Imperial College London
Department of Computing

Diffingo
Generating application-specific message parsers and serializers

by

Eric Seckler

Submitted in partial fulfilment of the requirements for the MRes Degree in High Performance Embedded and Distributed Systems (HiPEDS) of Imperial College London

September 4, 2015

Abstract

Network functions and applications, such as proxies, load balancers, or intrusion detection systems, use parsers and serializers to translate messages they process into an application-accessible format. Developers typically employ custom parser and serializer implementations and optimize them for their specific application to achieve the best possible performance. However, because these parsers and serializers are hand-rolled and tightly integrated with specific applications, they are difficult to reuse and maintain and prone to bugs, which often result in vulnerabilities.

To overcome these issues, we propose Diffingo, a framework for generating application-specific message parsers and serializers from reusable message format specifications. Diffingo aims to generate highly efficient code. For this purpose, it employs static analysis to determine which parts of a message are accessed by a particular application and generates parsers and serializers optimized for use in this application. Furthermore, Diffingo supports incremental parsing of messages, avoids dynamic memory allocations, and is designed to integrate with a user-level TCP stack.

Our approach facilitates the reuse of message format specifications across applications and reduces the risk of programming errors and security holes in the input processing code of a network application. A preliminary evaluation of our Diffingo prototype on the Memcached binary protocol shows promising results. Using our optimizations, we can achieve a performance comparable to or better than that of a hand-rolled parser and serializer.

Acknowledgments

This work would not have been possible without the support and help from many people. I am deeply grateful to my supervisor, Alexander Wolf, whose feedback and advice were invaluable both during the project and in the preparation of this report. I would also like to thank him for finding time even for long discussions in these busy times, as well as for his confidence in me and the project and his support of the abrupt shift in its topic.

Further, it has been a pleasure working with Paolo Costa and Peter Pietzuch. Paolo supported me with very useful advice both for the project, as well as research in general, and always had an open ear for me. Likewise, Peter provided valuable feedback and ensured that the results of the project would integrate well with the Flick project. A big thank you to both of you!

I would also like to thank my fiancée, family, friends, and coworkers for moral support and technical discussions. Special shouts go out to all HiPEDS'lers and members of the large-scale distributed systems group.

Finally, I'm very grateful to EPSRC for their financial support of my studies.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The recent trend towards virtualizing network functions [12, 17, 35] has renewed the interest in the design and implementation of highly efficient software-based network applications, such as proxies, load balancers, firewalls, intrusion detection systems, and alike. Researchers have proposed various languages, frameworks, and tools that support the development of such applications [32, 35, 31, 42, 45]. However, most current work in this area focuses on applications that perform packet-level processing, while we focus on those that process application-layer messages.

The functionality of such application-layer network applications can generally be divided into three basic components: First, the application needs to parse messages it receives from their wire-format into an application-accessible representation. This includes determining message boundaries in the input stream as well as value transformations, such as converting integer endianness or string encoding. Next, it can process the parsed messages and perform necessary computations. Lastly, it may forward the potentially modified messages or send replies or other messages by translating (serializing) them back into their wire-format. We can model these components as processing stages in a data flow, as shown in Figure 1.1.

Instead of reusing existing parsers and serializers, today's network applications typically use custom, hand-rolled implementations to realize the transformation stages. This is primarily in order to ensure their efficiency, as it allows two key optimizations: First, applications often only need access to a subset of the information encoded in the messages. Utilizing a custom implementation allows application developers to ignore unimportant aspects of the message during parsing. Second, developers can integrate parts of other application processing with the parsing and serialization logic to increase the overall application performance.

As a result, developers sacrifice the modularity of the parser and serializer implementations as well as of the application as a whole. This makes the applications difficult to maintain and debug, and further limits the reuse of their parsers and serializers in other applications. In addition, developers have to take extreme care when implement-

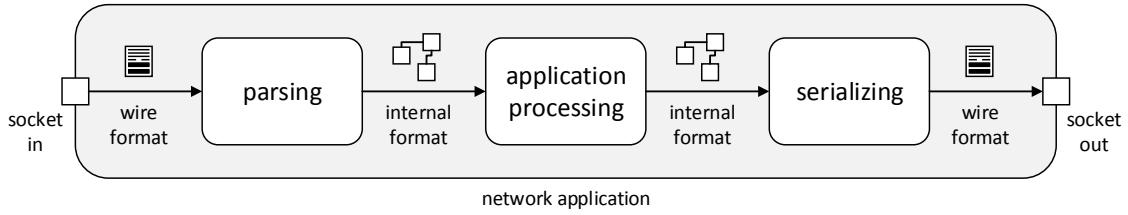

Figure 1.1: Parsing and serializing stages in a network application.

ing the custom parsing logic. Parsing and processing application inputs is generally an error-prone task and bugs can often lead to exploitable security holes.

In this work, we try to overcome these issues by generating optimized and application-specialized parsers and serializers from reusable message format specifications. For this purpose, we devise Diffingo, a new parser/serializer generation framework. Diffingo builds on the following insight: By employing static analysis, we can (over-)approximate the parts of parsed messages that a particular application accesses. Utilizing this information, we can then specialize the generated parser and serializer to ignore the remaining message parts.

In the design of the parsers and serializers generated by Diffingo, we further keep efficiency in mind: The parsers and serializers can pause during the processing of a message and resume where they left off; this is of particular use when messages arrive in a piece-meal fashion at a network socket. Additionally, the generated code relies on pre-allocated memory buffer pools, rather than using dynamic memory allocations. Lastly, we designed the parsers in such a way that they can be integrated into a user level TCP stack, such as mTCP/DPDK [28, 26], which can reduce memory copies by storing pointers into the TCP packet buffers instead.

Our approach facilitates a modular application development and allows reusing message format specifications across applications, while simultaneously enabling the optimization of generated parsers and serializers to specific application requirements. Furthermore, by automating the generation of parsers and serializers, we reduce the risk of programming errors and security holes in the input processing code of the application.

Diffingo follows in the footsteps of two other similar parser generator projects aiming at network protocols: Spicy [46] (the successor of BinPAC [39]) and Nail [5]. Diffingo uses a language to describe message formats that borrows heavily from Spicy, but extends it with support for value and stream transformations, as well as serializing messages, which is influenced by similar concepts from Nail. In contrast to Spicy and Nail, Diffingo makes parser and serializer state explicit to support efficient resumable parsing, avoids dynamic memory allocations, and optimizes parsers and serializers for use in specific applications.

Our efforts also relate to work in other contexts: Languages and parser/serializer generators for ad-hoc data formats, such as PADS [21] and DFDL [7], use similar abstractions, including field compositions, to define data formats, but do not typically target high-performance network applications. Data serialization frameworks, such as ASN.1 [27] or Protocol Buffers [23], aim to write and read data items to/from specific encodings, but define only the logical format of data items as schemata and not their wire-format. Finally, parser generators and combinators, which are used primarily to parse programming languages, employ similar compositional mechanisms and grammars as Diffingo, but separate the construction of internal representations of the parsed data from their grammar language and do not consider serialization.

1.1 Contributions

The contributions of this work are as follows:

- The design of Diffingo, a parser/serializer generator framework that specializes generated parsers and serializers to application needs (Chapter 2).
- A corresponding message format specification language, which builds on the language used by the Spicy parser generator (formerly Binpac++) [46] and supports length-encoded and look-ahead parsing, as well as value transformations (Chapter 3).
- Mechanisms to optimize the generated parsers and serializers, including mechanisms to statically determine those parts of a message that are not required in the context of a specific application and to simplify the message format specification accordingly, thereby removing unnecessary computation (Chapter 4).
- A prototype implementation of large parts of the presented framework and mechanisms in C++ (Chapter 5), as well as a preliminary evaluation of the presented ideas in the context of this prototype implementation and an exemplary use case (Chapter 6).

In addition, this report also includes a discussion of related work (Chapter 7) and a conclusion (Chapter 8).

1.2 Motivational example

Throughout this report, we will use a Memcached request router as an example application to demonstrate the motivation of design decisions. In this section, we briefly introduce this application, the messages exchanged, and the opportunities for specializing the generated message parser and serializer for use in this application.

Memcached [36] is a distributed in-memory key-value store that is often used to cache small items or chunks of data for delivery to clients. The Memcached request router we consider here acts, similar to mcrouter [34] and twemproxy [43], as a load balancing proxy for messages sent to a Memcached server. Data is distributed across multiple Memcached instances according to a hash function, which is applied to the key of each entry. The router receives request messages from clients, applies the hash function to the key contained in each request, and forwards them to the corresponding server instances.

Our router is based on the binary Memcached protocol [37]. We visualize the wire-format of the Memcached messages exchanged in this protocol in Listing 1.1. A message consists of a header (bytes 0-23) followed by variable length fields *Extras*, *Key*, and *Value*. The lengths of these fields are given as values in the header. In addition, the header also contains multiple other fields describing various attributes of the message, such as a magic number denoting whether a message is a request or response, a status code, and a field whose contents will be echoed back by a Memcached server. Integer field values are encoded in network byte order (big-endian) on-the-wire, but application processing typically occurs on a little-endian system.

The router application, however, does not need access to the values of all fields. In order to implement the processing logic of the application itself, it only needs access to the (parsed) contents of the fields *Opcode* (to distinguish between different kinds

```
Byte/     0       |       1       |       2       |       3       |
   /              |               |               |               |
  |0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|
  +---------------+---------------+---------------+---------------+
 0| Magic         | Opcode        | Key length                    |
  +---------------+---------------+---------------+---------------+
 4| Extras length | Data type     | vbucket id / status           |
  +---------------+---------------+---------------+---------------+
 8| Total body length                                             |
  +---------------+---------------+---------------+---------------+
12| Opaque                                                        |
  +---------------+---------------+---------------+---------------+
16| CAS (data version)                                            |
20|                                                               |
  +---------------+---------------+---------------+---------------+
24| Command-specific extras (as needed)                           |
 +|  (note length in extras length header field)                  |
  +---------------+---------------+---------------+---------------+
 m| Key (as needed)                                               |
 +|  (note length in key length header field)                     |
  +---------------+---------------+---------------+---------------+
 n| Value (as needed)                                             |
 +|  (note length is total body length header field, minus        |
 +|   sum of the extras and key length header fields)             |
  +---------------+---------------+---------------+---------------+
  Total 24 bytes header + body
```

Listing 1.1: Memcached binary protocol message format (adopted from [37]).

of requests) and *Key* (to compute a hash), if it is available. In order to determine the boundaries of the *Key* field, as well as the message boundaries themselves, the values of the *Key length*, *Extras length*, and *Total body length* fields also need to be accessed during parsing of the message. The values of all other fields, including, for example, the extras, value, magic, and status code fields, are neither necessary to determine message or field boundaries, nor used in the decision making process of where to forward the message to and, therefore, do not need to be parsed or made application-accessible.

In this example, Diffingo will generate a parser and serializer that combines these unused fields together in larger, simplified fields, to which no further parsing logic is applied. For instance, the *Opaque*, *CAS*, and *Extras* fields can be combined into a single field with a length of 12 + *Extras length* bytes. If the application does not need to re-serialize the message again, the contents of these unused fields can be skipped over by the parser and thrown away immediately. In the case of the Memcached request router, however, the message will later be forwarded and, therefore, the on-the-wire contents of these fields are later needed for serialization. In this case, the combined values will simply be copied into the internal representation of the message. To avoid these copies, the parser can also store pointers into the input data buffers (for example, the underlying TCP packet buffers) instead. We discuss these mechanisms in detail in Chapter 4.

While this example application uses a protocol with length-encoded fields (as is typical for binary protocols), other common applications also employ delimiter-encoded fields (as, for example, is more typical for textual protocols, such as HTTP[1]). Combining length-encoded fields into larger fields is a relatively simple task; doing the same for delimiter-encoded fields is more complex. We discuss the differences between and mechanisms for handling either case in Chapter 4.

[1]Hypertext Transfer Protocol [18]

2 Design of the Diffingo parser/serializer generator

In this chapter, we outline the general design and architecture of the Diffingo parser/serializer generator framework. We first describe the interaction of the architectural components of the framework (Section 2.1). Subsequently, we discuss supported message formats, parsing/serializing characteristics, and other design decisions (Section 2.2).

2.1 Framework architecture

Diffingo translates message format specifications into message data structures, parsers, and serializers in the context of a specific application. We visualize the components interacting in this process in Figure 2.1.

The Diffingo generator takes two items as inputs: A message format specification, which specifies a message's wire-format and its internal data structure representation in a domain specific language (Chapter 3), as well as an instantiation of this specification in the context of an application. Such an instantiation configures which parts of a parsed message need to be accessible by the application. The generator then creates an optimized version of the message format specification for this instantiation by removing or combining irrelevant parts and uses it to generate data structures that describe the internal representation of a message, as well as a parser and serializer that can transform messages from wire-format to internal representation and vice versa.

While it is possible to manually specify the configuration of instantiations, we argue that they can also be generated in an automated way by statically analyzing the source

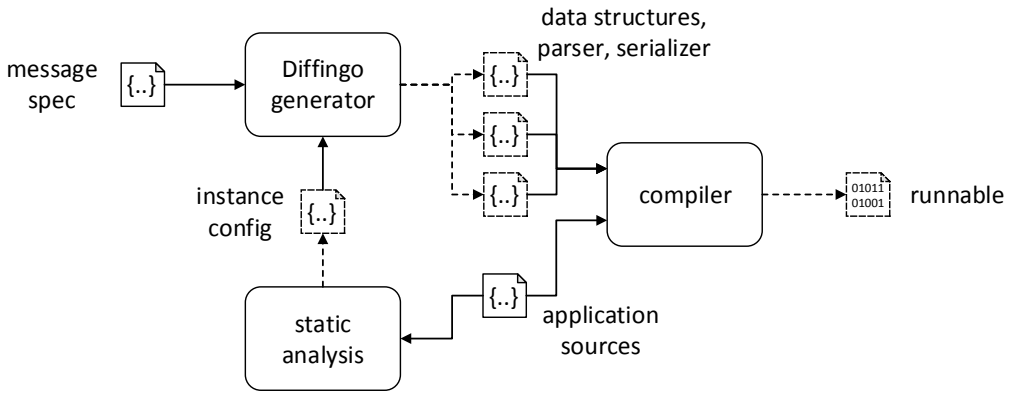

Figure 2.1: Architecture of the Diffingo framework. Generated artifacts are shown with dashed outlines and arrows, inputs to stages with solid arrows.

code of a specific application to determine (over-approximate) which parts of a parsed message are accessed by the application. In particular, strongly typed languages, such as the Flick language for network functions [1], can make this analysis simple.

In a final step, the outputs of the Diffingo generator are fed together with the application source code to a compiler (or interpreter, depending on the target language), which creates a runnable program.

2.2 Message format, parser, and serializer design

Diffingo's design needs to translate between wire-format and internal representation of messages while supporting common network application protocol messages to be applicable to a wide range of use cases. In this section, we discuss how Diffingo aims to achieve these goals.

2.2.1 Relationship of wire-format and internal representation of messages

The message format specification that serves as primary input to Diffingo's generator describes the wire-format of a message. Simultaneously, it also determines its internal representation as a data structure in the application's host language. That is, Diffingo's specification language allows expressing both parsing and serializing rules, as well as whether and how parsed data is stored in the internal representation. For example, it allows transforming values into application-accessible formats and hiding meaningless information, such as white-space characters or constant values, from the application by excluding it from the internal representation.

As a result, parsing and re-serializing a message may not result in bit-identical inputs and outputs. The result of such a transformation, however, will be semantically equivalent to the original input (in the sense that it would be parsed into the same internal representation).

2.2.2 Modular messages, parsers, and serializers

To support a wide variety of protocols, Diffingo uses an extensible modular architecture for the specification of message formats. The basic elements of a message in this architecture are atomic and composed fields, which also correspond to atomic parsers/serializers and composed parsers/serializers. We model the types of composed fields and messages as so called *units*, which consist of a sequence of (atomic or composed) fields.

Atomic fields fulfill a role similar to a lexer in a traditional parser [1]: They specify how to parse the next token in the wire-format of the message. Additionally, they also specify the field of the unit in which to store the parsed value of the token and any potential conversions that need to be applied to it, as well as how the field's internal value can be (re-)serialized into a wire-format token. The format of the token is determined by a type associated with the atomic field.

[1] Traditional parsing is often split into two phases: a lexing phase that determines token boundaries and a parsing phase that works on the tokens identified by the lexer.

Units combine multiple atomic or composed fields and specify their composed parsing and serialization behaviour; thereby, they resemble the role of grammars in traditional parsers. For instance, units can link other fields sequentially together, contain repetitions of a single field, or make choices between different fields. Diffingo's specification language (Chapter 3) includes elements that facilitate declaring these relationships.

This modular architecture also allows Diffingo to support embedding messages of one protocol into messages of another: For instance, many applications use the HTTP protocol as a lower layer transport protocol for their own application messages; Hadoop's [3] result messages are one such example. In this case, an instantiation of Hadoop's result message format specification in Diffingo could declare that the result messages are embedded in the body of an HTTP message, whose format specification is also given.

2.2.3 Look-ahead parsing

To support delimiter-encoded or more complex protocols, Diffingo can generate parsers with a one-token look-ahead. That is, when the parser encounters a state where two or more different continuing paths could be chosen, it can match the value of the next token on the input stream with the respective first tokens of the different paths to make a decision. We chose a one-token look-ahead, as most common protocols and languages are designed not to require larger look-aheads [4].

This allows Diffingo to support many common textual protocols. For example, an HTTP message contains a list of header lines which are separated from its body by an empty line. With a one-token look-ahead, the parser can identify whether the next line contains another header line or whether it is empty and the list of header lines is complete.

Diffingo thus supports a super-set of LL(1) grammars (assuming a left-to-right parsing implementation with leftmost derivation). Diffingo's grammars, however, are not necessarily context-free, as they also support field dependencies (Section 2.2.4) and stream transformations (Section 2.2.5).

2.2.4 Field dependencies and variables

Messages of many protocols are data-dependent; that is, they use values of certain fields to determine, for example, the length of or choice between other fields. The *Extras*, *Key* and *Value* fields of the Memcached binary protocol described in Section 1.2, whose lengths are encoded in other message header fields, are one such example.

To support such protocols, message fields can have attributes (for example, a length attribute) that reference the value of other fields in the message. Depending on the kind of attribute, these references can have implications for both parsing and serializing of the message. The field referenced by a length attribute, for example, has to be parsed first to then parse the referencing field. Likewise, this fields value has to be updated during serialization to correspond to the (potentially updated) value of the referencing field or attribute.

Diffingo further supports computed fields, called *variables*, inside a unit. Variables serve to support computation during parsing or serialization. For example, in the Memcached

protocol, the length of the *Value* field can only be obtained by subtracting the values of multiple different fields in the message header. Here, a variable can be used to compute the value during parsing and update the header field values accordingly.

2.2.5 Value/stream transformations

The internal representation of field values should be easily accessible by the application and as close to the application semantics as possible. For example, an integer value that is encoded as decimal ASCII value in the wire-format is better presented to the application in the integer type of the application's underlying language. Similarly, integer-encoded enumeration values, such as status codes, can be better represented in the language's enum types or the encoding of string fields may need to be converted.

For this purpose, fields can have user- or system-provided *transformations* associated with them, which can perform such conversions to and from the internal representation during parsing and serializing respectively. Transformations consist of two functions, one for parsing and one for serializing. They are not only limited to value conversions, but can also be provided with access to the input (respectively output) streams to perform more complex computations, for example, to calculate a checksum, access values at stream offsets, or implement compression.

2.2.6 Incremental parsing and serializing

Application messages sent over the network often arrive at end-hosts in a piece-meal fashion: They are typically split across multiple network packets and read from a socket incrementally. Similarly, they are typically sent in multiple packets incrementally as well. Diffingo's generated parsers and serializers can pause and resume the processing of a message to accommodate for this fact.

This also allows the parsers and serializers to process messages that are embedded within a larger lower-layer protocol message incrementally. Consider again the example that Hadoop result messages are embedded within HTTP messages. In this case, the parser can parse up the HTTP message until it can emit the first result message and then pause until it is asked for the following result message. Likewise, the serializer can fill an HTTP message piece-by-piece with result messages as they are added to it.

3 Message format specification language

To describe protocol message formats and their instantiations for specific applications, Diffingo uses a domain specific language. To facilitate a fast prototype implementation, we adopt large parts of the corresponding language used in the Spicy parser generator [46], port them to a C++ execution environment, and extend the language with features for serializing, transforms, and instantiations.

In this chapter, we describe the elements of this language. We also provide examples for message specifications for the Memcached binary protocol (Listing 3.5) and a simplified version of HTTP (Listing 3.6). We first describe the different language elements individually (Sections 3.1 to 3.5) and then explain their use in the examples (Section 3.6). Table 3.1 provides an overview of the basic language elements.

