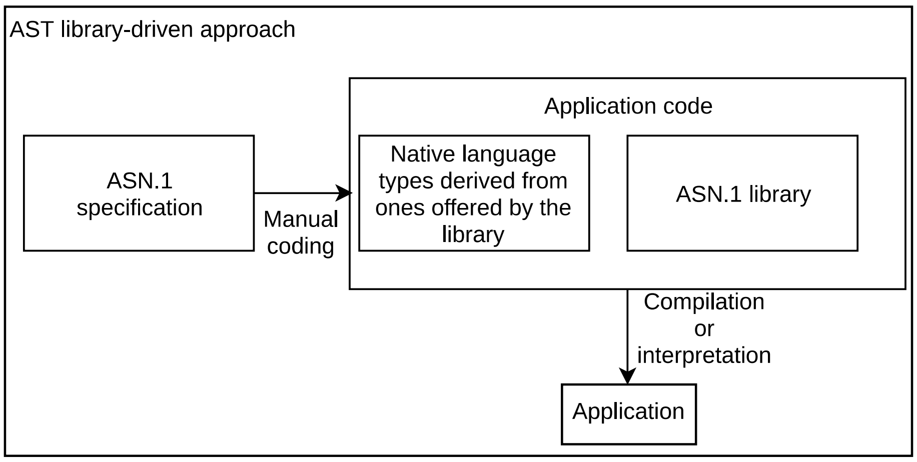
AST library-driven approach
Application code
ASN.1 specification
Manual coding
Native language types derived from ones offered by the library
ASN.1 library
Compilation or interpretation
Application

Fig. 2. Typical schema of AST-based approach

The next part explains the necessary set of features that a language should provide and architecture principles that allow a developer to partially or completely mitigate the issues described in this section.

4 Compile Time Metaprogramming Test Case: the Prerequisites and the ASN::META Module

Let's take a look at the necessary components to work with ASN.1. The prerequisites to operate on ASN.1 definitions consist of ASN.1 format file parser and a tool for decoding and encoding of data to and from the language's native types. The ASN.1 standard covers both type and value definitions enclosed in modules which can import and export types.

A common practical application of ASN.1 is the LDAP standard, and its definitions are used as a test suite for the implementation. The test suite consists of a single ASN.1 module which is mostly comprised out of type definitions.

4.1 Parsing Module: ASN::Grammar

Given a textual representation of an ASN.1 description of LDAP types, the module has to parse it and convert into it native Perl 6 types that user can work with later. With Perl 6 as a main implementation language, considering its relations to Perl family, implementing a parsing component for a certain textual format requires little effort and is handled by a grammar [21] that describes a subset of the ASN.1 standard which is required for full parsing of the LDAP specification.

The subset covered is summarized in table 1.

Table 1. Scope of the ASN.1 specification subset implemented

Definition	ASN.1 specification	ASN::Grammar
Simple types	+	+
String types	+	+/-
Complex types	+	+
Simple values	+	+/-
String values	+	-
Complex values	+	-
Recursive types	+	+/-

Here is an example of two types and a value definition in ASN.1 notation:

```
A ::= SEQUENCE {
    id INTEGER (0 .. maxInt),
    message OCTET STRING,
    value1 [0] INTEGER OPTIONAL,
    value2 [1] INTEGER OPTIONAL
}
B ::= NULL
maxInt INTEGER ::= 2147473647
```

Types can be divided into native types and custom types. Native types are predefined in the ASN.1 standard and can be simple (a type that represents a single piece of data, such as INTEGER, NULL or OCTET STRING) and complex (a type that represents a set of values, ordered or not, of certain types, such as SEQUENCE). Custom types are defined by a particular specification itself and can be used everywhere where native type can be used. Constraints (such as the type of the “id” field is constrained to be within the range of positive numbers up to the value defined later as “maxInt”) and traits (fields “value1” and “value2” are optional in “A” type) can be applied to a type.