3.1 Modules

A Diffingo specification file contains a list of declarations. To facilitate their reuse and combination, these declarations are defined within the namespace of a module. For simplicity, each file only contains declarations that belong to a single module. The name of this module is declared at the beginning of the file, as shown on line 1 in Listing 3.1. To use declarations that are part of a different module and were defined elsewhere, other modules can be included with the help of an import statement.

Modules contain declarations of types, such as units or enumerations, constants, functions, transformations, and unit instantiations. Lines 5-7 in Listing 3.1 show a unit type declaration. Declarations can be made visible to other modules by specifying the keyword `export`. We discuss the different kinds of declarations and their characteristics in the following sections.

In addition, modules can also define the (default) value of system-provided properties for entities within the module. An example for such a property is the byte-order of integer values in the wire-format of a message, which is set on line 3 in Listing 3.1.

```
1 module Memcached;
2
3 %byteorder = ByteOrder::big;
4
5 export type MemcachedCommand = unit {
6     # [...]
7 };
```

Listing 3.1: Example of a Diffingo specification file for a module named `Memcached`. We show the module declaration, as well as a property definition and type declaration.

Table 3.1: Basic elements of the Diffingo specification language.

Element	Example	Description
Module declaration	`module Memcached;`	Specifies the module that declarations within the file belong to.
Import statement	`import Memcached;`	Makes the declarations contained in the specified module accessible from within the current module.
Property definition	`%byteorder = ByteOrder::big;`	Defines the value of a system-provided property.
Constant declaration	`const NewLine = /\r?\n/;`	Declares a new constant with the specified value.

Function declaration	`bool success(status: uint64) %{`
	`  return status >= 200 &&`
	`  status <= 300;`
	`%};`
		Declares a function that can be called from within expressions. The body of the function contains C++ code, parameters are passed by reference.

Transformation decl.	`transform int32_big_endian`
	`  <stream, int32> {`
	`  ParseResult decode(`
	`  input: stream,`
	`  value: int32) %{ ... %};`
	`  SerializeResult encode(`
	`  output: stream,`
	`  value: int32) %{ ... %};`
	`};`
		Declares a transformation consisting of functions for decoding and encoding values. The transformation is specified together with the types that it operates on or converts between.

Type declaration	`type OpCode = enum { ... }`	Declares an alias name for a specific type, such as an enum or unit.

Unit type def.	`unit {`
	`  opcode : uint8`
	`  &transform_to(OpCode);`
	`  key_len : uint16;`
	`}`
		Defines the format of a unit as a composition of fields and their attributes; can also define variables and properties (see Section 3.2).

Enum type def.	`enum {`
	`  REQUEST = 0x80,`
	`  RESPONSE = 0x81`
	`}`
		Defines an enumeration type as a collection of labels and (optionally) their values.

Bit-field type def.	`bitfield(8) {`
	`  system : 0;`
	`  id : 1..7;`
	`}`
		Defines an a bit-field type with a given width as a set of labels and their bit ranges.

Expression	`self.key_len + self.extras_len`	Specifies a reference to an identifier or a computation. Expressions are used, for example, within attribute or property definitions (see Section 3.5).

3.2 Types and Units

Type declarations can be used to define aliases to type definitions, that is, to give a name to an atomic type, unit, enumeration, bit-field, or container.

```
1  type LengthValue = uint16;
2
3  type OpCode = enum {
4      GET = 0x00,
5      SET = 0x01
6  };
7
8  # parses a binary representation of a command.
9  # here, we use a parameter to determine if the command also contains a value.
10 type Command = unit(has_value: bool) {
11     opcode  : uint8 &transform_to(OpCode);
12     key_len : LengthValue;
13     key     : bytes &length=self.key_len;
14     val_len : LengthValue                if has_value;
15     value   : bytes &length=self.val_len if has_value;
16 };
17
18 # parses a textual representation of a command, such as
19 # "GET key;" or "PUT key value;".
20 # here, we use the opcode field to determine if there is a value.
21 type TextCommand = unit {
22     opcode  : /[^ ]*/;
23             : / / &transient;  # eat whitespace
24     key     : /[^ ;]+/;
25             : / / &transient if self.opcode == "SET";
26     value   : /[^ ;]*/       if self.opcode == "SET";
27             : /;/ &transient;  # eat semicolon
28 };
```

Listing 3.2: Examples for type declarations and units.

Atomic types. Diffingo supports a range of atomic types for Booleans, variable-width integers, strings, and byte arrays, which can later be extended to include, for example, floating-point numbers. We show a type declaration for an atomic type in line 1 of Listing 3.2.

Units. As described before, units define a composed message type, which can be parsed and serialized. We show examples for unit declarations in Listing 3.2 and language elements used to define units in Table 3.2. A unit lists the fields of the message (or of a part of a message) in the order in which they occur on-the-wire. Units are composable themselves as well: They can occur as type of a field inside another unit. Furthermore, units can be parameterized (line 10 in Listing 3.2) and can also contain variable definitions, whose value is calculated during parsing and/or may be used during serializing, and property definitions, which assign values for system-provided properties, such as the byte-order property mentioned above. Thereby, a value of a property defined inside a

Table 3.2: Language elements used to define units.

Element	Example	Description
Unit field decl.	`key_len : uint16;`	Declares a typed unit field; can also define attributes or a condition. Field name can be left out for anonymous fields.
	`padding : b"\x00";`	As above, but using a constant field value specified as a byte string.
	`number  : /[0-9]+/;`	As above, but using a regular expression to define the token that will be parsed into the field.
	`headers : list<Header>;`	As above, but using a list of elements of a certain type (here, a `Header` unit); can also specify a `&length` attribute.

Choice	`switch (self.opcode) {`
	`  OpCode::GET -> a : int16;`
	`  OpCode::SET -> b : int32;`
	`};`
		Choice between two or more fields or field sequences depending on value of an expression.

	`switch {`
	`  a : /[a]+/;`
	`  b : /[b]+/;`
	`};`
		As above, but depending on the value of a look-ahead symbol.

Unit variable decl.	`var value_len : uint32` `  &parse=... &serialize=...;`	Declares a variable used during parsing and/or serializing; value will also be accessible to application.

Unit instantiation	`instantiate unit Msg`
	`  as CompactMsg {`
	`  opcode;`
	`};`
		Declares and defines an instantiation of a unit with a given set of accessible fields (as scoped paths relative to the unit).

Attribute def.	`&length = self.value_len`	Defines the value of an attribute of a field, such as the length of a field with a variable-length type.

unit overrides any value that was set in the module it is contained within. We explain unit fields and variables in more detail in the following subsection. Internally, units are represented as a struct type.

Enumerations. Enumerations can provide a way to define the meaning of different value of a unit field. For example, the value of an integer-encoded unit field can be transformed to an enum value to make it easier accessible in the application (line 11 in Listing 3.2). Enumerations are defined by listing the labels together with their (integer) values. Labels can also be given without values—in this case, they are assigned by the system. This can be useful if the enumeration is used internally in the module and/or does not correspond to a value of a unit field. Enumerations are represented by an enum type internally. All enumerations are automatically supplied with a default label `UNDEF` that indicates that the value of an enum field or variable is undefined.

Bit-fields. A common paradigm in binary protocols is to encode flags or other pieces of information in certain bits of a larger constant-length field. Diffingo captures this

paradigm with so called bit-field types. Bit-fields have a statically defined width (number of bits) and define labels and bit ranges for different fields within the bit-field. An example is shown in Table 3.1. Bit-fields are represented by a struct type internally.

Containers. Diffingo currently only supports a single kind of container type: an indexable list of elements. Lists can be used inside units to define that a number of elements of a certain type are contained within it. During parsing, the list is filled until a defined length or a look-ahead symbol indicating the end of the list is reached; during serializing, the elements are written to the output one-by-one. We show examples for this in the following subsection, as well as in the HTTP specification in Section 3.6. Internally, lists are represented with an integer length value and a sequence of the element type.

3.2.1 Unit Fields and Variables

Units contain a list of typed fields and variables. Fields describe the wire-format layout of the unit; variables can be used to convert field or variable values or perform other arithmetic or conditional operations on field/variable values during parsing or serializing. During parsing, variables are evaluated in the order that they occur within the unit definition interleaved with the parsing of fields. That is, a variable defined after a field is evaluated after the field is parsed, while a variable defined before a field is evaluated before the field is parsed. During serializing, variables are evaluated in reverse order and before any serialization of field values occurs (but after dependent variable/field updates were performed, such as updates of `&length` attribute variables). The behaviour of variables and fields can be influenced by specifying field attributes, such as `&length` or `&transient` on lines 13 resp. 23 in Listing 3.2. We show the most common attributes and their meaning in Table 3.3. Attributes can be defined with a value given as an expression (see Section 3.5) either in brackets or in the form of an assignment.

Names. While variables need to be named, fields can also be unnamed. In this case, they become anonymous fields and are not made accessible to the application. A typical use case for this are constant fields, which we describe below. Often, anonymous fields are transient fields (`&transient` attribute), that is, their value is not stored between parsing and serializing. If a transient field is serialized, a default value (for example, a constant) will be used instead.

Types. Both fields and variables are typed. For fields, the type specifies the wire-format of the field (that is, the format of its token), as well as the type of the internal representation. For example, a field of type `int16` will be a 16-bit long binary field on-the-wire and an 16-bit width integer in the internal format (line 12 in Listing 3.2). In general, however, the wire-format and internal format do not need to be exactly the same (consider, for example, a different byte order in the host). For variables, the type specifies only the internal format, as variables do not have a on-the-wire representation.

In some cases, it may make sense to convert a field's value into an internal type other than the default internal type corresponding to the field's wire-format type. For

Table 3.3: Attributes of unit field and variables with their semantics.

Attribute name / example	Description
`&length = 8`	Specify the length of a variable-length field, such as a byte array field or a list.
`&byteorder = ByteOrder::big`	Specify the byte-order that should be used to parse an integer field; overrides any default byte-order specified elsewhere.
`&transform = decimal_int32`	Specify a transformation that will be applied to a field or variable after parsing and before serializing.
`&transform_to(OpCode)`	Specify that the value of a field or variable should be converted to another type, such as an enumeration. The generator will determine which transform to apply for this purpose.
`&eod`	Specify that a field should be parsed until an end of data event is encountered (for example, a closed connection).
`&chunked`	Specify that a field should be made accessible to embedded parsers in a piece-meal fashion as data arrives.
`&transient`	Specify that a field or variable is only used during parsing and serializing, but not accessible to the application. A transient field's value is set to a default value during serializing.
`&parseUntil($$.length == 0)`	Specify that a field containing a list of elements should be parsed until a given condition holds.
`&parseUntilIncluding($$.length == 0)`	As above, but including the element where the condition holds.
`&parse = self.total_len - self.key_len`	Specify how a variable's value is calculated during parsing.
`&serialize = self.total_len = $$ + self.key_len`	Specify how a variable's value is used to update other fields during serializing.

this purpose, a transformation can be specified with the `&transform` or `&transform_to` attributes. We describe transforms in more detail in Section 3.3.

Constant and regular expression fields. Fields can not only be defined by a type, but also alternatively with a constant value or regular expression, as shown in Table 3.2. A constant value field specifies that a constant token is expected in the wire-format. During parsing, existence of this token will be ensured and during serializing, the token will be generated. Often, constant fields are also transient fields, as mentioned above. The type of the constant value determines the type of the field.

Fields specified by a regular expression have a byte array type. During parsing, the regular expression is evaluated on the next bytes on the input stream. The byte sequence starting at the current position of the input stream that matches the regular expression becomes the value of the field. Our implementation of Diffingo advances the input stream for as long as the regular expression continues to match (or may eventually match). That is, we do not try to find the longest possible match, but the first match extended to its maximum length. We also constrain regular expressions not to use backwards references and some other complex operations. During serializing, the value of the field is serialized as a byte array. If the field is transient, we try to determine a default value that fulfills the requirements of the regular expression during generation-time.

Sequential composition. The default method for composition of fields is a sequence. The fields inside a unit are specified in the order in which they occur in the wire-format. During parsing and serializing, they are considered in sequence (with empty delimiters). Internally, sequences are represented as fields of a struct type.

Choices. Diffingo also allows to specify a choice between two or more alternative sequences of fields. For this purpose, it provides a switch statement (shown in Table 3.2). The choice can be made either depending on the value of an expression, for example, the value of another field or variable, or depending on the value of a look-ahead symbol. As described in Chapter 2, Diffingo supports a one-token look-ahead. In the case of a look-ahead switch, it will consider the first fields in the switch choices and determine which field's token matches the longest byte sequence starting at the current stream position and choose its switch branch to continue parsing. Internally, choices are represented with an enum that describes which branch is active and a union type of the different choice field sequences.

Lists. As mentioned before, units can also contain a list of elements of a specific type. These lists can contain either a specific number of elements (determined by an expression given with the `&length` attribute) or be terminated by a look-ahead symbol. In the latter case, Diffingo will compare whether the next token on the wire matches either the token of the list element's first field or the token of the field following the list in the unit before parsing each element of the list to determine if the end of the list is reached. In either case, a list will be serialized as if it were a fixed-size sequence of fields of the same type. Internally, lists are represented with an integer length value and a sequence of the element type.

Sinks. Some protocols allow the body of a message to be split across multiple message parts, such as is the case for the chunk-encoding of an HTTP message. For this purpose, Diffingo provides a data sink paradigm, that allows recombining the different parts into a single byte stream during parsing. We provide an example for a use of sinks in the HTTP specification in Section 3.6.

Fields with a byte array type can stream their value into a sink variable by specifying the variable's name after an arrow (`->`). With this, they become a source of the sink. When the field is parsed, its contents are added to the sink output, which is also of byte array type internally. During serializing, a sink outputs into a single field until the serializer is advised otherwise by the environment.

3.3 Functions and Transformations

Diffingo's specification language provides ways for developers to extend the functionality of the generated parser and serializer with custom behaviour. Developers can specify functions, which can be called from within expressions and perform arbitrary C++ computation, and transformations, which can convert values or implement more complex computation.

```
# operating on parsed integer value
transform int32_big_endian <int32, int32> {
  ParseResult decode(input: int32, value: int32) %{
    value = ntohl(input);
    return ParseResult::DONE;
  %};

  SerializeResult encode(output: int32, value: int32) %{
    output = htonl(value);
    return SerializeResult::DONE;
  %};
}

# operating on input and output stream directly
transform int32_big_endian <stream, int32> {
  ParseResult decode(input: stream, value: int32) %{
    if (input.remaining() < sizeof(value))
      return ParseResult::OUT_OF_DATA;
    value = *reinterpret_cast<int32*>(input.pos);
    value = ntohl(value);
    input.pos += sizeof(value);
    return ParseResult::DONE;
  %};

  SerializeResult encode(output: stream, value: int32) %{
    if (output.remaining() < sizeof(value))
      return ParseResult::OUTPUT_FULL;
    *reinterpret_cast<int32*>(output.pos) = htonl(value);
    output.pos += sizeof(value);
    return SerializeResult::DONE;
  %};
}
```

Listing 3.3: Examples for transformation declarations. Both have the same effect.

Functions. Functions are declared and defined with the `function` keyword. They can take a number of arguments, which are passed to the function by reference, and define a body of C++ code. Table 3.1 includes an example for a function declaration.

Transformations. Transformations consist of two functions: one used during parsing to decode a value, one during serializing to encode it. Transformations are declared together with the types of wire-format and internal values that they operate on. A simple transformation, for example, may operate on atomic types and simply convert between them. More complex computation on input and output streams, such as custom tokenizing, compression, checksum computation, or stream offset references, can be performed by declaring the wire-format type as `stream`. In this case, the encode/decode functions are provided with a reference to a stream object, which contains information about, for example, current stream position, token, and length. We show examples for an implementation of a byte-order converting transformation using either mechanism in Listing 3.3.

When a transformation is attached to a field, the system determines the correct way to invoke the decode/encode functions during generation-time. If it operates on parsed values, the system will first apply the parsing/serializing logic determined by the type of the field, otherwise it will apply the functions directly to the input/output streams. Transformations operating on streams cannot be attached to variables.

3.4 Instantiations

Units can be instantiated in the context of an application to define which variables and fields should be accessible to the application. Diffingo uses this information to optimize the generated parser and serializer code, as discussed in Chapter 4. In the code below, we show how the MemcachedCommand unit is instantiated for the exemplary router application we described in Section 1.2. It defines the name of the new unit instance as `CompactMemcachedCommand` and the accessible variables/fields as the `opcode` variable and `key` field.

```
module Memcached;

instantiate unit MemcachedCommand as CompactMemcachedCommand {
    opcode;
    key;
};
```

Defining accessible variables/fields. Accessible variables and fields in an instantiation are defined by providing the path to the field from within the instantiated unit. For fields or variables of that unit, this amounts to simply providing their name. For fields or variables of units contained within the instantiated unit, a path identifier is given instead. We show examples for this in Listing 3.4. Here, we instantiate a `Command` unit that contains a list of `Header` units and a `Body` unit. Once, we only make the `Body` unit's `data` field accessible, and once, we only make the `Header` units' `name` field accessible. To simplify the task, the language also provides two wildcard identifiers: a single star can be used to include all fields within a (contained) unit (for example, `body.*`), and a double star can be used to include all fields of a (contained) unit and its children (for example, `body.**`). To access values of list elements, a `[]` identifier is used. What is not shown in the examples, but also supported by the language, is the definition of accessible fields contained in recursive units by means of a star-operator that also specifies the type of a (recursive) unit.

Restricting list elements and field/variable values. It is further possible to specify that the application only wants to access certain elements of a list. For this purpose, a conditional expression can be given inside the list element identifier `[]`. Only those elements will be included, for which the condition evaluates to true. For example, the Command unit in Listing 3.4 could be instantiated as follows to restrict access to the value of a header with the name `"Type"`:

```
instantiate unit Command as TypeHeaderOnlyCommand {
   headers[self.name == "Type"].value
};
```

Similarly, the value of fields or variables can be constrained. This is useful to define that certain alternatives in the unit specification cannot be taken, for example, to disable certain complex features (such as the different delivery modes in the HTTP example in Section 3.6). To give a simple example, one could constrain that the length of the `Body` unit inside the `Command` unit is always 0 (see code below) to state that processed `Command` messages never have a body. Therefore, the generator can simplify the parsing logic to exclude the `data` field completely.

```
instantiate unit Command as EmptyBodyCommand {
    # [ ... ]
    constrain body.len == 0;
};
```

Embedding units. Finally, unit instantiations can be used to embed a unit into a field or variable of another unit (outer unit). This type of this field or variable has to be a byte array (internally). Diffingo assumes that the multiple objects of the embedded unit can occur as a list within this field. We show an example for embedding the `Command` unit inside an HTTP request below:

```
1  unit Header {
2      name    : /[^:\n]+/;
3              : /:/ &transient;   # eat colon
4      value   : /[^\n]+/;
5              : /\n/ &transient;  # eat newline
6  };
7
8  unit Body {
9      len     : /[0-9]+/ &transform = dec_int32;
10     data    : bytes &length = self.len;
11 }
12
13 unit Command {
14     headers : list<Header>;
15             : /\n/ &transient;  # eat newline
16     body    : Body;
17 };
18
19 instantiate unit Command as BodyDataOnlyCommand {
20     body.data;
21 };
22
23 instantiate unit Command as HeaderNamesOnlyCommand {
24     headers[].name;
25 };
```

Listing 3.4: Example for defining accessible variables/fields in an instantiation.

```
instantiate unit Command as HTTPEmbeddedCommand {
    **
} embed into HTTP::Message(body.contents) {
    **
};
```

The generated parser processes the outer unit until it can emit the first element of the embedded unit list. On repeated invocation, it continues processing the list of embedded units until the end of the outer unit's field value is reached. Parsed objects of the embedded unit contain a reference to the parsed object of the outer unit that they were contained in. The instantiation defines which fields in either unit object are accessible by the application.

During serializing, the serializer serializes embedded unit objects as elements into the outer unit's field, until it is advised to complete the serialization (that is, transmission) of the outer unit by the environment. The generated serializer provides two API's: one that allows on-the-fly serialization and transmission as new embedded units arrive and one that only serializes the outer unit when all embedded units have been serialized. The latter case may be needed if the field that the embedded units are contained in is length-encoded in the outer unit. In this case, the length of the field is only known once all embedded units were serialized.

3.5 Expressions

Diffingo's specification language also contains a small sub-language for specifying expressions. These expressions can be used, for example, in attribute or unit parameter definitions. The expression language contains elements for common arithmetic and comparison operations, accessing values of parameters, fields, and other (scoped) identifiers, constant values, regular expressions, conditionals, type casts, as well as lambda expressions, which can be used in a `find` operation. We summarize these elements in Table 3.4.

All expressions are designed in a way that the type of their result value can be statically determined. Diffingo needs to be able to resolve which fields are referenced by an expression during generation-time, so that it can determine the dependency relationships between fields needed for simplifying unit instantiations, as described in Chapter 4. For this purpose, we also provide a `find` operation, which allows Diffingo to reason about which elements of a collection are accessed by the expression.

3.6 Examples

3.6.1 Memcached binary protocol

We show the implementation of the Memcached binary protocol described in Section 1.2 in Listing 3.5. Here, we use enumeration types to capture the semantic value of the `magic_code`, `opcode` and `status` fields. A Memcached command itself is represented by the `MemcachedCommand` unit.

Table 3.4: Expression language elements (subset).

Expression	Example	Description
Identifier	`self`	Reference to current unit.
	`$$`	Reference to value of current field.
	`delivery_mode`	Reference to a unit parameter.
	`ByteOrder::big`	Reference to a scoped identifier declared in another module.
Constant	`1024`	Integer constant value (by default, a signed 64-bit integer).
	`uint16(1024)`	Integer constant value (here, unsigned 16-bit integer).
	`true`	Boolean constant value.
	`"abc"`	String constant value.
	`b"abc"`	Byte string/array constant value.
Regular expression	`/\r?\n/`	Construct a regular expression for parsing/serializing a field.
Member access	`self.total_len`	Reference to the value of a field or variable inside a unit.
Member update	`self.total_len = $$`	Update the value of a field or variable inside a unit.
Arithmetic operation	`self.a + self.b`	Perform an arithmetic operation with two operands, here addition.
Comparison op.	`self.a == self.b`	Perform a comparison operation with two operands, here testing for equality.
Conditional	`self.a ? 1 : 0`	Choose between two expressions depending on the outcome of a conditional expression.
	`if self.a then 1 else 0`	As above, alternative syntax.
Cast	`cast<uint8>($$)`	Cast a value to a different type.
Lambda expression	`h => h.content`	Specify a lambda expression consisting of an identifier and an expression where the identifier is bound to a value.

Find operation	`find(self.headers,`
	`  h => h.is_important,`
	`  h => h.content,`
	`  default_value)`
		Iterate through a list of values to find the first entry for which a condition holds and return the value of a lambda expression evaluated for it or a default value if no matching entry is found.

The layout of the MemcachedCommand unit is primarily just a sequence of the different fields in the command message with one exception: The 16-bit field that either contains a bucket ID or a status code is parsed differently depending on whether the command is a request or response. In the former case, the bucket ID is stored in the `vbucket_id` field (line 36). In the latter, the status code is converted into an enum value (via a `&transform_to` attribute) and stored in the `status` field (lines 37-38).

Similarly, the magic code value is converted to an enum in line 25. For the opcode field, we choose a different approach: The protocol specifies that the valid values of this field can be extended with new ones as is seen fit by the project. Therefore, we convert the value in a new unit variable instead (`var opcode` in line 27). We parse its value by casting the binary opcode value to the enum type and serialize it by casting back—but only if a valid enum value is specified; otherwise, the value set in the `opcode_binary` field is left untouched. (Here, we use the `$$` identifier to reference the current variable's value.) This way, it is still possible to parse and re-serialize messages with unknown opcodes.

To define the length of the value field, we also introduce a new variable (`var value_len` in line 45). During parsing, its value is calculated by subtracting the sum of the value of the `extras_len` and `key_len` field from that of the `total_len` field. Analogously, during serializing, the `total_len` field's value is updated accordingly.

One other interesting aspect to note is the usage of `&transient` in line 34. Here, we declare that the value of the anonymous field that models a reserved byte in the message should not be kept in the unit object between parsing and serializing.

3.6.2 HTTP

The specification of the messages exchanged in the hypertext transfer protocol (HTTP) [18] is more complex than the Memcached example described in the previous subsection. We show a simplified version, which supports content-length- and chunk-encoded HTTP messages in Listing 3.6.

On first sight, the main difference between the two examples is that HTTP is a textual protocol and thus most of the fields are specified by regular expressions rather than integer or byte array types. We define common regular expressions as constants at the beginning of the file (lines 5-15). To distinguish between the different delivery modes of the message body (either by end of data or content length, or in chunks) we use a variable (in the `Message` unit) of type `DeliveryMode`, which we declare in line 3.

We define HTTP requests and replies separately in the `Request` and `Reply` units, but use a common `Message` unit to capture the similarities between them (headers and bodies). A request consists of a request line (`RequestLine` unit) and a message, a reply of a reply line (`ReplyLine` unit) and a message. An interesting aspect of the `ReplyLine` unit's specification is the status field (line 44), which contains a string-encoded decimal integer. We convert its value to an internal integer type by applying the transform `decimal_uint32` (declaration omitted).

The `Message` unit consists of a list of `Header` units and a `Body` unit. An interesting aspect of the `Header` unit is the use of the uppercase function during parsing (line 58): We store the name of the header converted to uppercase in a separate variable. In the definition of the `Message` unit, we then use the `find` operation and these uppercase header names

to extract values from certain headers into unit variables (lines 65-78). This includes the content-length and transfer-encoding headers, whose values determine the delivery mode that is used for the message body. While we only extract the value of the transfer-encoding header, we extract and convert the value of the content-length header to an integer value. During serialization, we update the values of the corresponding headers.

We then set the value of the `delivery_mode` variable according to the extracted values of the headers (lines 85-90). If the transfer-encoding header is given and set to chunk-encoding, then the delivery will occur using chunks. Otherwise, the existence and value of the content-length header determines if the delivery occurs by length or until the end of input data is reached. The result of the delivery mode calculation is then passed on to the `Body` unit as a parameter.

This `Body` unit utilizes the sink paradigm we explained in Section 3.2. Depending on the delivery mode, different fields become sources of the `content` sink variable: In end-of-data delivery mode, it is a simple field of byte array type that parses until the end-of-data event is encountered (lines 97-98). In the content-length case, it is a similar field with a set length (lines 99-100). Lastly, in the case of chunk-encoding, a list of chunks is parsed as a field in the `Body` unit and their respective bodies become sources of the sink variable (line 101 and lines 107-120).

Similar to the Memcached specification, we mark many anonymous or constant fields in the HTTP specification as `&transient`, so that their value is not kept in the internal unit objects. We also use this attribute for the sources of the sink in the `Body`, `Chunk` and `Chunks` units, as their content is kept in the sink instead.

```
module Memcached;

export type MemcachedMagicCode = enum {
    REQUEST  = 0x80,
    RESPONSE = 0x81
};

export type MemcachedOpCode = enum {
    GET                 = 0x00,
    SET                 = 0x01,
    # [...]
    TAP_CHECKPOINT_END  = 0x47
};

export type MemcachedResponseStatus = enum {
    NO_ERROR                        = 0x0000,
    KEY_NOT_FOUND                   = 0x0001,
    # [...]
    TEMPORARY_FAILURE               = 0x0086
};

export type MemcachedCommand = unit {
    %byteorder = ByteOrder::big;

    magic_code          : uint8 &transform_to(MemcachedMagicCode);
    opcode_binary       : uint8;
    var opcode          : MemcachedOpCode
                            &parse = cast<MemcachedOpCode>(self.opcode_binary)
                            &serialize = if ($$ != MemcachedOpCode::UNDEF)
                                         then self.opcode_binary =
                                                cast<uint8>($$);
    key_len             : uint16;
    extras_len          : uint8;
                        : bytes &length = 1 &transient; # reserved field
    switch (self.magic_code) {
        MemcachedMagicCode::REQUEST  -> vbucket_id : uint16;
        MemcachedMagicCode::RESPONSE ->
          status        : uint16 &transform_to(MemcachedResponseStatus);
    };

    total_len           : uint32;
    opaque              : bytes &length = 4;
    cas                 : bytes &length = 8;

    var value_len       : uint32
                            &parse = self.total_len -
                                     (self.extras_len + self.key_len)
                            &serialize = self.total_len = self.key_len +
                                         self.extras_len + $$;

    extras              : bytes &length = self.extras_len;
    key                 : string &length = self.key_len;
    value               : bytes &length = self.value_len;
};
```

Listing 3.5: Diffingo format specification for binary Memcached messages.

```
module HTTP;

type DeliveryMode = enum { EndOfData, Length, Chunked };

const Token      = /[^ \t\r\n]+/;
const URI        = /[^ \t\r\n]+/;
const NewLine    = /\r?\n/;
const RestOfLine = /[^\r\n]*/;
const FullLine   = /[^\r\n]*\r?\n/;
const Integer    = /[0-9]+/;
const HexInteger = /[0-9a-zA-Z]+/;
const WhiteSpace = /[ \t]+/;
const OptionalWhiteSpace = /[ \t]*/;
const HeaderName  = /[^:\r\n]+/;
const HeaderValue = /[^\r\n]*/;

export type Request = unit {
    request: RequestLine;
    message: Message(True);
};

export type Reply = unit {
    reply:   ReplyLine;
    message: Message(False);
};

type Version = unit {
    :        /HTTP\// &transient;
    number:  /[0-9]+\.[0-9]*/;
};

type RequestLine = unit {
    method:  Token;
    :        WhiteSpace &transient;
    uri:     URI;
    :        WhiteSpace &transient;
    version: Version;
    :        NewLine;
};

type ReplyLine = unit {
    version: Version;
    :        WhiteSpace &transient;
    status:  Integer &transform(decimal_uint32);
    :        OptionalWhiteSpace &transient;
    reason:  RestOfLine;
    :        NewLine &transient;
};

type Header = unit(msg: Message) {
    name:    HeaderName;
    :        /:[\t ]*/ &transient;
    content: HeaderValue;
    :        NewLine &transient;

    # name_upper is only a variable for parsing,
    # it is not considered during serializing
    var name_upper: string &parse=uppercase(self.name);
};

```

Listing 3.6: Diffingo format specification for a simplified version of HTTP messages.

```
type Message = unit(is_request: bool) {
    headers:    list<Header(self)>;
    end_of_hdr: NewLine &transient;

    var content_length: uint64
      &transform(decimal_uint64)
      &parse = (find(self.headers, h => h.name_upper == "CONTENT-LENGTH",
                     h => h.content, None))
      &serialize = (find(self.headers, h => h.name_upper == "CONTENT-LENGTH",
                         h => h.content = $$, None));

    var transfer_encoding: string
      &parse = (find(self.headers,
                     h => h.name_upper == "TRANSFER-ENCODING",
                     h => uppercase(h.content), None))
      &serialize = (find(self.headers,
                         h => h.name_upper == "TRANSFER-ENCODING",
                         h => h.content = $$, None));

    var has_body: bool
      &parse = ((self.content_length && self.content_length > 0) ||
                (self.transfer_encoding));

    var is_request: bool = is_request;
    var delivery_mode: DeliveryMode
      &parse = if (self.transfer_encoding &&
                       self.transfer_encoding == "CHUNKED") then
                 DeliveryMode::Chunked
               else if (self.content_length && self.content_length > 0) then
                 DeliveryMode::Length else DeliveryMode::EndOfData;

    body:       Body(self, self.delivery_mode) if (self.has_body);
};

type Body = unit(msg: Message, delivery_mode: DeliveryMode) {
    switch (delivery_mode) {
      DeliveryMode::EndOfData -> : bytes &chunked &eod &transient
                                     -> self.content;
      DeliveryMode::Length    -> : bytes &chunked &length=msg.content_length
                                       &transient -> self.content;
      DeliveryMode::Chunked   -> : Chunks(self, msg);
    };

    var content: sink;
};

type Chunks = unit(body: Body, msg: Message) {
    chunks:    list<Chunk(body)> &parseUntil($$.length == 0) &transient;
    trailers:  list<Header(msg)>;
    :          NewLine &transient;
};

type Chunk = unit(body: Body) {
    length:    HexInteger &transform(hex_uint64);
    :          OptionalWhiteSpace &transient;
    extension: RestOfLine;
    :          NewLine &transient;
    :          bytes &chunked &length=self.length &transient -> body.content;
    :          NewLine if ( self.length != 0 ) &transient;
};
```

Listing 3.6: (continued).

4 Optimization mechanisms

Diffingo employs three techniques to ensure the efficiency of the generated parsing and serializing code. First, it avoids dynamic memory allocations (Section 4.1). Second, parsers and generators support incremental input and output (Section 4.2). Lastly, we use static analysis and unit instantiation definitions to optimize the layout of units and avoid processing unnecessary fields (Section 4.3).

4.1 Memory management

Diffingo generates parsers and serializers that do not allocate memory dynamically during parsing or serializing. Instead, the parsers and serializers use pre-allocated buffers to store the results of parsing and any data intermittently used during parsing or serializing. This avoids expensive system-calls in the performance-sensitive parts of the code.

We currently use fixed-size buffers and assume that a single message object can be contained within one such buffer. However, the architecture of the system would also allow us to extend this mechanism to better support messages that are highly variable in size. In such a case, it may make more sense to use a collection of buffers of different sizes and decide during parsing what size buffer to use. Alternatively, one could consider a more complex user-level memory management, where a message object is split across multiple buffers, for example, to place variable-size field values in separate buffers.

As one message object is stored within a single pre-allocated buffer, we need a mechanism to layout the different parts of the message object within the buffer. Our chosen mechanism is relatively simple: The fixed-size parts of the object are stored continuously at the beginning of the buffer. The values of any fields of dynamic size, such as elements of a list or variable-size byte-array fields, will be placed one after another after the message object in the buffer; pointers to these locations will then be stored within the message object. We discuss our implementation of this principle in Chapter 5.

4.2 Resumable parsers and serializers

As described in Section 2.2, parsing and serializing are commonly tasks that proceed incrementally. In particular, during parsing, input data becomes available piece-by-piece as new packets arrive. Oftentimes, generated parsers do not consider this fact. Instead, they signal a failure if they run out of data half-way through the parsing process and then retry parsing the message from the beginning when more data arrives.

A parser generated by Diffingo makes its state explicit, which allows it to save the parsing state and position when all available input data was processed. It then returns a status code signalling that it has only parsed a partial message. Upon its next invocation,

it will restore the saved parsing state and position and resume where it has left off. This way, no redundant computation is performed, which makes the parser more efficient if data arrives in pieces. This feature also makes it possible to integrate the parsing process with a user-level TCP stack, such as mTCP/DPDK [28, 26]. Such an integration could avoid copying data between the stack and parser by reading directly from the packet buffers in the stack. As we discuss in the following section, such an integration could be even more profitable, as the values of irrelevant fields would not need to be copied out of the packet buffers at all.

We also support a similar feature for the generated serializers: They are designed in a way that allows splitting the wire-format output of the serialization across multiple smaller memory buffers. The serializer saves its state when the output buffer is full and resumes where it left off when it is invoked again on another empty buffer. This way, the serializer could be integrated with a user-level TCP stack by writing into its packet buffers directly, thereby eliminating the need for copying the data between the stack and the serializer.

We discuss the implementation of resumable parsers and serializers in more detail in Chapter 5 and show their benefit in Chapter 6. We leave the integration of our generated parsers and serializers with a user-level TCP stack as topic for future work; however, another student is currently looking into the feasibility of such an approach with hand-rolled parsers. His initial findings suggest that there is good potential for such an optimization and that additional benefits can be gained by performing parts of the parsing within the network threads of the stack (and thereby avoiding context switches between the network and applications threads).

4.3 Optimizing message specifications

Diffingo leverages the fact that many (in-network) applications, such as the Memcached router, do not need access to all fields of messages objects during parsing and serializing: If the value of some field is not read or updated by the application, it is left out of the application-accessible portion of the message object. Furthermore, there is no need to parse such a field into its internal representation and then re-generate its wire-format during serialization later. If the message will not be re-serialized later, it is even possible to discard the value of these fields completely.

We also argue that it is possible to avoid copying the value of such an irrelevant field from the input data buffer of the parser into the internal representation of the message object. This can be of particular benefit if the parsing and serializing stages are integrated with a user-level TCP stack, as described above. In such a case, the parser can simply attach references to data in the input packet buffers to the message object and keep a reference count to these packet buffers. The serializer can copy the data from there into output packet buffers during serialization. When the message object is later destroyed, the reference counting releases the references to the corresponding input packet buffers, which can then eventually be freed (or, for efficiency, added back into a memory pool). Regardless of whether an integration with a user-level TCP stack is used, keeping (or copying) the wire-format value of an irrelevant field is not necessary if the message is

never re-serialized anyway. This could, for example, be the case in a monitoring-only application. In such a scenario, the parser can simply skip over the irrelevant fields.

In addition to the above optimization, we can further try to optimize the specification of a unit by utilizing the constraints specified in the instantiation, combining multiple irrelevant fields into a single field, or by skipping over multiple such fields at once. By optimizing the unit specification, Diffingo is then later able to produce more efficient parser and serializer code. We describe methods that implement such an optimization below. First, however, we explain how we determine the irrelevant parts of a message.

4.3.1 Field dependency analysis

We use unit instantiation specifications as a way to define which fields (and variables) of a message (unit) specification should be accessible to an application. In our current prototype implementation of Diffingo, we manually specify the instantiations. We are currently working on an integration of our prototype with the Flick language [1], a strongly typed programming language for in-network functions. In this integration, we will use static analysis of Flick programs to over-approximate which fields of a message are accessed by the application and to auto-generate corresponding instantiations. This is easily possible by traversing the abstract syntax tree of the program and noting down field accesses, as all field accesses occur on fully typed variables. Constrained accesses to fields (for example, to only some elements of a list) can be recorded by inspecting conditional statements and list operations.

In an optimal scenario, the fields specified in the unit instantiation are all the fields whose value will need to be considered during parsing and serialization. In reality, however, fields often have dependencies to other fields or variables. One example for this are length-encoded fields with variable size, where the size is defined in another field, such as the `value` field of a Memcached binary command. Variables also almost always depend on some other field or variable. In addition, the parser still needs to be able to determine the boundaries of fields during parsing, for example, to be able to determine if it has reached the end of a message. Thus, it needs to know the size of all variable-size fields during parsing, even if they are not required to parse application-relevant fields.

To determine the set of relevant fields, Diffingo first determines the dependencies between fields and variables before applying any of the described optimizations. Dependencies can exist between a unit and a field or variable, as well as between two fields, two variables, a field and variable or a variable and field. By computing these dependencies, Diffingo fills a dependency graph, which is then later used to determine all those fields and variables that are required for parsing or serializing the fields specified in the unit instantiation, or for determining field lengths.

We use four different types of dependencies to distinguish between the different scopes that a dependency can exist in. First, a field or variable's value may be needed to parse another field or variable (*parsing dependency*), for example, to make a decision between different alternatives in a `switch` statement or to determine the length of a variable-length field with a `&length` attribute. Second, a field's dynamic length may need to be determined by evaluating its regular expression on the input data to determine the length of the unit and offsets of following fields (*parsing-length dependency*). Third, a field or

variable may be needed to serialize another field or variable (*serializing dependency*), for example, because it is required to evaluate a `switch` statement's condition. And lastly, a field or variable may need to be updated during serializing by another field or variable (*serializing-update dependency*), such as a field referenced in a `&length` attribute. In addition to these dependency types, we add all fields and variables specified in the instantiation as *application dependencies*.

Diffingo computes these dependencies by statically analyzing the unit specification, its fields, and their attributes. Thereby, it also analyzes the expressions used in the attribute values and records references to other fields. `find` expressions are handled in a special way, recording only dependencies to those elements of the list for which the supplied find condition holds.

Using the resulting dependency graph, Diffingo then determines the dependency scopes in which each field is required in the context of a single unit instantiation by iteratively adding more dependencies, starting with the fields and units required by the application. Based on the resulting scopes associated with each field, the mechanisms described below will optimize the unit specification for this instantiation. Furthermore, the parser and serializer generation will take the scopes of a field into account. For example, the value of a field or variable whose only dependency scope is a *parsing dependency* does not need to be stored in the message object itself, but can rather be stored only during parsing in the parser's own state.

4.3.2 Approaches to optimizing a unit specification

We can utilize the field dependency scope information and constraints specified in the unit instantiation to optimize a unit specification in four primary ways: First, we can use constraints specified on variables and fields to try to remove alternatives from `switch` statements. Determining which values of a conditional expression of such a `switch` statement are viable amounts to specifying a new constraint over the conditional expression and the given constraint, which can then be given to a constraint solver for evaluation. Similarly, we can also try to determine the value of conditions specified with a field or variable and remove the field or condition if the value can be determined statically.

Second, we can use the constraints specified on list elements to determine if a specific list element needs to be parsed and made application-accessible at parsing-time. That is, we can parse the element up to a state where all fields and variables required to evaluate the constraint are available. If the constraint does not hold for the element, the remainder of the element does not need to be parsed. Otherwise, parsing of the element proceeds as required for the instantiation.