It is important to note the so-called “tags” in the “value1” and “value2” field definitions. The ASN.1 group of standards is divided into two parts: a description of the ASN.1 schema language and encoding-decoding rules for values of noted types, with these two parts being loosely coupled by common type names. A number of encoding-decoding rules exist, including binary encoding formats, such as BER (Basic Encoding Rules), DER (Distinguished Encoding Rules), CER (Canonical Encoding Rules) and PER (Packed Encoding Rules), as well as textual formats such as XER (XML Encoding Rules) or ASN.1 SOAP.

Values encoded using binary encoding rules for type A can be seen as ambiguous, because an integer parsed after the “message” field value can be recognized as either the value of the “value1” field or of the “value2” field. This case is resolved using tags – binary encoding rules’ standards that do not include explicit field names along with the value itself encode its tag according to particular tagging schema, so the original value can be parsed without ambiguity.

The ASN::Grammar module provides a “grammar” – a built-in Perl 6 mechanism for writing parsers, comprised from rules defined in the grammar’s body block. Each syntax part is described as a rule using the Perl 6 rule syntax, which is somewhat based on traditional regular expression syntax. Rules can refer to other rules and be recursive, allowing for parsing of non-regular languages. With the ability to express a parser in this way, the definition of a grammar is similar to a syntax definition that is made in Backus-Naur form, making such porting trivial. An example of rules in the grammar:

```
grammar ASN::Grammar {
    # Basic part
    token TOP { <module>+ }
    rule module { \n* <id-string> \n* 'DEFINITIONS' <de-
fault-tag> '::=' 'BEGIN' \n* <body>? \n* 'END' \n* }
    rule default-tag { <( <explicit-or-implicit-tag> )>
'TAGS' }
    token body { [ <type-assignment> || <value-assign-
ment> ]+ }
    …
    # Type part
    rule type-assignment { <id-string> '::=' <type> }
    …
}
```

Internally, Perl 6 analyzes the grammar description and automatically produces both a lexical analyzer and a parser, avoiding having to write and maintain the two separately. This may be considered as the frontend part of the first pass of compilation in a two-pass compilation approach. Thus, we don't need to resort to an external lexical analyzer and syntactic parser generator tools. As the test document does not contain complex values, this part of the ASN.1 specification was deliberately omitted when implementing the ASN::Grammar module. As a next step, a class in ASN::Grammar describes actions to be done on parse tree produced by the grammar. It converts the parse tree into a more useful Perl 6 data structure: class instances, strings, integers, lists and associative maps that hold all necessary information for further type generation: particular types, constraints, tag information etc.

4.2 Encoder/decoder Module: ASN::BER

The second necessary component is an encoder-decoder library which can help the user to map data values of certain native Perl 6 types into binary data and parse it back. It is named ASN::BER as it is implemented according to the Basic Encoding Rules (BER) standard (which is used in LDAP specification used as a test suite), using both native language types and a set of rules with extensive use of multiple dispatch and the meta-object protocol [22].

ASN::BER can be used in a manner similar to that described in section 3.2: it provides an encoder and decoder tool that works utilizing a mapping from ASN.1 types into native Perl 6 types which the user can use to express the necessary ASN.1 type definitions manually.

Both built-in and constructed types (such as OOP classes and roles) were used in the mapping. Its simplified form (with trivial and repetitive cases being omitted) can be seen in table 2.

Table 2. Key part of mapping of ASN.1 types into Perl 6 native types

ASN.1 type	**Perl 6 mapping**
INTEGER	Int
NULL	class ASN-Null
OCTET STRING	Str
UTF8String	Str
ENUMERATED	Perl 6 enum
SEQUENCE	A class that implements the ASNSequence role
CHOICE	A class that implements the ASNChoice role

At the beginning, both OCTET STRING and UTF8String (as well as all other string types) are mapped into the Perl 6 “Str” type which represents a string. However, string types in ASN.1 can demand different binary representations or enforce specific restrictions based on exact type, so the ASN::BER module has to have a means to get access to the particular ASN.1 string type that is implied for an encountered Perl 6 Str value.