Third, we can combine subsequent fields that aren't required at all (that is, without dependency scopes) into a larger field by adding up their lengths. We describe this approach in Section 4.3.3.

Fourth, we can try to combine subsequent fields that are only required with a scope of a *parsing-length dependency* into a larger field by determining a regular expression of a delimiter that separates them from the following fields. We describe this approach in Section 4.3.4.

To discuss how we can combine irrelevant fields, it is useful to revisit the ways in which fields can be composed together in units: First, they can be composed sequentially one after another. Second, sequences of fields can be composed as alternatives of a switch statement, which can be embedded into another sequence of fields. Finally, a sequence of fields defined in a separate unit can be embedded into another sequence of fields either directly or as a list.

4.3.3 Combining length-encoded fields

In this subsection, we explain how Diffingo tries to combine multiple irrelevant fields that don't have any dependency scope into a single larger field. We consider the different ways of field composition individually to discuss how irrelevant fields composed in each a way can be combined with each other.

Sequence. Consider first a sequence of such irrelevant fields. Because they are irrelevant, their length is either statically fixed or determined by another field or variable's value, which has at least a dependency scope of a *parsing dependency*. This means that we can replace the sequence of fields by a single byte-array type field whose length is the sum of the lengths of all fields in the sequence. For fields with static lengths, this amounts to a statically determinable length of the resulting field. If any of the fields in the sequence has a variable length, then the sum will be the sum of the length expression of this field (given in attribute `&length`) and the sum of the lengths of the other fields. Because directly embedded units are just sequences of fields as well, we can handle them as if their field sequence was integrated into the field sequence of the containing unit.

We show an example for such field combinations below. We use a `&dependency_scope` attribute to show the dependency scope that was determined by the dependency analysis for each field. If no such attribute is given, the field has no scope associated with it and is thus irrelevant.

```
type A = unit {
    a : int16 &dependency_scope = "parsing";     # required for length of e
    b : int16;
    c : int16;
    d : int16 &dependency_scope = "application"; # required by application
    e : bytes &length = a;
    f : bytes &length = 4;
    g : G;
};

type G = unit {
    h : bytes &length = 8;
};
```

In this example, fields `b` and `c`, as well as fields `e`, `f`, and `h` (within the embedded unit `G`) can be combined together into larger fields with accumulated lengths. Field `a` is only required during parsing and could later also be combined into the resulting field of the combination of fields `b` and `c` as it does not need to be stored in the internal representation. The result of this optimization would be the simplified unit below.

```
type A = unit {
    a : int16 &dependency_scope = "parsing";
      : bytes &length = 4;
    d : int16 &dependency_scope = "application";
      : bytes &length = a + 12;
};
```

Choice. Next, consider a `switch` statement with multiple different (viable) alternatives. If, after combination, all alternatives in the choice have a single irrelevant field with the same length (same static length or length expression), then the `switch` statement can be replaced by a single byte-array type field with that length. More generally, if any two or more alternatives have a single irrelevant field with the same length, they can be combined together as a byte-array type field in a single alternative with multiple possible conditional values. Furthermore, if the lengths of such irrelevant fields in two or more alternatives are not the same, they can still be combined into a single field where the length of the resulting byte-array type field is determined by a conditional expression over the value of the `switch` statement's condition and set to the length of either of the two or more original fields depending on the value of the condition.

Consider the following example:

```
type A = unit {
    a : int16 &dependency_scope = "parsing";  # required for switch
    switch (a) {
      0 -> b : bytes &length = 4;
      1 -> c : int32;
      2 -> d : int16;
      # default case is empty
    };
    e : bytes &length = 20;
};
```

In a first step, fields `b` and `c` can be combined into a field of the same length within an alternative for the case that `a` is either 0 or 1:

```
type A = unit {
    a : int16 &dependency_scope = "parsing";
    switch (a) {
      0,1 ->   : bytes &length = 4;
      2 ->   d : int16;
      # default case is empty
    };
    e : bytes &length = 20;
};
```

Next, both remaining alternatives (and the default alternative) can be combined into a single field with a length that is determined by a conditional expression over `a`. It can now also be combined with the irrelevant field `e` after the switch.

```
type A = unit {
    a : int16 &dependency_scope = "parsing";
      : bytes &length = ((a == 0 || a == 1) ? 4 : (a == 2 ? 2 : 0)) + 20;
};
```

Diffingo may even be able to perform some optimizations if (after combination) there are multiple fields in the sequences of the alternatives in the switch statement. In this case, it may be possible to move fields common to all alternatives at the beginning or end of their sequences outside the scope of the alternatives to combine them with other fields before or after the choice using a similar mechanism with conditional. However, the benefits of this optimizations are likely small.

List. Lastly, consider a unit embedded into a sequence as a list with a length given as expression in a `&length` attribute. If none of the fields of this embedded unit are relevant (that is, it consists of a single irrelevant field after compaction), then it can be reduced into a single byte-array type field with a length of the product of the unit's inner field's length and the length of the list.

In the example below, we can combine the list of `B` units with the field `c`. In the first step, we can combine fields `d` and `e` into a single field with a length of `8` bytes (not shown). Then, the list can be reduced in a field with a length of `a * 8` bytes (not shown). Finally, it can be combined with field `c` into a field with a resulting length of `a * 8 + 20`.

```
type A = unit {
    a : int16 &dependency_scope = "parsing";  # required for length of b
    b : list<B> &length = a;
    c : bytes &length = 20;
};

type B = unit {
    d : int32;
    e : int32;
}

# result:
type A = unit {
    a : int16 &dependency_scope = "parsing";
      : bytes &length = a * 8 + 20;
};
```

Notes and Limitations. In the above description of the algorithm, we did not discuss the occurrence of variables that are interleaved with irrelevant fields, recursive units, and stream transformations or sinks. In the case of interleaved variables which are not irrelevant, these variables can be easily moved before or after the sequence of irrelevant fields, as they do not depend on the values of these fields. That is, they do not prevent combining irrelevant fields that occur before and after them.

When considering units which are recursive (that is, they contain a field of their own unit type either in their own unit body or inside one of their child units), care has to be taken. If such units are embedded into a sequence of fields, they cannot simply be integrated into the sequence. Instead, they have to remain an extra unit. In our implementation, we track additional dependencies between units for this purpose. If we detect a cycle in these dependencies, the involved unit is handled specially.

Stream transformations and sinks do not prevent the optimization described so far. They have to be considered in a special way during the dependency analysis though,

since stream transformations may have arbitrary effect on the input and output stream and the value of sinks is difficult to re-serialize if its sources are declared `&transient`. Therefore, we currently define both fields with stream transformations, fields of sink type and fields with sink attributes as fields that are required by the application. For sinks, we can make an exception if there is only a single source defined for the sink. In this case, we consider the source as a non-transient field in either case. A more complex implementation may even support multi-source sinks in a better way.

4.3.4 Combining fields based on delimiters

In this subsection, we explain how Diffingo tries to combine multiple fields that only have a *parsing-length dependency* scope into a single larger field. These fields are fields whose regular expression needs to be applied to the input stream to determine the length of the field during parsing.

A simple way to combine a sequence of regular-expression fields is to replace them with a single field to which we apply the regular expressions of the original fields in sequence. Thereby, we can reduce the number of fields and thus simplify the unit specification, which can make the parser and serializer code, as well as the internal data structure representation of the message, more efficient.

However, we can even do a little better. Our mechanism also tries to exploit the following observation: Oftentimes, sequences of fields that are specified by regular expressions are delimited by a sequence of characters. If we can determine a simple delimiter character sequence for a sequence of parsing-length fields, we can replace the field sequence with a single field specified by a regular expression that over-approximates the original field sequence by searching for the delimiter characters. This way, the parser does not need to apply multiple regular expressions in sequence, which can further improve its performance.

To give an example, consider a sequence of parsing-length fields with the regular expressions `/[0-9]+/`, `/\./`, `/[0-9]+/`, and `/\n/`. Here, intuitively we can observe that the new-line character of the last regular expression acts as a delimiter that separates the sequence from the following fields, as it cannot occur within the previous fields' values in the field sequence. We can replace this field sequence with a single field with the following regular expression: `/[^\n]+\n/`. This sequence over-approximates the possible values of the first three fields: These can now contain any characters that are not a new-line character. Once a new-line character is found, the combined field's regular expression terminates. This way, the parsing code for the combined field can be more efficient. One should note, though, that it now also accepts inputs that are ill-formed according to the original specification. This is a trade-off between performance and validity, which is often made in performance-critical contexts.

To determine a delimiter character or character sequence for a composition (for example, a sequence) of parsing-length fields, we use a heuristical approach. First, we guess a set of potential delimiters; then we check if they can occur anywhere else in the composition apart from at the end. The latter is simple if we are testing for a single-character delimiter, as we only need to check if any of the regular expressions can produce the character (apart from at the end of the last regular expression). Performing this test for

a character sequence is a more complex task. For this purpose, we utilize an algorithm similar to generating follow-sets for LL(k) context-free grammars [4]. Follow-set calculation aims to generate a table of input tokens (here, single characters) and their k possible follower tokens (again, single characters) in the grammar. For a delimiter character sequence with x characters, we can compute the LL(x-1) follow set for a grammar specified by the regular expressions of the considered field composition and use the result to check if the sequence can be generated at any other location than at the end of the field composition. Other literature also generalizes the problem to determining sub-string containment in arbitrary context-free grammars [38] and LR(k) grammars [6]. To be able to create a simplified regular expression for a sequence of delimiter characters, we extend our support for regular expressions to include the negation of regular expressions, noted as `/(!regexp)/`.

Next, we again consider the different ways of field composition individually to discuss how parsing-length fields composed in each a way can be combined with each other and how we guess potential delimiters in each case.

Sequence. First, we consider a sequence of fields with only a *parsing-length dependency* scope. In this case, we can combine the fields as discussed above: We can combine them into a single field with either a list of regular expressions taken from the individual fields or a simplified regular expression. As in the case of length-encoded fields discussed earlier, we can integrate an embedded unit into the field sequence it is embedded into (as long as it is not a recursive unit).

We guess a delimiter by looking at the regular expression of the last field in the sequence and of the following field after the sequence. If the former ends with a sequence of characters, we consider each suffix of this sequence as a potential delimiter. Analogously, if the latter begins with a sequence of characters, we consider all prefixes of the sequence as a delimiter. If we find a delimiter in the former case, we simplify the regular expression of the resulting field into the form `/(!suffix)suffix/` (that is, consuming the delimiter). In the latter case, the resulting regular expression will be of the form `/(!prefix)/` (that is, not consuming the delimiter). If we are unsuccessful in determining a delimiter for a given sequence of fields, we can try to consider only a sub-sequence of this sequence instead: In some cases, there may be a delimiter that separates parts of the sequence from each other.

As we already described the combination for a field sequence whose delimiter is contained in the regular expression of the last field of the sequence, we show a sequence whose delimiter is contained in the regular expression of the field after the sequence in the following example:

```
type A = unit {
    a : /[0-9]+/           &dependency_scope = "parsing-length";
    b : /\./               &dependency_scope = "parsing-length";
    c : /[0-9]+/           &dependency_scope = "parsing-length";
    e : / Value is [^\n]+/ &dependency_scope = "application";
};
```

In this case, we determine that the prefixes of the regular expression of field `e` are potential delimiters. We can determine that the white-space character `" "` cannot be

contained in any of the parsing-length fields. Therefore, we can simplify the unit to obtain the following result:

```
type A = unit {
       : /[^ ]+/             &dependency_scope = "parsing-length";
    d : / Value is [^\n]+/ &dependency_scope = "application";
};
```

Choice. For sequences of parsing-length fields composed together through a `switch` statement, we can try to identify a common delimiter for all alternatives. Either, this could be a suffix of the regular expressions of the last fields in the different alternatives, or a prefix of the field following the `switch` statement. If the statement occurs within a sequence of parsing-length fields as well, we can try to extend our search for a delimiter to the joined sequence considering with all alternatives simultaneously. Then, a potential delimiter of this sequence must not occur in any of the sequences resulting from taking either alternative branch of the `switch` statement to be a valid delimiter. The follow-set approach to determine if a delimiter is valid (described above) supports such a check.

Consider the following example with a look-ahead-based `switch` statement with three alternatives, all with two fields each.

```
type A = unit {
    a : /[0-9]+/            &dependency_scope = "application";
    b : / /                 &dependency_scope = "parsing-length";
    switch {
      { c : /[a-d]+/        &dependency_scope = "parsing-length";
        d : /:/;            &dependency_scope = "parsing-length"; };
      { e : /[e-w]+/        &dependency_scope = "parsing-length";
        f : /:/;            &dependency_scope = "parsing-length"; };
      { g : /[x-z]+/        &dependency_scope = "parsing-length";
        h : /:/;            &dependency_scope = "parsing-length"; };
    };
    i : /[0-9]+/            &dependency_scope = "parsing-length";
    j : /\n/                &dependency_scope = "parsing-length";
};
```

We can replace the switch statement alone with a field with the regular expression `/[^:]+:/`, as the colon character functions as a delimiter of all cases in the switch statement. However, the switch statement also occurs inside a sequence of parsing-length fields delimited by a newline character. Therefore, a more efficient way to combine the fields is shown in the result below.

```
type A = unit {
    a : /[0-9]+/            &dependency_scope = "application";
      : /[^\n]+\n/          &dependency_scope = "parsing-length";
};
```

Similar to the way we handle compacting length-encoded fields in `switch` statements, we can also extract common parsing-length scoped fields at the beginning or end of all alternatives out of the statement.

List. Consider the case that a unit is embedded as a list with look-ahead (that is, without given length). If all fields of this unit are parsing-length dependencies, we can not only combine them inside the unit, but also try to combine them with any parsing-length fields surrounding the list in the containing unit. This is similar to the way that we can combine a `switch` statement with surrounding fields.

However, lists with look-ahead are also often delimited from the following fields to signal the end of the list. So if it is not possible to determine a delimiter for the whole parsing-length field sequence in which the list occurs, it is likely that we can guess a delimiter to skip over the list by using prefixes of the regular expression of the field following the list. Sometimes, however, the delimiter between the list and the following fields is the same delimiter as the one used to separate entries in the list and is used twice in sequence to signal the end of the list. In this case, we consider both the last field of the unit in the list and the following field together to guess potential delimiters.

One example for this is shown below. Here, a list of `B` units, which are delimited from each other by a new-line is embedded into an `A` unit.

```
type A = unit {
    a : /[0-9]+/             &dependency_scope = "application";
    b : list<B>              &dependency_scope = "parsing-length";
    c : /\n/                 &dependency_scope = "parsing-length";
    d : /[0-9]+/             &dependency_scope = "parsing-length";
    e : /[a-z]+/             &dependency_scope = "application";
};

type B = unit {
    f : /[a-z]+/             &dependency_scope = "parsing-length";
    g : /\n/                 &dependency_scope = "parsing-length";
}
```

In this example, we can see that we cannot combine fields `b`, `c`, and `d` all together, as the regular expression of field `d` cannot be used to guess a delimiter. Furthermore, if we only consider fields `b` and `c`, the new-line character guessed from field `c`'s regular expression is not a delimiter either, as it can occur inside the list in field `g`. We could combine only the fields `f` and `g` in unit `B` using a regular expression `/[^\n]+\n/`. Alternatively, we can use information both from fields `g` and `c` to guess that a sequence of two new-line characters delimits the sequence of fields `b` and `c` if there is at least one element in the list. We can use this to generate a choice between two options and replace the list with a look-ahead switch statement instead. The result is shown below.

```
type A = unit {
    a : /[0-9]+/             &dependency_scope = "application";
    switch {
      { : /\n/               &dependency_scope = "parsing-length"; };
      { : /[^\n](!\n\n)\n\n/ &dependency_scope = "parsing-length"; };
    }
    d : /[0-9]+/             &dependency_scope = "parsing-length";
    e : /[a-z]+/             &dependency_scope = "application";
};
```

Utilizing constraints. We are currently working on an extension of this mechanism for combining parsing-length fields that leverages information given in list element and field constraints. Here, we describe the motivation of this extension based on an example for a list. Therefore, consider the following unit specifications:

```
type A = unit {
    a : /[0-9]+/              &dependency_scope = "application";
    b : list<B>               &dependency_scope = "application";
    c : /\n/                  &dependency_scope = "parsing-length";
    e : /[a-z]+/              &dependency_scope = "application";
};

type B = unit {
    f : /[a-z]+/              &dependency_scope = "parsing-length,parsing";
    g : /=/                   &dependency_scope = "parsing-length";
    h : /[0-9]+/              &dependency_scope = "application";
    i : /\n/                  &dependency_scope = "parsing-length";
}
```

Further, consider that the field `h` is only required for elements of the list of `B` units, where the constraint `self.f == "abc"` holds (which was given in the instantiation of unit `A`). In this case, we can try to use the combination of fields `f` with value `"abc"` and `g` as a delimiter to find the value of field `h` for elements of the list for which `f` is the correct value. The result of such a combination is shown below. It uses a `switch` statement inside the body of the `B` unit, which combines together the double new-line delimiter of the list, as well as an `"abc="` delimiter to find relevant list elements. To allow for multiple such elements in the list, the unit is made recursive. We currently believe that such an optimization may make the parsing process even more efficient if there are many elements in the list. However, to make the result more convenient to access by the application, it would be better to store the results in the form of the original list of elements of the `B` unit, instead of a recursive structure as shown here. We leave a generalization of this extension as future work. We also envision an analogous extension that may leverage field constraints specified in the instantiation.

```
type A = unit {
    a : /[0-9]+/                 &dependency_scope = "application";
    b : B                        &dependency_scope = "application";
    e : /[a-z]+/                 &dependency_scope = "application";
};

type B = unit {
    switch {
      {   : /\n/                 &dependency_scope = "parsing-length"; };
      {   : /(!abc=)abc=/        &dependency_scope = "parsing-length";
        h : /[0-9]+/             &dependency_scope = "application";
          : /\n/                 &dependency_scope = "parsing-length";
          : B                    &dependency_scope = "application"; };
      {   : /[^\n](!\n\n)\n\n/   &dependency_scope = "parsing-length"; };
    };
};
```

Notes and Limitations. In the above description of the mechanism for combining delimiter-encoded fields, we did not explicitly discuss the effect of variables that are interleaved with fields that only have a *parsing-length dependency* scope. As in the case for the combination of length-encoded fields, they can be moved before or after the sequence of parsing-length fields, as they are not dependent on their value.

We also did not mention how constant value fields can be integrated into the mechanism. For example, a field with a constant string or byte array value can be considered and utilized in the mechanism as if it was a regular-expression field with a constant regular expression, too.

Furthermore, our current mechanism only considers constant character sequences as delimiters. It may be possible to extend our mechanism to a more general description of delimiters (for example, we could see a benefit in supporting delimiters such as `/\r?\n/`, where parts of the delimiter character sequence are optional). This, however, requires a change to the way we determine whether the delimiter is valid for a sequence of fields.

As in the previous subsection, we also did not discuss the effect of fields with stream transformations, recursive units, and sinks on the mechanism. Sinks are currently not supported for regular expression fields, therefore, we do not have to consider them. Recursive units are supported by the delimiter validity check based on follow-set calculation. Thus, if no fields within a recursive unit are required, it is possible to reduce it to a single field with a regular expression if a delimiter can be found. Fields with stream transformations are considered in the dependency analysis as described in the previous subsection.

Yet another thing to note is that the mechanisms for combining length-encoded and delimiter-encoded fields may need to be applied together to a specific unit, if there are fields of different kinds (both length- and delimiter-encoded) present in the unit. One way to do this is to define which method may be used for each field in the specification. Interestingly, for some fields, both mechanisms are possible. For example, constant value fields have a static length, but could also be considered as constant-value regular expression fields. Employing a heuristic to determine which choice is sensible depending on the types of the surrounding fields is one solution for this issue.

5 Implementation of the Diffingo prototype

We implemented a prototype of the Diffingo parser/serializer generator framework in C++. It's architecture and implementation is roughly based on that of the Spicy parser generator [46]; but while Spicy generates parsers in the HILTI intermediate language for network applications [46], our prototype generates C++ code to make integration with existing software easier. We also add more extensive support for serializing, as well as a prototype implementation of the optimization mechanisms described in Chapter 4. In this chapter, we describe the architecture of our prototype and aspects of its implementation, as well as the design and implementation of the generated parsers and serializers.

5.1 Generator architecture

The architecture of the Diffingo generator implementation is shown in Figure 5.1. As previously described in Chapter 2, it takes as inputs a message specification and an instantiation configuration. In our prototype, these two inputs are specified in the language described in Chapter 3 and provided in one or more files, which are combined by the module and import mechanisms of the language.