This is achieved by usage of roles that are mixed into the attribute metaobject if the string is a part of a complex type, or by deriving a class from the role itself during compilation time.

A typical ASN.1 string type implementation is made using three parts shown below:

```
# UTF8String wrapper role
role ASN::Types::UTF8String does ASN::StringWrapper {}
# “Trait” definition
multi trait_mod:<is>(Attribute $attr, :$UTF8String) {
    $attr does ASN::Types::UTF8String
}
# Usage example
class A does ASNSequence {
    …
    has Str $.name is UTF8String;
    …
}
```

Firstly, an ASN::Types::UTF8String role is declared. Secondly, a “trait” is declared which can be applied to an Attribute (class attributes are first-class citizens in Perl 6). Finally, an example of such an application is presented. Perl 6 traits are similar to Java annotations in the sense that they are being applied to declarations based on special syntax and such application executes certain code on this declaration, but unlike Java annotations the trait code runs at compile time and so can participate in the compilation process (in this case, it adds a role to an attribute).

Complex types, such as SEQUENCE or CHOICE, can be expressed by composition of roles offered in ASN::BER module. A user has to provide the data necessary for the encoding-decoding process by implementing methods required by those roles.

Considering an ASN.1 type specification and with help of types of the module, a user can express the necessary type, construct a value of this type and pass it to the encoder, as shown in case 1 in Fig. 3.

A type value is a value that has to be encoded and the definition type is type information that is obtained by the ASN::BER encoder from the value itself using the metaobject protocol and data provided in method implementations of a particular role. For the passed value, the encoder calls appropriate multimethod based on the value's type. For complex values, information about attributes that comprise the complex value is obtained by a call to an abstract method that returns attribute names in a particular order. In this enforced order attributes of the class are retrieved and their value is encoded based on its own appropriate type and value (if present). As a next step, encoded bytes for each part of a complex type are concatenated and returned as the encoding result of the type. This way, the implementation is made to be compact, extensible and self-contained (no metadata has to be passed other than a typed value to get encoding results).

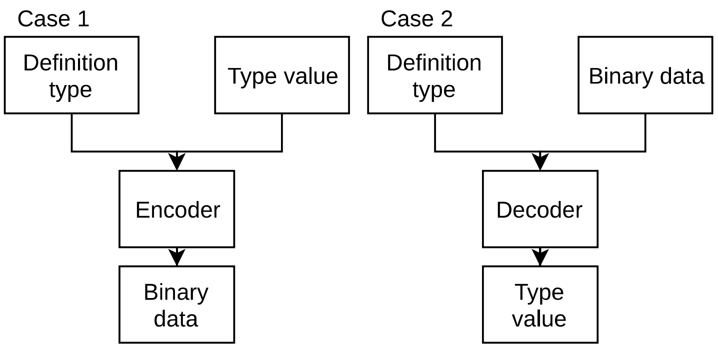

Fig. 3. Encoder/decoder schema

The decoder architecture is made in a way that allows it to imitate the encoder structure to a high extent as shown in case 2 of Fig. 3. The main difference compared to the encoder implementation lies in definition type: while in the case of the encoder its role is taken by type metadata that is stored in a value itself, the decoder has to be instantiated with a type object to take this role. In Perl 6, types (including classes, roles, metaobjects etc.) are first-class citizens and are represented as type objects. Using a type object, all necessary information for decoding a type can be gathered. Abstract role methods overloaded by the user in case of a complex type that provide necessary information for decoding, for example, order of attributes in a complex type, can be called on the type object as well as on type instances during the decoding process.

For every simple and complex type a value is decoded. Simple values can be immediately returned. Parts of the value of a complex type are added into an associative map, and, as a next step, the map is passed into the type constructor method called on the type object as arguments.

This way, normal native Perl 6 types have to be mixed with types offered by ASN::BER module only to offer metadata that is used during binary encoding (such as tags, exact ASN.1 string type etc.) and metadata that helps extensibility (for example, an order of attributes and their names in a complex value is given explicitly, as a user may want to store additional, non-ASN.1 data, in the class, and a heuristic that simply collects all type attributes makes it impossible to implement without needing to introduce additional rules for naming or composition schemas).

The architecture of the ASN::BER module is not specific to the BER standard, and can be generalized to handle other encodings. With a common type interface, encoder and decoder can be extracted into abstract roles. Each specialized encoder and decoder pair is required to implement a number of multimethods that handle particular operations for specific types according to specialized rules enforced by the specification.

As the LDAP protocol is based on the BER standard, the primary goal was to implement this particular standard, while designing such a module architecture that will be generic enough to ensure a high level of extensibility for implementing other standards.

4.3 ASN::META Module

The core of the system is a module called ASN::META. It utilizes the described approach of combining compile time code execution and the metaobject protocol to serve as a bridge between the specification and end-user application [23].

The overall architecture is depicted in Fig. 4. Three main components are necessary: the source text of the LDAP ASN.1 specification, an end-user application that uses the types described in the specification, and the ASN::META module. It is worth noting that while this implementation is specialized to ASN.1, the generic approach demonstrated is not tied to particular library implementation or task.

As mentioned before, in Perl 6 it is possible to explicitly run arbitrary Perl 6 code at compile time: a number of language constructions can be used to run code during a compilation unit compilation process. The user of the library can pass data known at compile time to the library and it is able to export different sets of symbols depending on context (which includes the data passed). Exploiting this feature, the application code passes a path to an ASN.1 specification to the library.

The Perl 6 compiler starts compiling ASN::META. It receives the path to the specification from the import statement, retrieves the specification text, parses definitions using the ASN::Grammar module and generates a number of types based on it using the metaobject protocol. The created types are exported and their metaobjects serialized into the bytecode of the importing module.

Application code imports the types generated at compile time as if they were defined explicitly in the ASN::META module.

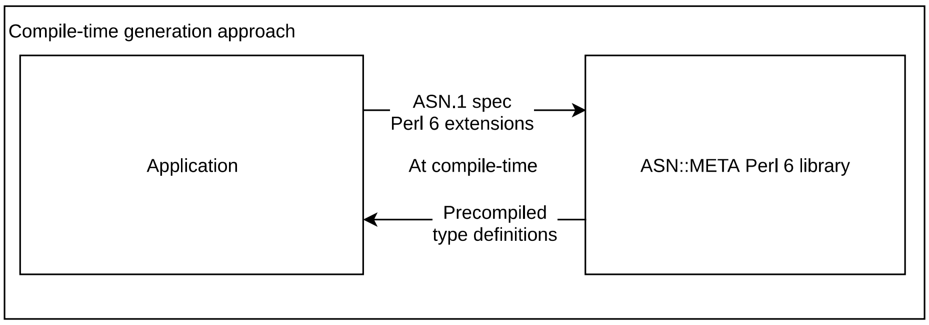

Fig. 4. Compile time type generation approach application

Application's possible ASN::META module inclusion code:

```
use ASN::META <file docs/spec/ldap.asn>;
# Example of compilation unit inclusion statement is
given below
use Foo;
```

As shown, the statement that includes a compilation unit with certain data (a list of strings formed from two elements in this case) does not differ much from the common module inclusion statement presented next in the code example. The path passed is received by the library code and then the specification is processed as described above.

As opposed to compilation into a textual representation of the high-level language, no additional textual sources are generated. Data types are generated using the metaobject protocol and then stored by the Perl 6 compiler as a bytecode as if they were defined in the library using textual representation. A bytecode representation pre-compiled this way is stored and is being reused by the Perl 6 compiler with the application as long as the specification path remains the same (or a recompilation can be forced if necessary for other conditions).