The generator's output are three entities: a set of data structures (C++ classes) that describe the internal representation of the messages, a parser class that provides an API to parse a message of the instantiated unit, and a serializer class that provides an analogous API to serialize a message. These entities are then printed into a C++ header and implementation file, which can be used by the target application.

Diffingo processes the inputs in three primary stages to generate the output entities: First, the specification files are parsed into a combined abstract syntax tree (AST). The tree is then fed through a number of preprocessing passes in the second stage. These passes resolve dependencies between elements in the AST and implement the specification optimization mechanisms described in the previous chapter. Their output is an optimized version of the AST. Finally, a code generation stage implements the task of transforming

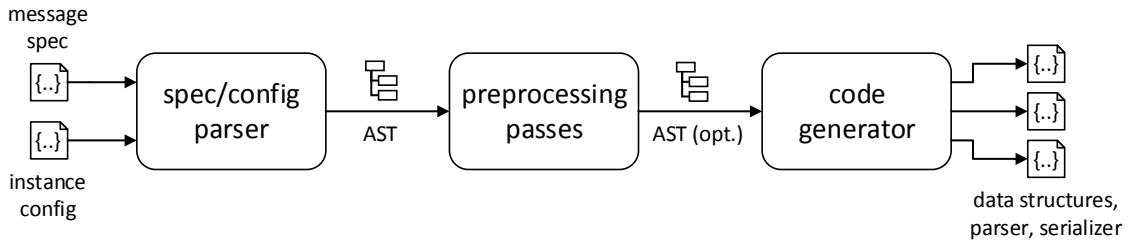

Figure 5.1: Diffingo generator implementation architecture.

the optimized unit specification into the output entities. In the following, we describe each stage in more detail.

5.2 Parsing specifications into an AST

To parse the specification files into an abstract object representation, we use the Bison parser generator. It is able to generate an LALR parser from a grammar file describing Diffingo's specification language. We could reuse large parts of the grammar file that is used in the implementation of Spicy by removing unnecessary parts, such as events and hooks, and extending it with Diffingo-specific language elements, such as transformations, embedded C++ function bodies, and lambda/find expressions.

The parser creates an object-oriented representation of the declarations in each in module (Figure 5.2). Modules are associated with their name, as well as the properties and declarations contained within them. The different types of declarations are reflected in the class hierarchy of the `Declaration` class. Functions, transformations, types, and instantiations are represented by individual classes, which store associations to the respective properties of the entities, such as function arguments and return types, transform functions, and instantiation fields. Constant declarations are represented by the expressions that define them. We do not show the classes used to represent expressions or constant values of different types.

Different types are represented by different subclasses of the `Type` class. Unit specifications are translated into an instance of the `Unit` type class. Its model is shown in Figure 5.3. A unit is associated with the fields, variables, and properties defined within it, as well as any parameters passed to it. Again, the different types of fields are represented by different subclasses of the `Field` class. Switch statements and lists are represented as separate types of fields. The unit model can be easily extended to support additional types of fields, either by adding new atomic types, that is, types whose wire-format length is known during parsing before starting to parse the field, or by adding new subclasses of the `Field` class. Each of these subclasses defines a new (custom) behavior for the parsing and serializing logic associated with the field. Adding such a new subclass, however, will need to be considered both in the generation of data types, parsers, and serializers, and in the preprocessing stage, for example, when optimizing unit specifications.

To make processing the information contained in the AST convenient, we apply the visitor pattern: All elements in the AST derive from a `Node` class. Nodes store references to their children, which allows a generic implementation of an abstract `Visitor` class with methods to traverse all elements in the AST in pre- or post-order. An in-order traversal can also be realized manually when needed. Furthermore, all references between nodes are kept via a special pointer implementation that allows visitors to replace an element in the AST by updating all references to it from anywhere in the AST to point to its replacement element. Amongst other things, this makes it easy to resolve textual references to other elements in the first passes of preprocessing.

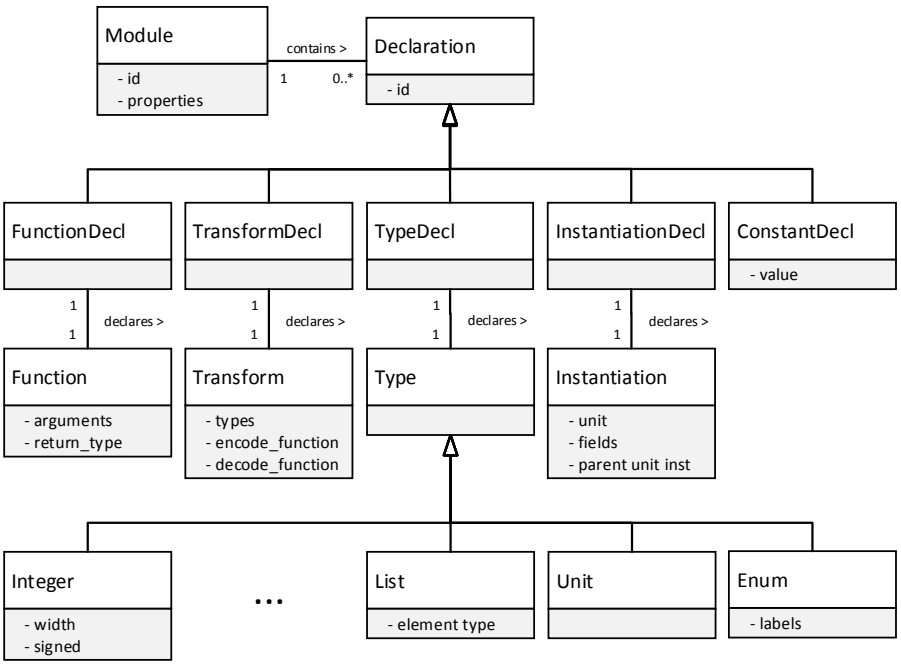

Figure 5.2: AST classes for declarations within a module (simplified, excerpt).

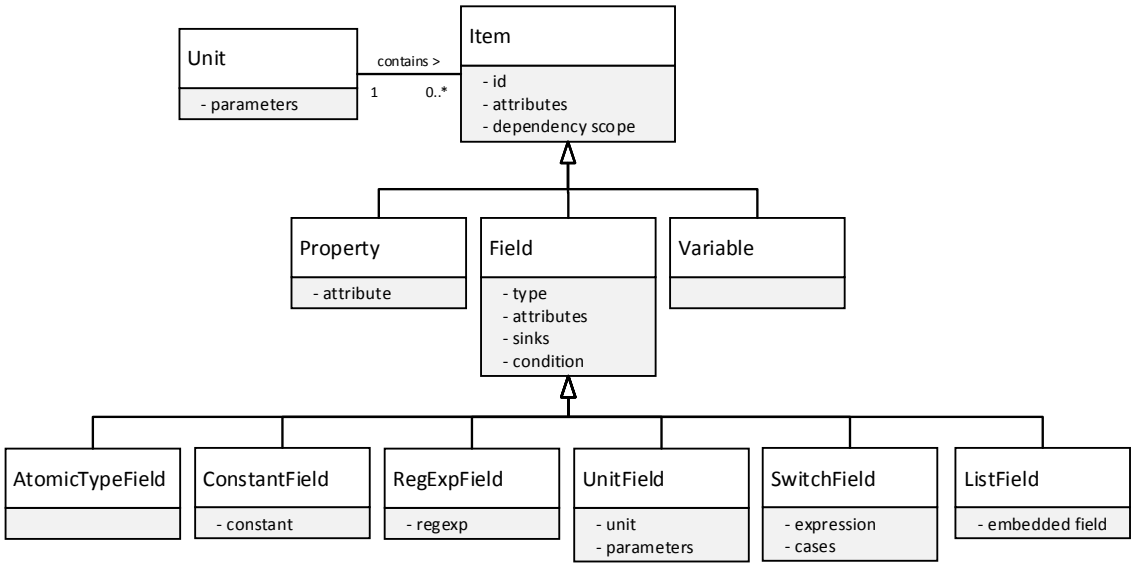

Figure 5.3: AST classes for unit specifications (simplified, excerpt).

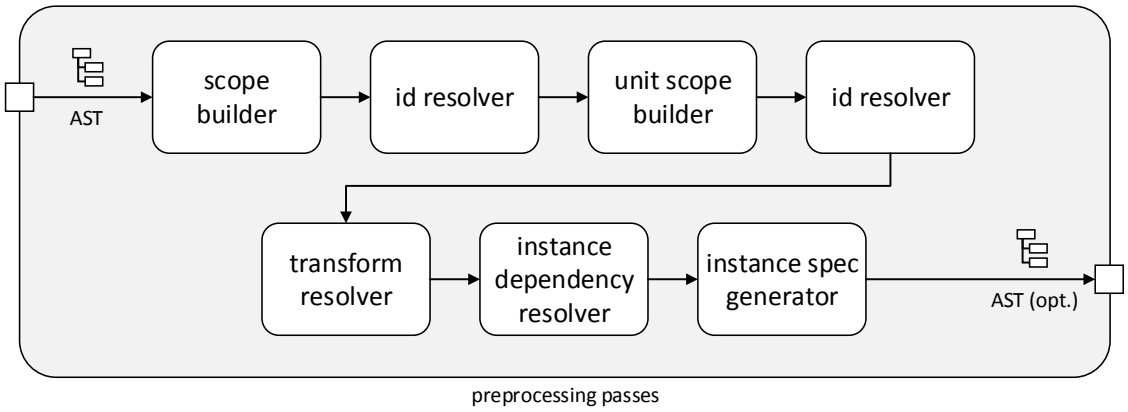

Figure 5.4: Diffingo generator preprocessing stage.

5.3 Preprocessing passes

The preprocessing stage consists of multiple passes (Figure 5.4). All passes are implemented as AST visitors. In the first four passes of preprocessing, references between AST elements are resolved. Such references include, for example, type name references in unit field or parameter definitions and field or enum value references in expressions. Next, transformations assigned to unit fields are resolved and their type information is propagated. Then, the instance dependency resolver assigns dependency scopes to different fields and variables in unit instantiations. And finally, the dependency information is used to create an optimized version of the unit specification for each instantiation. We describe these four parts of the preprocessing stage below.

Resolving AST element references. Unresolved references between elements in the AST are represented by objects of special expression and type subclasses. These objects store an identifier that corresponds to the (possibly scoped) name of the element that they are referencing.

To replace these objects in the AST with pointers to the elements that they are referencing, we first determine the names that are bound in different scopes, such as names of declarations in different modules, enum value labels in the scope of enum types, or parameter and field names in the scope of unit types. Next, we look up the identifiers of unresolved reference objects in the resulting name mappings of the different scopes. Thereby, we consider names in scopes closer the unresolved reference with higher priority than those bound in more global scopes.

Building the scope maps and resolving identifiers occurs in two separate phases: First, the scopes of modules and types other than units (for example, enums) are built and references to names bound in these scopes are resolved (in the *scope builder* pass and first *id resolver* pass). Second, the scopes of units are built and references to names bound in the unit scopes (for example, unit fields, variables and parameters) are resolved (in the *unit scope builder* pass and second *id resolver* pass), as they require that the types of unit fields, variables, and parameters are resolved already. In this second phase, `self` and `$$` references are also resolved.

Resolving transformations. After resolving element references, we perform an additional pass (*transform resolver*) in which transformations specified in `&transform` and `&transform_to` attributes of fields and variables are considered. This pass determines the correct transformation functions that should be applied during parsing and serializing and updates the internal type of the field accordingly. For example, if a wire-format integer value is transformed into an enum value, the internal type of the field becomes said enum type. This information is then later used during code generation to ensure that the generated data types contain the correct class member specifications.

Analyzing field dependencies for unit instantiations. The second-to-last preprocessing pass (*instance dependency resolver*) implements the dependency analysis for unit instantiations, which we described in Section 4.3. In this pass, we first build a map of dependencies between fields, variables and units (representing the dependency graph) according to the mechanism described before. We then iteratively collect the dependency scopes for the individual fields, variables, and units by adding dependencies until the algorithm converges, starting with the fields specified in the instantiation and its original unit. In our current implementation, we do not yet consider constrained accesses and instead over-approximate them.

The resulting dependency scopes for different fields and variables are then stored with the unit instantiation. In our current prototype implementation, we do not yet consider constraints put on fields, variables, or list elements.

Optimizing unit instance specifications. In the final preprocessing pass (*instance specification generator*), we generate optimized unit specifications for unit instantiations. Our current prototype implements the optimization mechanism for length-encoded fields described in Section 4.3. We copy those parts of the original unit specification that cannot be optimized. For other parts, we determine sequences of irrelevant fields that can be combined together and replace them with single byte-array fields with the accumulated length in the instantiation specification.

5.4 Code generation

The last generator stage implements the actual generation of C++ code. It consists of three concurrent passes, again implemented as AST visitors (Figure 5.5). They realize the generation of C++ data structures and functions, parser classes, and serializer classes respectively. The generated code makes use of a runtime library that provides data structures for common data types and functionality that simplifies the implementation of parsers and serializers.

Declarations inside a module are translated into C++ data types and functions. Type declarations other than units and enums are represented by simple C++ typedefs. Enum type declarations are translated into C++ enum declarations and units as classes, as we describe later. Function declarations in the module become C++ functions. The functions of transformations are inlined into the parser and serializer code. Unit instance declarations were transformed in the preprocessing passes into new unit type specifications,

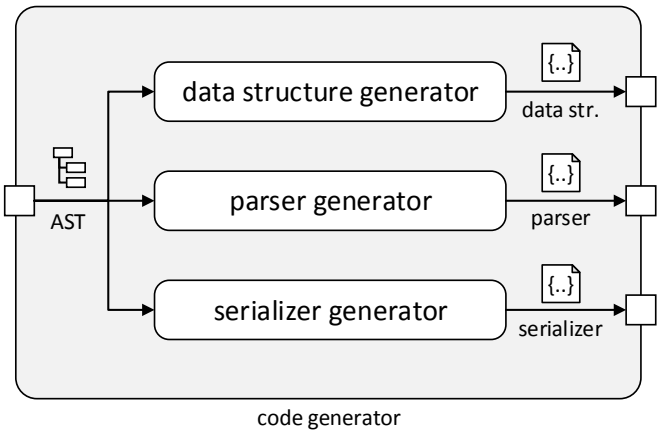

Figure 5.5: Diffingo generator code generation stage.

which will be translated just like normal units. We generate parsers and serializers for units or unit instantiations given as a command line parameter.

In the remainder of this section, we first describe the layout and generation of unit data structures and of the data types provided by the runtime library. Next, we describe the interface and implementation of the generated parsers and serializers.

5.4.1 Unit data structures

Each unit is represented as a C++ class in the generated data structures. The unit's application-accessible, non-constant value fields and variables become members of this class. For simplicity of generation, we make all member variables of the class public.

As mentioned in Section 4.1, we layout parsed message objects inside a fixed-size memory buffer. Values of fields and variables with static size are simply embedded into the class of their unit. For other field types, we place the data at dynamically determined positions in the buffer after the message object and store a pointer to the data location and its length in the unit class.

In our current prototype, we place variable-length data items in sequence in the memory buffer. That is, we keep track of the next free location in the buffer during parsing. When we need to store a new variable-length item, we place it at the next free location in the buffer and update the next free location to point to the location after the item. As a result, variable-length fields/variables cannot later be expanded by the application without copying their data to a different location in the buffer. Alternatives would be to reserve space for these fields/variables in the buffer according to a maximum-length provided for each such field/variable or to employ a more complicated user-level memory allocation mechanism. These alternative approaches trade larger memory consumption or higher complexity for a more efficient update mechanism.

In the following, we describe the way in which the different field/variable types and compositions are translated into unit class members.

Field types. Most atomic field/variable types can be represented by corresponding C++ types. For example, `bool` fields/variables are translated into C++ boolean member

variables and `int` fields/variables of different bit-sizes and signs are translated into C++ integer type members, such as `int32_t` or `uint64_t`.

An exception are byte-array and string field/variable types. We represent these by C++ structs that store a pointer to the beginning of the byte array or string and its length. The corresponding struct types `var_bytes` and `var_string` are provided by the runtime library and defined as follows:

```
struct var_bytes {
    size_t len_;
    char* data_;
};

struct var_string : public var_bytes {};
```

Sequences. Field/variable sequences are translated into a sequence of member variables in the C++ class for the unit they are contained in. We describe how sequences in alternatives of a `switch` statement are handled below.

Choices. A choice, that is, a `switch` statement in Diffingo's specification language, is translated into a member variable of a C++ struct type that stores the active alternative of the choice (that is, the alternative taken during parsing) as an enum value and the translated field/variable sequences for the different alternatives within a C++ union. The C++ member variable for the switch statement is given a generated name. An example for the type of this variable for a two-alternative `switch` statement is shown below. Such a type will be generated and embedded into the definition of the C++ class of the unit that contains the `switch` statement.

```
struct {
    enum {alt_1_, alt_2_} alternative_;

    union {
        struct {
            // member variables for alternative one
        } alt_1_;
        struct {
            // member variables for alternative two
        } alt_2_;
    } value_;
} switch_x; // name of switch field is auto-generated
```

In addition to the translation for choices described above, we also support adding the fields and variables of all alternatives of the switch statement successively and directly as member variables to the unit's C++ class. This may result in a larger internal representation of the unit in the memory buffer, but makes it easier for applications to access the fields of different alternatives.

Lists. Fields of list type are represented by a member variable of a C++ struct type similar to those for byte-array and string type fields. For a list of dynamic length, however, we need to store both a variable number of list elements as well as a variable number

of pointers to these list elements (as list elements may have fields of variable length as well). As the number of pointers is not known before all elements of the field are parsed, we store the array containing the pointers after the actual list elements in the memory buffer and place a pointer to it in the member variable representing the list field. The C++ struct type that stores this pointer and the length of the list is shown below.

```
template <typename ElementT>
struct list {
    typedef ElementT* pointer_array[];

    size_t len_;
    pointer_array* elements_;
};
```

Example. Consider again the unit specification for the Memcached binary protocol we provided in Section 3.6. We show the corresponding C++ class for the MemcachedCommand unit in Listing 5.1. The enum types for the `magic_code`, `opcode`, and `status` fields are not shown, but are also translated into corresponding C++ enum types.

```
1  class MemcachedCommand {
2    public:
3      MemcachedMagicCode magic_code;
4      uint8_t opcode_binary;
5      MemcachedOpCode opcode;
6      uint16_t key_len;
7      uint8_t extras_len;
8      var_bytes anon1_; // "reserved" field
9
10     struct {
11         enum {alt_1_, alt_2_} alternative_;
12         union {
13             struct {
14                 uint16_t vbucket_id;
15             } alt_1_;
16             struct {
17                 MemcachedResponseStatus status;
18             } alt_2_;
19         } value_;
20     } switch1_;
21
22     uint32_t total_len;
23     var_bytes opaque;
24     var_bytes cas;
25     uint32_t value_len;
26     var_bytes extras;
27     var_string key;
28     var_bytes value;
29 };
```

Listing 5.1: C++ class for the `MemcachedMessage` unit.

5.4.2 Parser and serializer interfaces

The parser and serializer generated by Diffingo are two separate classes that provide a `parse` and `serialize` method respectively. In this subsection, we describe the interface of these methods (show below).

```
ParseResult parse(char* in_buf_start, char* in_buf_end, UnitArea* area,
                  ParserState* state, size_t* bytes_read);

SerializeResult serialize(char* unit, char* out_buf_start, char* out_buf_end,
                          SerializerState* state, size_t* bytes_written);
```

For our prototype, we implement parsing and serializing independent of the underlying network communication. This allows us to perform an evaluation of the benefits of our optimizations independent of network bottlenecks or variations. Therefore, we use input and output memory buffers as input to the generated parser and output of the serializer. Thus, the `parse` method is given pointers to the beginning and end of the input buffer (`in_buf_start` and `in_buf_end`) as parameters; analogously, the `serialize` method is given pointers to the beginning and end of the output buffer (`out_buf_start` and `out_buf_start`).

As output of the `parse` method, we provide a pointer to a `UnitArea` object (`area` parameter), which is our implementation of the memory buffer that stores a parsed message. A `UnitArea` wraps a single statically allocated memory buffer and maintains a pointer to the next free position in the buffer. The caller of the `parse` method can later obtain a pointer to the message in the buffer from the `UnitArea` object, as well as the size of the data in the buffer. The application can then process the message and later pass it back to the `serialize` method (`unit` parameter) to be serialized. For an efficient memory layout, the UnitArea object itself lives inside the memory buffer it wraps, as well.

Furthermore, the `parse` and `serialize` methods are resumable. For this purpose, they need a way to store their state between invocations. We make this state explicit by passing a state object (`state` parameter) to both of these methods, which uses a pre-allocated memory buffer to store the parser/serializer state. To resume parsing or serializing, the method is simply re-invoked with the last state object.

If the method pauses its execution, either because more input data is required (`parse` method) or because the output buffer is full (`serialize` method), it returns a corresponding status code. When they finish parsing/serializing, it returns a success code. If errors occur, an error code is returned. The different possible return values are defined as enum types `ParseResult` and `SerializeResult`. Both methods also return the number of bytes they have read or written via the `bytes_read` and `bytes_written` parameters.

5.4.3 Parser and serializer implementation

The parsers and serializers generated by our prototype are resumable and can store references into the input data buffers. In the following, we describe how we realize these features.

Resumable parsers and serializers. To support resumable parsing and serializing, the implementation of the generated parsers and serializers keeps track of the current state. For a modular implementation, we use a hierarchical method for tracking this state and store the state in the form of a stack: On the highest level, we track the current unit that we are parsing/serializing and the current position in the input buffer; on the next level, we track the current field that is parsed/serialized; and on the lowest level, we track the state of the parsing/serializing of the current field, including, how much of the field has been parsed or serialized and, in the case of the parser, the current state of a regular expression that is evaluated on the input stream. We store this information on a stack, because we can encounter recursive units. Thus, whenever we need to parse or serialize an embedded unit, we push a new state hierarchy on the stack.

To be able to easily resume from a state stored on the state stack, we store pointers to the code block that the parser or serializer should continue from. Therefore, we employ an advanced feature of the GNU C++ compiler[1] called *labels as values*. It allows us to declare names of for code blocks using `goto` labels and retrieve a pointer to the location of the label using a special notation. We can store this pointer in the state and then later jump to the location by de-referencing it in a `goto` statement.

The advantage of this approach is that it makes it possible to generate parsers and serializers that execute linearly until they need to pause their execution. Alternatively, we could have chosen to implement the support for resumption using an approach similar to that traditionally used by byte-code interpreters. In this approach, we would loop over a large switch statement over the current state to proceed from one state to another. However, with the hierarchical state that we designed, this approach would require three nested switch statements. The overhead of executing these statements between each state can be significant. For this reason, byte code interpreters also often employ an optimization called "threaded code" where each interpreted byte code calls or jumps to the correct handler routine for the next byte code [9]. Our approach can be compared to an equivalent of this optimization for our generated parsers and serializers.

Yet another alternative would be not to make the parser and serializer state explicit, but instead to use the function call stack as their state and pause and resume the parser and serializer by saving and restoring the call stack between invocations. Such an approach is, for example, used by the Spicy parser generator [46].

Storing references into input data buffers. As described in Section 4.3, we can replace fields in units that aren't application-accessible by pointers into the input data buffers. In our current prototype implementation, we do not yet integrate our parsing and serializing logic with a user-level TCP stack. Instead, we can simulate the advantages we can gain by avoiding copying between the input data buffers and the memory buffers that store the parsed messages. In this case, we simply store references to ranges in the input data buffers in the unit class. During parsing, these ranges are updated; during serialization, data is copied from the input data buffer range to the output buffers. Our implementation of this feature is designed to be extended and integrated with a user-level TCP stack in the future.

[1] This feature is also available for Clang/LLVM compilers.

5.4.4 Parser and serializer generation

Our current prototype can generate parsers and serializers for messages with length-encoded fields. We are currently working on extending it with support for delimiter-encoded (regular expression) fields and fields that require look-ahead. The architecture of the generated parsers and serializers is designed to be easily extended in this way in the future. Subsequently, we describe how different field and variable types and compositions are translated into parser and serializer code.

Fields. To parse a field, we generate a (labelled) code block that implements the parsing logic for the field. For a constant-length field, we check at the beginning of the code block if we have enough data available on the input stream to parse the field. If there is not enough, we save the current state and return a corresponding result code. Otherwise, we parse the value of the field into the corresponding member variable of the current unit's class and advance the pointer into the input buffer. For variable-length fields, we either perform the same length check (if the field is small), or parse the field incrementally. In our prototype, we implement incremental parsing and serializing for variable-length byte fields by copying as much data as available and advancing the input pointer.

If there are any value transformations associated with a field, we parse its value according to its data type into a temporary location first and then apply the value transformation (inlined) to save it into the member variable. Stream transformations directly operate on the input stream and member variable; we can simply inline them.

When we encounter an embedded unit field, we add a new state hierarchy to the state stack and jump into the code block that parses the first field of said embedded unit. When the last field of a unit is parsed, the state is popped from the stack and parsing returns to the next field in its parent unit if the state stack is not empty. (Otherwise, parsing completes and returns a successful return code.)

To serialize a field, we generate a code block analogous to that for parsing it. Instead of checking for enough data on the input buffer, we check for enough free space on the output buffer; and instead of parsing field values from the input buffer, we serialize their values onto the output buffer. Embedded unit fields and transformations are handled in the same way as during parsing code generation.

Our generated parsing and serializing code is designed to be safe from common buffer overflow attacks. All generated code uses inlined helper functions that ensure that enough space is available in the memory buffers that they copy to before performing any copies.

Variables. For each unit variable, we generate a code block that updates its value during parsing according to the `&parse` attribute of the variable, if it is given. Likewise, we generate a code block that executes its `&serialize` attribute action, if given, during serialization. During parsing, these code blocks are executed in the order in which the variables occur interleaved with unit fields. During serialization, they are executed in reverse order before any field serialization occurs, as the effect of their `&serialize` attribute actions may otherwise change field values that were already serialized. Even before these code blocks, though, we execute blocks to update dependent fields and

variables, such as those referenced in `&length` attributes, to ensure that data from the internal representation, such as updated field or list lengths, has propagated to them.

Sequences. Sequences of fields are translated into succeeding code blocks that parse or serialize the individual fields of the sequence (possibly interleaved with code blocks that execute `&parse` attributes of variables, as described above).

Choices. As mentioned before, we currently only support choices not based on look-ahead in our prototype. Choices are translated into C++ `switch` or `if`/`else` statements depending on the type of the expression (as C++ does not support switching over `string` types, for example). For each alternative, we generate a sequence of code blocks representing the sequence of fields and variables of the case as described above. This sequence of code blocks is then executed in a body of one of the `case` blocks of the C++ `switch` statement, or of the `if`/`else` statement.

Lists. As mentioned before, we currently only support lists not based on look-ahead (that is, with a given length) in our prototype. Such lists are translated into a three-part code block. During parsing, the first part evaluates the `&length` attribute expression for the list and initializes the state of parsing the list field (i.e. the list length and current number of elements, as well as an array with pointers to the elements parsed so far) in the lowest level of the state hierarchy.

The second part implements the iteration through the elements of the list. As long as more elements are expected, it proceeds in a similar way as in the case of directly embedded units: First, it pushes a new state hierarchy on the stack; then, it jumps to the parsing logic for the list element unit. When the parsing of the embedded unit returns, the code block stores a pointer to the parsed element into the list parsing state and increments the number of elements parsed.

When all elements are parsed, the second part finishes. The third part then copies the array of pointers to the output buffer and updates the list field's member variable to point to this array and store the length of the list.

We show the generated code for parsing a list field in pseudo-code below:

```
list:
  // init list state in current state hierarchy:
  // store length of list, number of elements, array of pointers to elements
  while (state->list_length > state->element_num) {
    // push state hierarchy onto stack
    // jump to embedded unit parsing logic with return to list_it_return
list_it_return:
    // store pointer to parsed element to state, increment number of elements
  }
  // copy element pointers as optimal-length array into output buffer
  // update field's member variable to point to array
```

For serializing a list field, a similar code block is generated. In this case, however, element pointers and list lengths do not need to be stored in the serializing state hierarchy, as they are available in the member variable for the list field.

6 Evaluation

In this chapter, we provide a preliminary evaluation of the Diffingo prototype. With this evaluation, we do not aim to validate the prototype in a multitude of real-world applications, but rather to analyse the feasibility of the ideas behind Diffingo.

We have shown the applicability of Diffingo's specification language design for examples of binary and textual protocols in Section 3.6: A full specification of the messages exchanged in the Memcached binary protocol and a specification of HTTP requests and responses that includes support for content-length and chunked delivery of message bodies. We argue that the language is also suitable to capture the description of other common network application protocol messages, as its expressiveness is similar to that of related work in the area (see Chapter 7). However, as mentioned above, we leave the validation of this statement by means of specifying and evaluating additional example protocols as future work.

Instead, we focus on supporting the ideas, particularly the optimization mechanisms that aim to make Diffingo-generated parsers and serializers efficient, by providing a performance evaluation for the parsers and serializers generated by our Diffingo prototype for the Memcached binary protocol. In this chapter, we first discuss the setup and results of this performance evaluation. Subsequently, we discuss the limitations of the system and its evaluation.

6.1 Performance evaluation

In this section, we evaluate the performance of the parsers and serializers that Diffingo generates for the Memcached binary protocol. We compare their performance against a hand-rolled implementation of a Memcached message parser and serializer, which we have taken from the open-source implementation of libmemcached [15]. Furthermore, we show the performance difference that can be achieved by optimizing the generated parser and serializer to the Memcached router application described in Section 1.2, as well as the benefits than can be gained by avoiding copying application-irrelevant data into the internal representation of a message. Finally, we demonstrate that resumable parsers can be beneficial when individual messages arrive incrementally.

6.1.1 Experimental setup

For our experimental evaluation, we consider the parsing and serializing logic in isolation from its environment. That is, we perform our experiments independent of the underlying network communication that would typically occur in a network application. This allows us to perform an evaluation of the performance of our generated parsers and

serializers and of the benefits of our optimizations independent of network bottlenecks or variations.

For this purpose, we execute the parsing logic on an input memory buffer that holds a wire-format message. The parsing logic parses from this memory buffer as if it was an input buffer of the underlying TCP stack, provided, for example, through a socket-API or via an integration of the parsing logic with the stack. The parsing logic parses the wire-format message from this input buffer into an internal representation of the parsed message (also in a memory buffer). The serializing logic is then executed on this parsed message and writes its output into an output memory buffer, that represents the output buffer of the underlying TCP stack, analogous to the input buffer.

Diffingo-generated parser/serializer. We evaluate three different versions of the parser and serializer generated by Diffingo. The first version (*default* version) is generated from the full specification of the Memcached messages as defined in the specification provided in Section 3.6. It functions as a baseline for the other two versions.

The second version (*compact* version) is generated from the optimized specification for the instantiation of the Memcached specification for the Memcached router example application (as shown in Section 3.4). In this case, the only application-accessible fields are the *Opcode* and *Key* fields. In addition to these fields, the resulting optimized specification also parses the different length fields to determine field and message boundaries. Other fields are combined and their wire-format value is copied during parsing into the internal representation of the parsed message to allow later re-serialization of the message.

The parser and serializer of the third and last version (*pointer* version) are similar to those generated for the compact version. But instead of copying the wire-format value of irrelevant fields from the input buffer into the internal representation of the parsed message, we store only pointers to the beginning and end of their wire-format value in the input buffer. This version of the parsers simulates an integration of the parsing and serializing code with an underlying TCP stack to avoid copies, as previously described in Section 4.3. A similar benefit can also be achieved if the parsed message does not need to be re-serialized later. In our example, however, we need to re-serialize the message when it is forwarded by the router application. Therefore, the serializer for the pointer version copies the wire-format value of irrelevant fields from the corresponding ranges in the input buffer to the output buffer.

Libmemcached parser/serializer. To provide an external baseline for the performance of the Diffingo-generated parsers and serializers, we have isolated the parsing and serializing functionality of the libmemcached library [15]. As with the Diffingo-generated parsers and serializers, we execute the libmemcached parser and serializer on memory buffers.

The libmemcached parser uses similar memory management optimizations as Diffingo. The internal representation of parsed messages is stored in memory buffers that are only allocated once and reused for future messages. In contrast to Diffingo's parsers and serializers, though, the libmemcached parser and serializer are not resumable. That is, when they discover that they do not have enough data available to proceed with parsing,

or that the output buffer is full during serializing, they return an error code and have to restart parsing again when more data (respectively, space) is available. Libmemcached parsers also do not employ any application-specific optimizations.

Experiments. We compare the performance of the three versions of the parser and serializer generated by Diffingo and the libmemcached parser and serializer in three experiments: First, we compare parsing time; second, we compare serialization time; and lastly, we show the effect of incremental processing by comparing parsing time for chunked messages.

In all experiments, we compare the time it takes to parse or serialize a single message depending on its size. As messages, we use Memcached requests with a variable value length. We consider messages with empty value and values with 1, 3, 5, 10, 15, and 20 kilo-byte (kB) length. To simulate incremental message parsing, we provide the message to the parser in chunks of 512-byte length.

To obtain accurate measurements, we parse and serialize a total of 50 million messages in 5 repetitions of each experiment (that is, 10 million per repetition). We run our experiments in a single thread on a 3.10 GHz Intel Core i7-3612QM core on an Ubuntu 14.04 x64 system with 8GB of RAM. Parsers and serializers are compiled with GCC 4.8.4, optimization level 3.

6.1.2 Results

In the following, we present and discuss the results of the three performance experiments described above.

Parsing messages. We show the time required to parse a Memcached message by the different parser versions in Figure 6.1 and Table 6.1.

As general trend, we observe that the default version and compact versions of the Diffingo-generated parser and the have a close-to linear increase of time required to parse a message with increasing message size. The runtime of Diffingo's pointer version parser, however, is close-to constant irrespective of message size. The reason for this dramatic difference in run time is due to the fact that the pointer version does not need to copy the value of the *Value* field of the message. As this is the field, whose size increases with increasing message size, the other versions of the parser need to copy more data, while the pointer version simply stores references to a larger range of the input buffer. Thus, its runtime remains constant. Obviously, this benefit can only be gained if the parser would be integrated in such a way into the underlying TCP stack, that the data can remain in the stack's input buffers until the message is serialized again, as discussed before.

Compared to the default version of the Diffingo-generated parser, the libmemcached parser is about 1.1–1.9 times as fast. Its performance benefit decreases with increasing message size. For very small messages (value field of 0–1 kB length), it is almost twice as fast; for middle-sized message (5–10 kB), it is about 1.6 times as fast; and for larger messages (15–20 kB), it is only 1.1–1.3 times as fast. We believe that the primary reasons for the performance benefit of the libmemcached parser over Diffingo's default parser

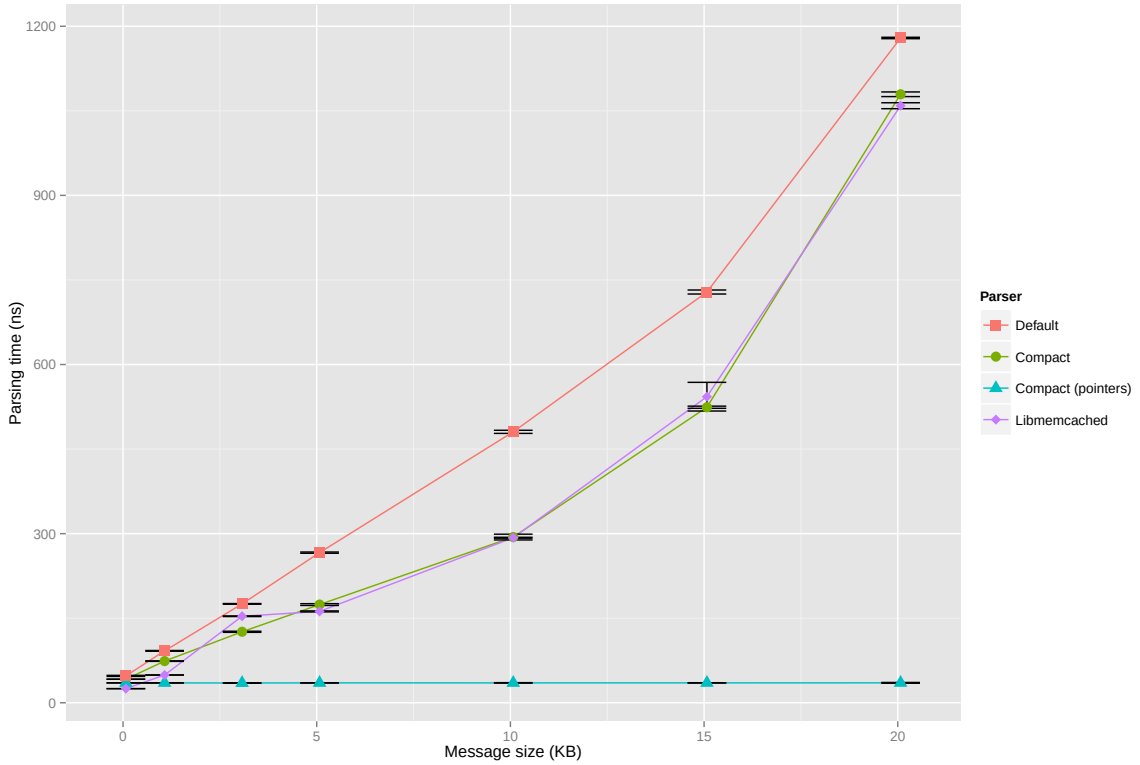

Figure 6.1: Performance of the different Memcached message parsers. We show the time (in nano-seconds) required to parse a request message depending on its size.

Table 6.1: Performance of the different Memcached message parsers. We show the time (in nano-seconds with a 95% confidence interval) required to parse a request message depending on the size of its value field.

Parser	0 kB	1 kB	3 kB	5 kB	10 kB	15 kB	20 kB
default	47.7 +/- 0.9	92.0 +/- 0.7	175.4 +/- 0.6	266.3 +/- 1.1	480.6 +/- 2.8	728.6 +/- 3.6	1179.2 +/- 1.3
compact	41.7 +/- 0.1	73.9 +/- 0.5	125.9 +/- 1.0	174.2 +/- 1.6	293.9 +/- 5.1	524.2 +/- 1.9	1079.4 +/- 4.1
pointer	35.3 +/- 0.2	35.3 +/- 0.3	35.1 +/- 0.4	35.3 +/- 0.3	35.3 +/- 0.4	35.3 +/- 0.3	35.3 +/- 0.7
libmemcached	25.0 +/- 0.1	49.2 +/- 0.2	153.5 +/- 0.6	162.0 +/- 0.9	292.7 +/- 0.8	542.9 +/- 25.4	1059.0 +/- 5.3

version are the overhead of the support for resumable parsing in Diffingo's parser and certain optimizations that the hand-rolled parser implementation of libmemcached can perform. For example, the hand-rolled implementation copies the whole of the message header at once and then fixes the byte-order of the fields later where necessary, while the default generated parser copies and converts values field-by-field. We believe that it should be possible to extend Diffingo to apply such an optimization in a general way in the future. Furthermore, the parser for libmemcached does not handle the *vbucket/status* field in different ways depending on whether the message is a request or response, whereas the Diffingo parser stores the value of this field in different member variables of the parsed message object.

If we compare the libmemcached parser with the compact version of the Diffingo-generated parser, however, the performance benefit is much smaller and disappears almost completely with messages over 1 kB size. For very small messages (0–1 kB), libmemcached is still about 1.6–1.7 times as fast as the compact parser. We believe that for the small message sizes, the performance benefit is primarily due to the overhead of the resumable parsing support in Diffingo's parser. The optimizations of the hand-rolled implementation of libmemcached are matched by the combination of message fields in Diffingo's compact parser for larger messages.

To put the absolute time required for parsing single messages in context: For very small messages (0–1 kB), the libmemcached parser achieves a throughput of about 20–40 million messages per second or 20–160 Gbit/s and Diffingo's default parser 10–20 million messages per second or 10–90 Gbit/s. For large messages (20 kB), they achieve throughputs of about 950 million packets per second or 144 Gbit/s and 850 million packets per second or 130 Gbit/s respectively.

Serializing messages. We show the time required to serialize a Memcached message by the different parser versions in Figure 6.2 and Table 6.2.

Contrary to the performance of the different parser versions, the performance of the different serializers is very similar. All serializers have a near linear relationship between time and message size. The pointer version cannot achieve a linear runtime, as it also needs to copy data of irrelevant fields into the output buffer.

For very small message sizes (0–1 kB), the hand-rolled serializer from libmemcached is about 1.6–1.8 times as fast as the default version of the Diffingo-generated serializer, but only about 1.2–1.3 times as fast as the compact and pointer versions. Its performance benefit can be explained, as before, by the overhead of resumable serializing and the optimizations in the hand-rolled code. For larger messages, it has a constant performance benefit of about 20–30ns per message over the Diffingo's default serializer, but is on-par with the compact and pointer versions.

The compact and pointer versions are about 20–30 % faster than the default Diffingo version for small messages (0–1 kB). For larger messages, the compact version can achieve a constant benefit of 30–60ns over the other two. The slight benefit of the pointer version over the default version seems to disappear with larger message sizes. We assume this is due to the fact that the pointer version also copies from input buffers, and not only from the buffer holding the parsed message, which may have an effect on memory locality.