This is an example of code taken from the META::ASN module, which generates classes for the ASN.1 SEQUENCE OF type, with additional comments added:

```
# New class is created with name of $symbol-name
my $new-type = Metamodel::ClassHOW.new_type(name => $sym-
bol-name);
# ASN.1 local tag is assigned to lexical variable
# to be captured into method's closure
my $tag-value = .value;
# Add necessary method to newly created class
$new-type.^add_method('ASN-tag-value', method { $tag-
value });
# If type of SEQUENCE OF is a builtin:
if $of-type.type (elem) $builtin-types {
    $new-type.^add_role(Positional[$of-type.type]);
} else { # Else just compile inner type
    $new-type.^add_role(Positional[compile-type($of-type,
%POOL, $of-type.type)]);
}
# When Positional role that indicates SEQUENCE OF type
# is applied to new class, compose type object instance
$new-type.^compose;
# At last, populate custom types pool
# to cache custom types already compiled
return %POOL{$symbol-name} = ('SEQUENCE OF', $new-type);
```

Perl 6 differentiates precompiled modules based on passed data. The created metaobjects are saved into the importing module's precompilation data. It makes it possible to work with multiple end-user applications that use different ASN.1 specifications without interference between them. For the end-user it looks like the ASN::META module provides different type definitions for every specification passed, while it has none and consists of purely type generator code.

Generated types are based on the specification passed and do not need manual encoding as opposed to an AST-based approach. However, a system of type generator extensions can be implemented if demanded: as the whole system is language-heterogeneous, the end-user can write metaobject protocol-based code that transforms the type created based on certain conditions. Then, the ASN::META library receives the extensions as files and evaluates the code passed for necessary types, allowing for flexible handling of special cases. Besides the described ad-hoc solution, other extension system architectures are possible.

4.4 Encountered Issues

Considering the points above, it is worth noting certain obstacles that were encountered as well as scope limitations of this paper:

- As Perl 6 was released not so long ago, relatively advanced features such as metaobject protocol support remain a complex aspect of the language usage. Two bugs related to class attributes attaching to a class using the metaobject protocol were discovered by us and fixed by Rakudo team core members.
- During this research the most complete Perl 6 compiler, Rakudo, was used and while the speed of code it generates is being quickly improved by its developers, it still cannot compete with more mature implementations of scripting languages in terms of efficiency.
- The presented parsing module, ASN::Grammar, does not cover a complete ASN.1 grammar, which is a purely implementation matter and its coverage and security is to be improved eventually.

It is worth noting that the described issues are not a result of the usage of the specific approach chosen, but rather temporary, implementation-only issues that can be mitigated. Rakudo matures with time and its stability and generated bytecode performance is being constantly improved. The ASN::META, ASN::Grammar and ASN::BER modules are still being developed and the issues will be fixed eventually.

We can note discovered implementation-independent issues of the approach:

- For implementers, such a structure is more complex when it comes to debugging. Bugs in code related to precompilation or a metaobject protocol implementation are harder to debug compared to a situation where the target is source code.
- The approach demands a rich runtime environment to work. If the task demands an efficient implementation, both memory-wise and performance-wise, generating a highly optimized C code for a bare metal platform is the best possible approach.

Given that the aim of this paper is to apply a compile time metaobject programming approach and analyze its advantages in solving the problem, the successful creation of a useable ASN.1 toolchain that is suitable to serve as a foundation for LDAP implementation in Perl 6 meets this goal.

5 Conclusions

It can be concluded that the compile time metaobject programming approach used in this paper can serve as a foundation for a sophisticated architecture for software libraries. Being the case of metaprogramming with reflexive and self-modifying code it makes it possible to build complex systems without introducing an additional maintenance burden and keeping whole code base concise.

This test case implementation, a set of modules to work with ASN.1 in Perl 6, while having room for further improvements (improving coverage of ASN.1 syntax that the system can understand and compile, increasing test coverage, improving performance and security means, implementing advanced features such as extensions support etc.), has shown not only the ability to use Perl 6 for rapid development and producing compact code, but also the approach itself being able to introduce new solutions to known issues and being able to compete with well-known approaches to resolving a code generation issue, fusing theirs best parts while reducing disadvantages.