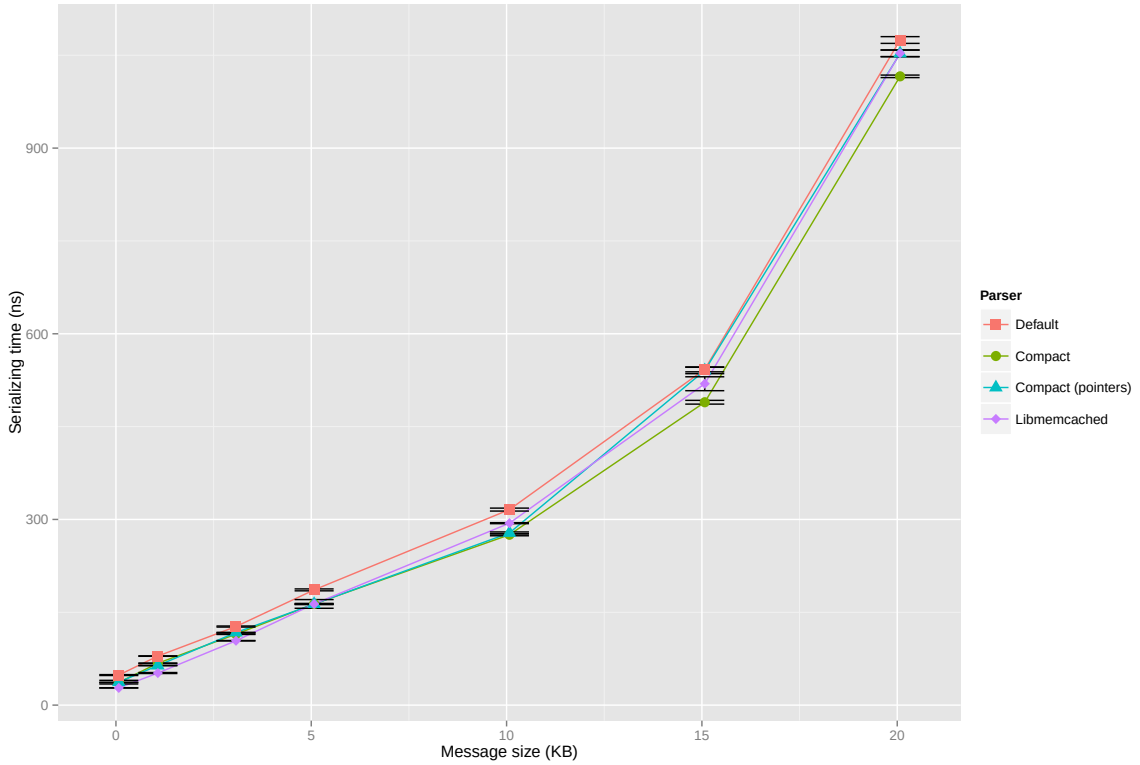

Figure 6.2: Performance of the different Memcached message serializers. We show the time (in nano-seconds) required to serialize a request message depending on its size.

Table 6.2: Performance of the different Memcached message serializers. We show the time (in nano-seconds) required to serialize a request message depending on the size of its value field.

Parser	0 kB	1 kB	3 kB	5 kB	10 kB	15 kB	20 kB
default	48.6 +/- 0.7	79.2 +/- 0.4	127.0 +/- 0.8	186.2 +/- 1.5	315.9 +/- 2.4	542.5 +/- 4.0	1074.7 +/- 5.4
compact	36.6 +/- 0.3	67.6 +/- 0.3	114.6 +/- 0.5	163.5 +/- 7.0	275.3 +/- 1.7	489.4 +/- 3.0	1015.9 +/- 2.0
pointer	37.1 +/- 3.0	63.9 +/- 0.4	117.3 +/- 0.3	163.2 +/- 0.4	278.2 +/- 1.6	540.7 +/- 5.2	1052.9 +/- 5.4
libmemcached	27.8 +/- 0.3	51.8 +/- 0.8	104.0 +/- 0.4	163.4 +/- 0.9	294.0 +/- 0.8	519.3 +/- 11.2	1053.3 +/- 5.4

Incremental processing. Next, we consider parsing a Memcached message that arrives in chunks of 512 byte length and is incrementally processed by the parser. We show the time required to parse such a Memcached message by the different parser versions in Figure 6.3 and Table 6.3.

There is no significant difference in performance for very small messages with an empty value field (0 kB) compared to the results for non-chunked parsing. But the benefit of resumable parsers is immediate for any messages larger than the chunk size. The compact and pointer versions of the parser surpass the performance of the libmemcached parser even for messages with a 1 kB value field. For any larger messages, even the default version of the Diffingo parser is faster than the libmemcached parser. For medium to large message sizes (3–20 kB), the default version is about 1.2–1.4 times as fast as libmemcached, the compact version is about 1.5–1.8 times as fast.

The pointer version of Diffingo's parser is again much faster than all other parsers, because it can avoid large copies. Its time per message is also linear with the increase in message size, but with a much lower factor than it is for the other parsers. This is primarily because its parse method is called multiple times for chunked messages.

6.1.3 Discussion

The results of the parser and serializer experiments show that there is some overhead in parsers and serializers generated by Diffingo, primarily for small messages. As discussed above, we believe this is primarily due to their feature for incremental/resumable processing. Another source of overhead may also be that our parsers/serializers process fields one at a time and track the current stream position as an explicit variable between fields. This design results from the modular implementation of our parsers and serializers, which separates the way different field types are processed from the way in which different compositions of field types are processed.

Nevertheless, we believe that the results shown are competitive. We compare against a hand-rolled parser/serializer implementation from libmemcached that also uses memory management optimizations. The default parser generated by Diffingo is in all cases at least half as fast as the hand-rolled parser. But its overhead diminishes with larger messages, where it is closer to only 10% of the libmemcached parser. The default serializer is even closer to libmemcached's performance.

We also observe that the compact version of our generated parsers and serializers can achieve performance very close to that of libmemcached, primarily for middle- to large-sized messages, and can achieve an advantage of 10–60% over Diffingo's default version. Thus, the optimization of the message specification for the compact version is fruitful and can reduce the overhead significantly. If we further avoid copying irrelevant fields into the parsed message object, we can reduce the parsing time drastically without significant effects on serialization time (in the common case that most of the message body is irrelevant). This result makes it important to further evaluate how we can exploit the opportunity for integrating the parsing and serializing with a user-level TCP stack.

For chunked messages, we observe that all Diffingo-generated parsers can obtain a performance benefit over the libmemcached parser because they can pause and resume parsing. This is even though the overhead of re-starting parsing from the beginning of

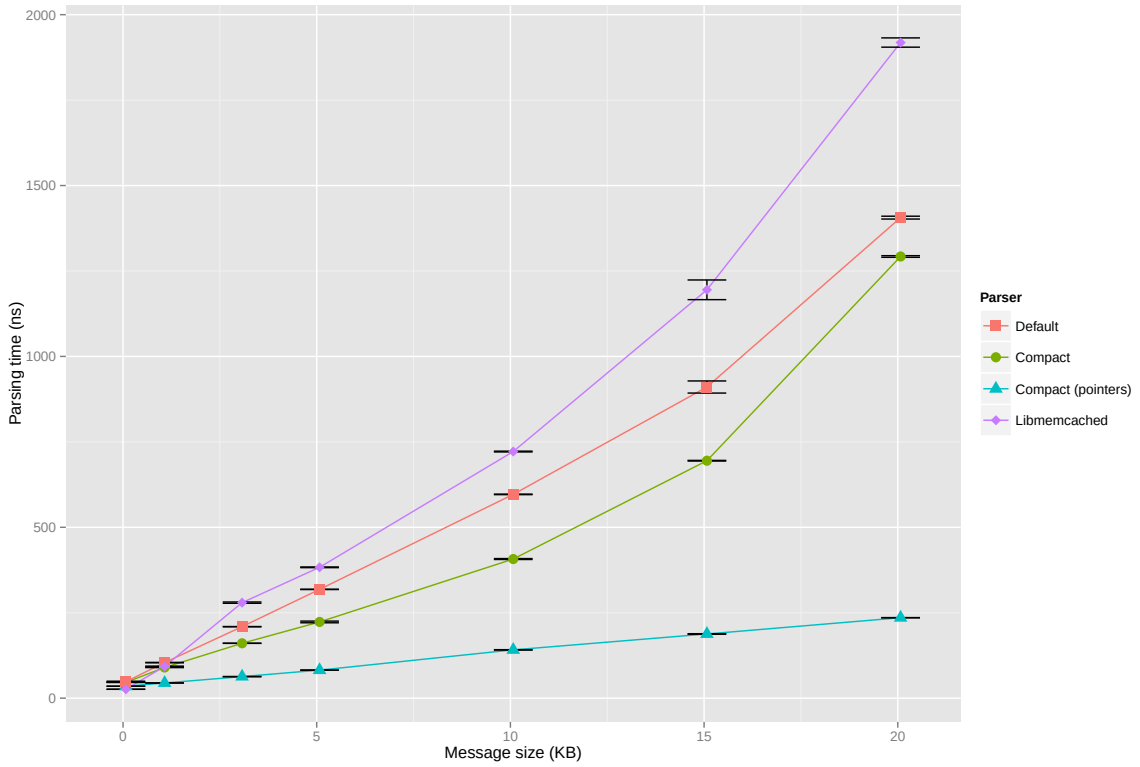

Figure 6.3: Performance of the different Memcached message parsers for incremental parsing. We show the time (in nano-seconds) required to parse a request message depending on its size if it is delivered incrementally in chunks of 512 bytes.

Table 6.3: Performance of the different Memcached message parsers for incremental parsing. We show the time (in nano-seconds) required to parse a request message depending on the size of its value field if it is delivered incrementally in chunks of 512 bytes.

Parser	0 kB	1 kB	3 kB	5 kB	10 kB	15 kB	20 kB
default	48.3 +/- 1.4	104.1 +/- 0.6	209.2 +/- 0.4	318.3 +/- 0.5	596.3 +/- 0.8	910.5 +/- 17.7	1406.2 +/- 4.2
compact	46.2 +/- 0.2	89.9 +/- 0.2	160.6 +/- 0.2	223.1 +/- 2.5	407.1 +/- 1.3	694.8 +/- 0.9	1292.4 +/- 2.4
pointer	34.9 +/- 0.0	44.5 +/- 0.2	63.1 +/- 0.2	82.1 +/- 0.3	141.4 +/- 0.6	187.8 +/- 0.8	235.5 +/- 0.3
libmemcached	26.0 +/- 0.1	93.0 +/- 0.8	279.6 +/- 1.9	382.9 +/- 1.0	721.7 +/- 1.1	1194.9 +/- 28.8	1918.5 +/- 13.6

the message is relatively small in the case of a Memcached parser: After parsing the first 24 header bytes, the parser can determine if enough data is available on the input buffer to begin parsing. The overhead, thus, is only to re-parse the header fields containing the body length. For other types of messages, such as delimiter-encoded messages, we expect that the overhead of re-starting parsing from the beginning of the message can be significantly larger. Similarly, the overhead will grow if messages become larger than the considered Memcached messages and/or are delivered in more chunks.

Of course, in all cases, the overhead of the Diffingo-generated parsers will also depend on the complexity of the message specification. To better understand this relationship, we plan to implement and evaluate parsers and serializers for more protocols in the future.

6.2 Limitations

While our initial performance evaluation shows promising results, there are many limitations, both to the evaluation and to Diffingo and its prototype implementation itself. In this section, we discuss these limitations and any solution approaches.

Evaluation. The evaluation presented in this report is very limited. For the time being, we could only evaluate our prototype on a single protocol, and only in performance tests isolated from network and application processing. This is primarily due to the time constraints of the project, which limited the scope of the implementation and evaluation.

To better support the applicability of our language abstractions and optimization mechanisms, extending both the prototype implementation and evaluation to more use cases are important items of future work. Furthermore, we plan to test the performance of the generated parsers and serializers in an end-to-end fashion in the context of a network application/service deployed in a network. In this context, we also intend to integrate the parsing and serializing logic with a user-level TCP stack and evaluate the challenges and benefits of such an integration.

Another part of the Diffingo framework that we did not implement or evaluate yet is the automatic extraction of unit instantiations from application source code. In its current state, the framework depends on a manual specification of the instantiation by the application developer. As we mentioned in Section 2.1, we intend to automatically derive the instantiation configurations via static analysis of the application sources instead, which takes the responsibility for this task away from the developer. To evaluate the feasibility of this approach (for example, how accurate such an extraction would be and where it would have to over-approximate field accesses), we plan to implement such a static analysis for network applications specified in the Flick programming language [1].

Specification language. Apart from requiring a more comprehensive evaluation (see above), the specification language has a significant limitation in that it currently only considers individual messages in isolation. In some protocols, the layout of a message or its semantics may be different depending on the history of messages exchanged between the communication partners. For example, an HTTP reply to an HTTP GET request will contain a body, while a reply to an HTTP HEAD request will not, even though

their headers look identical. In the future, we may consider extending the language to support passing information between messages exchanged in up- and down-flows of a communication channel.

Another feature that we considered to add to the language is a more explicit mechanism for enabling or disabling the support of certain aspects of a protocol in the generated parsers and serializers. For example, if the application developer knows that HTTP message bodies are always sent in end-of-data delivery mode in his application, then it would make sense to disable support for chunk-encoding and content-length delivery. In our current language design, we allow the developer to disable the support by constraining the unit variable that determines the possible delivery modes. However, this mechanism requires comprehensive static analysis to determine which code paths in the unit are impossible as a result of the constraining and may result in an inaccurate over-approximation. We leave the evaluation of this mechanism and a comparison with an alternative, more explicit mechanism for disabling protocol features as items of future work.

Optimization mechanisms. Our current description of the optimization mechanisms for unit specifications/instantiations is an algorithmic one. One issue with this is that it is difficult to determine the correctness or validity of the approaches without a more comprehensive evaluation of use cases. So far, we could only evaluate the mechanism for combining irrelevant length-encoded fields on a single use case. As future work, we therefore plan to evaluate both mechanisms on a multitude of use cases. We also intend to formalize the specification of the mechanisms to provide stronger support of their correctness.

Furthermore, as discussed in Section 4.3, we are looking into extending the mechanism for combining delimiter-encoded fields with support for more general delimiters. In addition, we would like to revisit the handling of stream transformations and sink-fields, for example, to determine whether fields with stream transformations or multi-sink fields can be combined during optimization. We are also looking into heuristics and other algorithms that can help decide whether a field should be combined using delimiter-based or length-based field combination mechanisms, as mentioned in Section 4.3.

As described in Section 6.1.2, we also consider extending Diffingo with additional optimization mechanisms that can combine parsing multiple relevant fields at once into an internal representation. This may be possible if these fields are successive and their value does not require any conversion between wire-format and internal format.

Prototype implementation. Our current prototype only supports parsing and serializing of length-encoded fields. Analogously, we have only implemented the optimization mechanism for combining length-encoded fields, but not for combining delimiter-encoded fields. As mentioned before, we are currently working on extending the implementation with support for regular expression and look-ahead fields, as well as delimiter-based field combination.

In our prototype, we also currently only support a very specific mechanism for laying out parsed objects in memory, which results in a very space-efficient layout, but makes

variable-length values difficult to modify. As discussed in Section 5.4, we also consider extending the implementation with support for different memory management mechanisms, such as reserving a larger space for variable length values or storing such values in separate pre-allocated memory buffers of different sizes. It may even be beneficial to decide on the type of memory management depending on the application use case.

7 Related work

Diffingo aims to solve the problem of parsing and serializing application messages for processing in network applications, such as proxies, load balancers, firewalls or other (software-based) network functions. We survey a range of languages and tools for developing such network functions in the context of network functions virtualization (NFV) [17] elsewhere [44].

In this chapter, we provide an overview of work related to parsing and serializing data formats, as well as parser generation in general. The most closely related category of work is that of languages and systems used to describe, parse, and serialize data formats. We discuss these works in Section 7.1. Our work also has some similarities with parser generators and combinators which are typically used to parse programming languages. We discuss these topics in Sections 7.2 and 7.3.

7.1 Describing, parsing, and serializing data formats

The challenge of making it easy to parse and/or serialize data formats and messages exchanged between applications has been investigated from multiple different angles. Historically, an area where the idea of describing data formats declaratively gained large traction was that of parsing data in non-standard, or ad-hoc, data formats. Prime examples of use cases for such data formats are monitoring services that parse domain-specific log files or traces, as well as scientific analysis software that parses simulation results. While high performance was typically not a key requirement, the work in this area had to deal with a large variety of different data formats and their description. As such, many of its concepts for describing data formats are foundational and similar to Diffingo's message format specification language.

While the work for parsing ad-hoc data formats tries to capture already existing data formats, another category of work takes a reverse approach: It provides mechanisms to serialize data items into custom, compact, and highly efficient wire-format encodings. Thereby, the structure of the data items is defined by a similar declarative format description. Data serialization tools implementing such mechanisms are often used for ad-hoc communication between different or distributed parts of an application. While they are not suited to parse data that is provided in a different, already fixed, wire-format (as is typically the case for network protocols), the description languages that these tools use also have some commonalities with Diffingo's language.

Finally, with the trend towards realizing more network functions in software, there have been first advances into applying techniques similar to those for ad-hoc data formats to network protocols. Diffingo is one such example. We know of two other projects that aim to solve the same problem, which we build on and compete with.

In the following, we first discuss Diffingo's direct competitors in the domain of network protocol parsing. Subsequently, we explore work in the areas of ad-hoc data formats and data serialization.

7.1.1 Network protocols

Spicy. Like Diffingo, the Spicy parser generator (formerly called BinPAC++) [46] aims to generate parsers for network protocols, such as DNS or HTTP, from a message grammar specification. Spicy is the successor of the BinPAC parser generator [39]. It extends BinPAC with support for LL(1) grammar look-ahead parsing, refines its grammar language and ports it to run on a special execution environment for network traffic analysis, called HILTI (high-level intermediary language for traffic inspection) [46]. Spicy is used in the context of the Bro intrusion detection system [41]; therefore, its focus lies on parsing messages, rather than serializing them.

Diffingo builds on Spicy in many ways. We have reused large parts of Spicy's grammar language and based the implementation architecture of our prototype on Spicy's design. Both Diffingo and Spicy use units and fields as basic abstractions to define message formats. The languages of both also support look-ahead choices, fields defined by regular expressions, lists, and unit variables. Spicy further integrates a mechanism to add *hooks* to a field, which are functions that will be executed when the field is parsed. These are used in the Bro intrusion detection system to alert the system of events, such as newly parsed messages or message parts.

Spicy also implements a mechanism to parse messages incrementally. However, in contrast to Diffingo, which makes the parser state explicit to be able to save it between invocations, Spicy relies on a low-level feature of the HILTI environment. Spicy generates recursive-descent parsers, which use the function call stack of the HILTI language to store parsing state and to recurse/descent. To be able to pause execution of this call stack, it is kept in heap memory. When the parser reaches the end of the input stream, it saves processor registers and yields execution to a different call stack. To resume from the saved state, the processor registers are restored to their previous values and the call stack is switched back to the parser's. This allows for a simple pause/resume functionality, but may lead to saving large amounts of irrelevant state between invocations of the parser, as the call stack may not only contain parsing state. Furthermore, determining the correct allocation size for the call stacks in the heap is a difficult problem. By making parsing and serializing state explicit, Diffingo avoids these issues.

Contrary to Diffingo, Spicy does not include support for stream or value transformations, serializing parsed messages, or lambda and find expressions. Furthermore, it does not optimize parsing and serializing to application needs or perform memory management optimizations. Instead, it allocates memory dynamically during parsing as needed.

The overhead of these dynamic allocations could be observed in Spicy's performance evaluation. For parsing DNS messages, it requires more than 2.5x as much time as a hand-rolled parser. Given Diffingo's similarity to Spicy, we plan to directly compare the performance of both parser generators against each other in a future evaluation.

Nail. Nail is a parser and serializer generator for data formats including network protocols. It focuses primarily on binary protocols and files, although its techniques should be applicable to text formats as well. Nail uses a grammar language similar to that of Spicy and Diffingo: Its basic elements are structures (cf. units) and fields with type information. Structures can be embedded in other structures and composed in sequences, non-deterministic choices, or repetitions (cf. lists). The length of a field or repetition can also be determined by the value of another *dependent* field, but Nail does not provide simple ways to evaluate expressions during parsing or serializing, like Diffingo and Spicy's unit variables do. Nail does, however, support stream transformations of fields in a way similar to Diffingo. In fact, this is an aspect of Diffingo's language that was influenced by Nail's design. Further, both Diffingo and Nail aim to define the grammars in ways that allow both parsing and serializing, while Spicy does not.