While the main objective was to observe and analyze particular traits that emerge from the use of compile time metaprogramming outside of the scope of a macro system in particular the test case of ASN.1 implementation for a programming language, this work provides a solid foundation architecture for LDAP implementation.

References

1. Brady, E.: The IDRIS programming language – implementing embedded domain specific languages with dependent types. In: Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, 8 July–20 2013, Revised Selected Papers, pp. 115–186 (2013). doi: 10.1007/978-3-319-15940-9_4
2. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. NCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008). doi: 10.1007/978-3-540-69850-0_1

3. International Telecommunication Union Homepage, Application fields of ASN.1. https://www.itu.int/en/ITU-T/asn1/Pages/Application-fields-of-ASN-1.aspx
4. Langendoerfer, P., Koenig, H.: COCOS – A configurable SDL Compiler for Generating Efficient Protocol Implementations. In: Dssouli, R., Bochmann, G.v., Lahav, Y. (eds.) SDL 1999 (1999). doi 10.1016/b978-044450228-5/50018-7
5. asn1c home page, Open Source ASN.1 Compiler. https://github.com/vlm/asn1c
6. Ai-qing, Y., Sheng-sheng, Y., Jin-li, Z., Hong-xing, G.: C++ template data structure mappings of ASN. 1 of ITU-T X. 680 suitable with PER (X. 691) and BER (X. 690). Wuhan University Journal of Natural Sciences. 5, 285-288 (2000). doi 10.1007/BF02830135
7. ASN.1 library for Python Homepage, http://snmplabs.com/pyasn1/
8. Seibel, P.: Practical COMMON LISP. Apress, Berkeley, CA (2005). doi 10.1007/978-1-4302-0017-8
9. Lenz, M.: Perl 6 fundamentals. Apress (2017). doi 10.1007/978-1-4842-2899-9
10. Kiczales, G., Des Rivières, J., Bobrow, D.: The art of the metaobject protocol. MIT Press, Cambridge, Mass. (1991). doi 10.7551/mitpress/1405.001.0001
11. Perl 6 Books Homepage, https://perl6book.com/
12. Merelo Guervós, J.J., García-Valdez, J.M.: Going stateless in concurrent evolutionary algorithms. In: Figueroa-García, J.C., López-Santana, E.R., Molano, J.I.R. (eds.) Applied Computer Sciences in Engineering - 5th Workshop on Engineering Applications, WEA 2018, Medellín, Colombia, October 17-19, 2018, Proceedings, Part I. Communications in Computer and Information Science, vol. 915, pp. 17–29. Springer, Cham (2018). doi 10.1007/978-3-030-00350-0_2
13. Cro Services Homepage, https://cro.services/
14. Comma IDE Homepage, https://commaide.com/
15. Perl 6 Modules Homepage, https://modules.perl6.org/
16. Sebesta, R.: Concepts of programming languages - 10th edition. Pearson Addison Wesley, Boston (2012).
17. Sheard, T., Jones, S.: Template meta-programming for Haskell. ACM SIGPLAN Notices. 37, 1-16 (2002). doi 10.1145/581690.581691
18. Solter, N., Kleper, S.: Professional C++. Wiley, Indianapolis, IN (2005).
19. Scharli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003). doi 10.1007/978-3-540-45070-2_12
20. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. Parallel Computing. 38, 157-174 (2012). doi 10.1016/j.parco.2011.09.001
21. Getting beyond static vs. dynamic slides, Jonathan Worthington, http://jnthn.net/papers/2015-fosdem-static-dynamic.pdf
22. ASN::Grammar code repository, https://github.com/Altai-man/ASN-Grammar
23. ASN::BER code repository, https://github.com/Altai-man/ASN-BER
24. ASN::META code repository, https://github.com/Altai-man/ASN-META