While Diffingo's primary focus is on performance, Nail's is on security. Thus, Nail neither avoids dynamic memory allocations, nor supports application-specific parser/serializer generation or incremental parsing. Instead, Nail implements recursive-descent parsers and serializers using temporary memory arenas, which are dynamically allocated and zeroed immediately after parsing to prevent attack code to remain in memory. To implement non-deterministic choices, Nail uses a backtracking algorithm, while Diffingo and Spicy use LL(1) parsers. Sadly, Nail was not evaluated in isolation to determine the performance overhead of these mechanisms; we expect that it would be significant.

P4. As a less directly related work, we also include a short discussion of the P4 language for programmable network switches [10]. P4 (Programming Protocol-independent Packet Processors) aims to allow a dynamic reconfiguration of the flow-table layout in programmable switches in order to support multiple different low-layer network protocols. Contrary to the protocols that Diffingo, Spicy, and Nail consider, these are typically much simpler in design than application-layer protocols. P4 includes a language that allows developers to define the different supported network protocols as sequences of fixed-width fields in their headers. The inter-dependencies between protocols of different layers are defined in the form of a state machine, which can later be installed in switches. In contrast to the other network protocol parser generators, the wire-format representation of packets in P4 is also directly used as internal representation. As a result, no value transformations, choices, variable-length fields, field repetitions etc. are supported and re-serializing a message is not necessary.

7.1.2 Ad-hoc data formats

PADS. The PADS (Processing At-hoc Data Sources) system [19] targets ad-hoc data formats (both binary and ASCII-based). It provides a language to describe their structure declaratively and a compiler that generates data structures and a parser for a data format based on such a description. PADS is maintained by the PADS project [21] and implementations of it exist for multiple languages. The original implementation was done in C (and generates parsers in C), the most current one is maintained in Haskell (and generates parsers in Haskell) [21].

PADS' description language uses abstractions similar to those of Diffingo's and Spicy's languages. Its basic entities are structures (cf. units), fields, unions (cf. choices), and arrays (cf. lists). It also supports non-deterministic choice. In addition, PADS allows the developer to specify constraints for field values, which are checked during parsing; this is useful to implement simple syntax and semantic checkers using PADS. PADS does not support mechanisms such as stream transformations or fields specified using regular expressions.

Interestingly, the parsers generated by PADS can be configured at run-time not to fill in certain fields of the internal data structures [19]. This idea is similar to Diffingo's in that it also allows to customize the parsing behaviour to a specific application. However, it is less powerful, because Diffingo can optimize the generated parsers at compile time and further combines irrelevant fields to make parsers more efficient. Simply not discarding the data in such fields is also not typically useful for network applications, as they may need to re-serialize the data. PADS, on the other hand, does not support serialization.

In the context of PADS, a later work describes a formal type system for data description languages and a corresponding operational semantics for a parser generated from it [20]. This formal framework could be one of the foundations for a more formal description of Diffingo's optimization mechanisms in the future.

DFDL. DFDL (Data Format Description Language) [7] is a standardized language for describing data formats, similar in concepts to PADS. The description of formats is captured in XML schema definitions (XSDs) annotated with wire-type information. In the XSDs, DFDL puts the internal structure of data items first; in the result, the language is not quite as close to a grammar specification as PADS' or Diffingo's languages. However, it supports similar concepts for the encoding of internal fields on-the-wire, which makes it possible to capture many existing data formats with DFDL. For example, it supports length- and delimiter-encoded fields, data dependent fields, computed variables based on an expression language, and value conversions through different pre-defined wire-format representations of internal types. DFDL is, however, not as easily extensible, because custom transformation functions (or an equivalent) cannot be applied. It is also not designed for handling compression or stream offsets.

Implementations of DFDL exist both in industry-scale from IBM (with a lot of tool support) [25] and as an open-source version called Daffodil for Java/Scala [14]. Compared to Diffingo, neither implementation targets in-network use cases, and, thus, they do not apply any of Diffingo's optimizations. So far, Daffodil only includes support for generating parsers dynamically during run-time and achieves throughputs of a few MB/s. The low performance is likely due to its beta state and the overhead of representing data items in XML types.

Construct. Construct [13] is a parser/serializer of data formats for Python. It is intended primarily for reading files or logs and uses mechanisms similar to parser combinators to define the structure of data items as object-oriented parsers/serializers in Python. Like Nail, its current implementation only supports binary formats (previously included text format support was removed due to poor performance). Construct's language supports

dependent fields, arrays (cf. lists), conditional choices, and stream offsets. Its parsers and serializers are executed in recursive-descent fashion and do not support look-ahead or non-deterministic choices. Like Daffodil, Construct does not target network functions, and thus doesn't perform any of Diffingo's optimizations.

7.1.3 Data serialization

ASN.1. ASN.1 (Abstract Syntax Notation 1) [27] is a standardized effort to describe the structure of data items and allow their serialization into standardized encodings. Conceptually, this is different from the data format description languages and tools we discussed so far, as ASN.1 captures only the variations in the logical structure of data items and not in their physical on-the-wire representation. While ASN.1 is useful to serialize and exchange data between applications or application parts, it is not well suited to describe data formats with already fixed on-the-wire representations, such as protocols or ad-hoc data formats.

Nevertheless, the language used to describe data item structures, again, has certain similarities to Diffingo: Its basic entities are typed fields, field sequences, repetitions, and type unions. However, the semantics are different: While Diffingo captures a grammar for parsing data formats, ASN.1 describes the logical layout of data items. For example, choices in Diffingo represent choices performed during parsing depending on values of other fields or look-aheads, type unions in ASN.1 represent different possible fields in the logical representation.

Protocol Buffers and alike. There are many other tools for serializing data items into a compact, standard format to facilitate communication between applications or application parts. The most prominent examples are Google's Protocol Buffers [23] and Apache Avro [2]. Both use a very compact and efficient wire-format to convert data items into. While the Protocol Buffers generate an API to read and write data items from/to this format at compile-time based on a schema that describes the layout of data items, Avro interprets a similar schema at runtime to provide a similar API. Schemata for Protocol Buffers are defined in a configuration language that supports required and optional fields, repeated fields, alternatives, and key/value maps. Avro uses JSON to describe schemata with similar compositional constructs. Both tools provide bindings for a large set of programming languages. Conceptually, they are very similar to ASN.1 (and thus, have equivalent similarities and difference to Diffingo), but only support a single, custom wire-format.

7.2 Parser generators

Parsing inputs is also a fundamental part of programming language, compiler, and interpreter design. Source code is parsed into an abstract syntax tree, which is then optimized and executed or used to generate executable files. Compared to the data formats described in the previous section, source code is almost always provided in text (ASCII) format. Its format is typically described in the form of a context-free or context-

sensitive grammar. The grammars for programming languages are often more complex than those for network protocols or ad-hoc data formats, but parsing performance is not as critical. For these reasons, there is a lot of interest in the programming language community for automatically generating parsers from language grammars.

Common tools used for this purpose are ANTLR (ANother Tool for Language Recognition) [40] and flex/bison [22, 16], the successors of lex and yacc (Yet Another Compiler Compiler) [47, 29]. In fact, we use flex and bison in our Diffingo prototype to parse the specification language.

These tools typically separate the parsing of source code files into two phases: In the first one, a scanner (or lexer) determines token boundaries, often based on regular expressions that define token formats. In the second phase, these tokens are processed by a parser generated from a grammar, which is specified in a form similar to the Backus–Naur Form (BNF). The grammars have similar mechanisms for composition of production rules as Diffingo, such as sequences, recursive rules for repetition, and non-deterministic choices. The parsers generated from such grammars by ANTLR use an LL(k) algorithm to implement parsing such a grammar, similar to Diffingo, which uses an LL(1) algorithm. Bison and yacc use LALR parsing algorithms.

In contrast to Diffingo, the grammars do not support data-dependent fields, stream offset fields, or similar features required to describe common binary formats and network protocols. A notable exception is an extension of ANTLR for binary file formats [48], which extends the context-free grammars used by ANTLR with attributes to capture such properties.

To build an internal representation of the parsed data, semantic actions can be defined for individual grammar production rules. These actions will be executed with the values of the tokens in a production rule when it is applied during parsing. Commonly, such actions create and add objects to an abstract syntax tree. This is a difference between these parser generators and systems such as Diffingo or Nail, where the format specifications determine both the serialized (wire-)format and the internal representation of data items.

7.3 Parser combinators

Similar to parser generators, parser combinators [11, 24, 33] aim to make it easy to create a parser for a new programming language. They are often used, for example, to enable a fast prototyping of domain-specific languages. Contrary to parser generators, though, parser combinator frameworks implement parsers directly in an (interpreted) host language by composing less complex parsers together to create more complex parsers. A parser is typically defined as a function (or object) in the host language, which accepts a string as input and returns an (object) structure (the internal representation of the parsed data). The framework provides higher-order combinator functions (parser combinators) that can be used to compose multiple such parsers together into a more complex parser. Typically, combinators for the common grammar production compositions are provided, such as sequences and choices. By implementing parsers and combinators in a general-purpose host language, parser combinator frameworks are very easily extensible

with custom combinators and parsing mechanisms. This makes it easy to extend such a framework, for example, with support for data-dependent productions.

The translation of parsed data into an internal representation is often handled in ways similar to those used for parser generators; that is, semantic actions can be defined that are executed by parsers during parsing. Again, this is different to Diffingo's message format specifications, which do not require such actions.

One of the most widely adopted parser combinator frameworks is Parsec [33]. It is implemented in Haskell and has spawned a multitude of clones for other languages. Parsec implements parsers as first-class entities of the underlying language and comes with combinators that allow the definition of context-sensitive grammars.

A common issue of parser combinators is their performance: They rely on the host language's support for the composition of functions or objects at runtime and typically employ a recursive-descent parsing algorithm with backtracking support for non-deterministic choices. To address this issue, recent work has applied staging [30] and macros [8] to eliminate the compositional overhead, which show promising results. The general idea behind both approaches is to automatically remove static control paths that occur in the composed parsers (using staging during just-in-time compilation at runtime in the former, and macros at compile-time in the latter).

An interesting topic of future work would be to investigate whether parser combinators optimized in such ways could be combined with some of the ideas presented in Diffingo: A more efficient memory management, incremental parsing support, or their optimization to specific use cases. It may, however, be more difficult to perform these optimizations, as the structure and behavior of parsers cannot easily be automatically determined: Developers are free to use arbitrary computation and specify custom semantic actions. However, it may be possible to detect common patterns at run-time, using techniques similar to staging and just-in-time compilation.

8 Conclusion

In this report, we presented Diffingo, our framework to generate parsers and serializers for network protocol messages from reusable message format specifications. Thereby, it encourages reuse of specifications between applications, improves application modularity, and reduces the risk of bugs.

Compared to its competitors, Diffingo aims to make auto-generated parsing and serializing code efficient by optimizing it for use in a particular network application, thereby utilizing the fact that many network applications only need access to a subset of the information contained in a message. Additionally, Diffingo applies memory management optimizations and supports incremental, resumable parsing and serializing of messages.

The results of our initial evaluation show that these optimizations are fruitful and that Diffingo's optimized parsers and serializers can compete with a hand-written parser and serializer. For medium to large message sizes, our optimized parsers and serializers show an almost identical performance to that of hand-written ones; but we notice some performance losses for very small messages, primarily due to our support for resumable parsing and serializing. This support, however, gives our parsers and serializers performance advantages when messages are processed incrementally.

While our current prototype implementation only supports messages with length-encoded fields, the general design of Diffingo is applicable both to messages with length- and delimiter-encoded fields. We also describe mechanisms for optimizing parsers and serializers of both kinds of messages for their use in a specific application.

In conclusion, we believe that we could show that the idea of generating high-performance parsers and serializers by adapting them to the needs of specific applications is a feasible one and that it deserves further attention.

8.1 Future Work

Our primary focus for future work is to address the limitations of our approach, implementation, and evaluation, which we discussed in detail in Section 6.2. This includes improving the expressiveness of the specification language, formalizing our optimization mechanisms, and extending our prototype implementation and its evaluation.

In particular, it is very important to evaluate our approach on a wider variety of use cases (including those that require delimiter-encoded fields) and in a less isolated fashion. For example, we would like to compare the effect on end-to-end application performance for multiple use cases. To better demonstrate the usefulness of incremental parsing/serializing, we also intend to show its benefit for much larger messages sizes than those used in the Memcached protocol. Furthermore, we plan to implement and evaluate an integration of our generated parsers with a user-level TCP stack to validate the perfor-

mance benefits that could be achieved by keeping irrelevant data in input buffers rather than internal data structures. To put our system in context, we also intend to compare the performance of its generated parsers and serializers with other parser/serializer generators, such as Spicy [46] and Nail [5].

Yet another aspect of our framework that has received little attention so far is the static analysis of applications to determine which fields the application accesses. We outlined a mechanism for this in Section 4.3. An implementation and evaluation of this mechanism for multiple use cases would confirm its applicability. As mentioned before, we are currently working on such an analysis for the Flick language [1].

A longer-term goal is the formalization of our optimization mechanisms to prove their correctness and thus ensure that all corner cases are supported correctly. We are also considering whether our approaches can be applied to parser combinators to further improve their performance, for example, through clever just-in-time compilation.

Bibliography

[1] Abdul Alim, Richard Clegg, Luo Mai, Lukas Rupprecht, Nik Sultana, Richard Mortier, Luis Oviedo, Masoud Koleni, Matteo Migliavacca, Paolo Costa, Peter Pietzuch, Alexander L. Wolf, Jon Crowcroft, Anil Madhavapeddy, and Andrew Moore. "Flick: Application-Specific Network Functions for Data Centres". In: *(under submission)*. 2015.

[2] *Apache Avro*. URL: https://avro.apache.org/ (visited on 2015-09-01).

[3] *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 2015-09-01).

[4] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Revised and expanded edition. Cambridge University Press, 1998. 544 pp. ISBN: 0-521-58390-X.

[5] Julian Bangert and Nickolai Zeldovich. "Nail: A Practical Tool for Parsing and Generating Data Formats". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 2014, pp. 615–628.

[6] Joseph Bates and Alon Lavie. "Recognizing Substrings of LR(K) Languages in Linear Time". In: *ACM Transactions on Programming Languages and Systems* 16.3 (May 1994), pp. 1051–1077. ISSN: 0164-0925. DOI: 10.1145/177492.177768.

[7] Michael J. Beckerle and Stephen M. Hanson. *Data Format Description Language (DFDL) V1.0 Specification*. Sept. 2014. URL: https://www.ogf.org/documents/GFD.207.pdf (visited on 2014-09-01).

[8] Eric Béguet and Manohar Jonnalagedda. "Accelerating Parser Combinators with Macros". In: *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM Press, 2014, pp. 7–17. ISBN: 978-1-4503-2868-5. DOI: 10.1145/2637647.2637653.

[9] James R. Bell. "Threaded Code". In: *Communications of the ACM* 16.6 (June 1973), pp. 370–372. ISSN: 0001-0782. DOI: 10.1145/362248.362270.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[11] William H. Burge. *Recursive Programming Techniques*. 1975.

[12] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. "Virtualizing the Network Forwarding Plane". In: *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. PRESTO '10. New York, NY, USA: ACM, 2010, 8:1–8:6. ISBN: 978-1-4503-0467-2. DOI: 10.1145/1921151.1921162.

[13] *Construct: A Parser and Builder for Binary Data*. URL: http://construct.readthedocs.org/en/latest/index.html (visited on 2015-09-01).

[14] *Daffodil: Open Source DFDL*. URL: https://opensource.ncsa.illinois.edu/confluence/display/DFDL/Daffodil:+Open+Source+DFDL (visited on 2015-09-01).

[15] Data Differential. *libMemcached*. URL: http://libmemcached.org/libMemcached.html (visited on 2015-09-01).

[16] Charles Donnelly and Richard Stallman. *GNU Bison - the Yacc-Compatible Parser Generator*. Jan. 23, 2015. URL: http://www.gnu.org/software/bison/manual/bison.pdf (visited on 2015-09-01).

[17] ETSI. *Network Functions Virtualization White Paper*. Oct. 2012.

[18] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol–http/1.1*. 1999.

[19] Kathleen Fisher and Robert Gruber. "PADS: A Domain-Specific Language for Processing Ad Hoc Data". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. PLDI '05. New York, NY, USA: ACM, 2005, pp. 295–304. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065046.

[20] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. "The Next 700 Data Description Languages". In: *ACM Sigplan Notices*. Vol. 41. ACM, 2006, pp. 2–15.

[21] Kathleen Fisher and David Walker. "The PADS Project: An Overview". In: *Proceedings of the 14th International Conference on Database Theory*. ICDT '11. New York, NY, USA: ACM, 2011, pp. 11–17. ISBN: 978-1-4503-0529-7. DOI: 10.1145/1938551.1938556.

[22] *Flex: The Fast Lexical Analyzer*. URL: http://flex.sourceforge.net/ (visited on 2015-09-01).

[23] Google. *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/ (visited on 2015-09-01).

[24] Graham Hutton. "Higher-Order Functions for Parsing". In: *Journal of Functional Programming* 2.3 (1992), pp. 323–343.

[25] *IBM: Get Started with the Data Format Description Language*. URL: http://www.ibm.com/developerworks/library/se-dfdl/ (visited on 2015-09-01).

[26] Intel. *Data Plane Development Kit (DPDK)*. URL: http://dpdk.org/ (visited on 2015-09-01).

[27] International Telecommunication Union. *Abstract Syntax Notation One (ASN.1) Standard (ITU-T Standard Documents X.680-X.695)*. Nov. 2008.

[28] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502. ISBN: 978-1-931971-09-6.

[29] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[30] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. "Staged Parser Combinators for Efficient Data Processing". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM Press, 2014, pp. 637–653. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660241.

[31] D. Joseph and I. Stoica. "Modeling Middleboxes". In: *IEEE Network* 22.5 (Sept. 2008), pp. 20–25. ISSN: 0890-8044. DOI: 10.1109/MNET.2008.4626228.

[32] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The Click Modular Router". In: *ACM Transactions on Computer Systems* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874.

[33] D. Leijen and E. Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Mar. 8, 2002.

[34] Anton Likhtarov, Rajesh Nishtala, Ryan McElroy, Hans Fugal, Andrii Grynenko, and Venkat Venkataramani. *Introducing Mcrouter: A Memcached Protocol Router for Scaling Memcached Deployments*. Facebook Code. URL: https://code.facebook.com/posts/296442737213493/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/ (visited on 2015-09-01).

[35] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "ClickOS and the Art of Network Function Virtualization". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 459–473. ISBN: 978-1-931971-09-6.

[36] *Memcached - a Distributed Memory Object Caching System*. URL: http://memcached.org/ (visited on 2015-09-01).

[37] *Memcached Wiki - BinaryProtocolRevamped*. URL: https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped (visited on 2015-09-01).

[38] Mauricio Osorio and Juan Antonio Navarro. "Decision Problem of Substrings in Context Free Languages." In: *CIC-X: Memorias del X Congreso Internacional de Computacion*. 2001, pp. 239–249.

[39] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. "Binpac: A Yacc for Writing Application Protocol Parsers". In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC'06)*. ACM, 2006, pp. 289–300.

[40] Terence J. Parr, T. J. Parr, and R. W. Quong. *ANTLR: A Predicated-ll(k) Parser Generator*. 1995.

[41] Vern Paxson. "Bro: A System for Detecting Network Intruders in Real-Time". In: *Computer Networks* 31.23–24 (Dec. 14, 1999), pp. 2435–2463. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(99)00112-7.

[42] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 227–240. ISBN: 978-1-931971-00-3.

[43] Manju Rajashekhar. *Twemproxy: A Fast, Light-Weight Proxy for Memcached*. Twitter Blogs. URL: https://blog.twitter.com/2012/twemproxy (visited on 2015-09-01).

[44] Eric Seckler. *Networking Meets Software: A Survey of Software Architecture and Engineering Challenges in Current Networking Topics*. Feb. 2015.

[45] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. "Design and Implementation of a Consolidated Middlebox Architecture". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 323–336. ISBN: 978-931971-92-8.

[46] Robin Sommer, Matthias Vallentin, Lorenzo De Carli, and Vern Paxson. "HILTI: An Abstract Execution Environment for Deep, Stateful Network Traffic Analysis". In: *Proceedings of the 14th ACM SIGCOMM conference on Internet measurement (IMC'14)*. ACM Press, 2014, pp. 461–474. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663735.

[47] *The LEX & YACC Page*. URL: http://dinosaur.compilertools.net/ (visited on 2015-09-01).

[48] William Underwood. "Grammar-Based Specification and Parsing of Binary File Formats". In: *International Journal of Digital Curation* 7.1 (Sept. 3, 2012), pp. 95–106. ISSN: 1746-8256. DOI: 10.2218/ijdc.v7i1.217.