Licentiate Thesis

A Language-Based Approach to Protocol Stack Implementation in Embedded Systems

Yan Wang
Technology

Studies from the School of Science and Technology at Örebro University 9
Örebro 2009

A Language-Based Approach to Protocol Stack Implementation in Embedded Systems

Studies from the School of Science and Technology
at Örebro University 9

Yan Wang

A Language-Based Approach to Protocol Stack Implementation in Embedded Systems

Abstract

Embedded network software has become increasingly interesting for both research and business as more and more networked embedded systems emerge. Well-known infrastructure protocol stacks are reimplemented on new emerging embedded hardware and software architectures. Also, newly designed or revised protocols are implemented in response to new application requirements. However, implementing protocol stacks for embedded systems remains a time-consuming and error-prone task due to the complexity and performance-critical nature of network software. It is even more so when targeting resource constrained embedded systems: implementations have to minimize energy consumption, memory usage and so on, while programming efficiency is needed to improve on time-to-market, scalability, maintainability and product evolution. Therefore, it is worth researching on how to make protocol stack implementations for embedded systems both easier and more likely to be correct within the resource limits.

In the work we present in this thesis, we take a language-based approach and aim to facilitate the implementation of protocol stacks while realizing performance demands and keeping energy consumption and memory usage within the constraints imposed by embedded systems. Language technology in the form of a type system, a runtime system and compiler transformations can then be used to generate efficient implementations. We define a domain-specific embedded language (DSEL), Implementation of Protocol Stacks (IPS), for declaratively describing overlaid protocol stacks. In IPS, a high-level packet specification is dually compiled into an internal data representation for protocol logic implementation, and packet processing methods which are then integrated into the dataflow framework of a protocol overlay specification. IPS then generates highly portable C code for various architectures from this source. We present the compilation framework for generating packet processing and protocol logic code, and a preliminary evaluation of our compiled code.

Yan Wang, Centre for Research on Embedded Systems (CERES),
School of Information Science, Computer and Electrical Engineering,
Halmstad University, Box 823, SE-301 18, Halmstad, Sweden.
Email: yan.wang@hh.se

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Verónica Gaspes for inspiring my research area and supporting me in the work I am going to present.

Likewise, thanks go to all the people who made it possible for me to carry out my research: my main supervisor, Prof. Thorsteinn Rögnvaldsson, my co-supervisor, Prof. Dimiter Driankov, as well as the company CEO Per-Arne Wiberg and his team at Free2move, for supporting our project.

Special thanks for Prof. John Hughes for generously joining my follow-up committee, and for giving important feedback to parts of my thesis.

I also want to thank Prof. Bertil Svensson and Prof. Magnus Jonsson for leading the Centre for Research on Embedded Systems (CERES) and making it a pleasant work environment.

Furthermore, I would like to thank all my friends and colleagues at the school of IDE at Halmstad University. It is a pleasure to work in such a nice atmosphere.

Last, but not least, hugs to my family, especially to my grandparents. They have supported me and encouraged me during all my life.

List of Publications

The publications which this thesis is based on are appended:

I Yan Wang and Verónica Gaspes, “A Library for Processing Ad hoc Data in Haskell – Embedding a Data Description Language”, in *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)*, Hatfield, United Kingdom, September 2008. To be published by Springer Verlag in the Lecture Notes in Computer Science (LNCS) Series.

II Yan Wang and Verónica Gaspes, “A Domain Specific Approach to Network Software Architecture – Assuring Conformance Between Architecture and Code”. To appear in *Proceedings of the 4th International Conference on Digital Telecommunications (ICDT 2009)*, Colmar, France, July 2009.

List of additional publications:

- Yan Wang, “Implementation of Protocol Stacks”, *Technical Report 0746*, School of Information Science, Computer and Electrical Engineering, Halmstad University, Halmstad, Sweden, June 2007.
- Yan Wang, “IPS: Implementation of Protocol Stacks for Embedded Systems”, in *the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 2007)*, Doctoral Colloquium, Sydney, Australia, November 2007.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In both research and business enterprises that deal with networked embedded systems, the implementation of protocol stacks is a central issue. Implementing protocol stacks is tedious, error-prone and time-consuming due to the complex and performance-critical nature of network software. The specifications of most modern protocols are quite large. Also, specifications are often not mapped into code in a straightforward manner, which makes it difficult to both achieve and check correctness. In addition, a range of mature optimization techniques designed to make protocol code more efficient tend to make implementations more complicated and are new sources of errors. This is even more so when targeting embedded systems, e.g., wireless sensor network nodes, where additional, non-functional constraints come into play: implementations have to minimize code size, energy consumption, memory usage, and other computation resources. In order to improve on time-to-market, scalability, maintainability and product evolution, even programming efficiency is relevant.

Therefore, we find it relevant to investigate how to provide program development support for protocol stack implementation. This opens opportunities for a language-based approach in the form of a Domain-Specific Language (DSL) [Hud97, MHS03]. Domain-specific languages have proved their ability to support code consistency, performance, systematic code reuse, and portability, in the development of software for various areas, such as financial products [JES00], communication services [CHR$^+$03], hardware device design [BCSS98], cryptographic algorithms [Lew07] and network protocols [Mcg04, BMvE98, CRL96]. Based on domain-specific knowledge, a DSL provides domain abstractions and notations, and thereby allows domain-oriented compiler optimizations, constraint enforcement, and language-level debugging in a friendly way for the domain experts.

This thesis describes ongoing work which aims at making the process of implementing protocol stacks easier, while realizing performance demands and keeping energy consumption and memory usage within the context of resource constrained embedded systems. Our work is conducted in terms of a Domain-Specific Embedded Language (DSEL) [Hud97], *Implementation of Protocol Stacks* (IPS) with focus of correctness, efficiency and maintainability. IPS combines a data description language for specifying protocol packets with a framework for constructing overlaid protocol stacks. Running an IPS description generates C code which considers the non-functional requirements for embedded systems such as execution time and memory usage. We have decided to limit the scope of the computations by addressing asynchronous message-passing protocols on single processor distributed real-time embedded systems.

1.1 Contributions

IPS is partially based on the Data Description Calculus (DDC) [FMW06] which uses types from dependent type theory to describe various forms of ad-hoc data. Types in DDC are interpreted as parsing functions that produce both internal representations of the external data and parse descriptors pinpointing errors in the original source. In our approach, a high-level packet specification is similar to a type in DDC and is *compiled* into two parts: 1.) an internal representation, e.g., a C data type, and 2.) packet processing methods, i.e. marshaling and parsing. The internal representation can be integrated into the implementation of protocol logic, while the packet processing program can be combined into the dataflow framework generated from protocol overlay specifications. This is clearly an innovation and promises three advantages:

- ease of specification,
- increased modularity and reuse of code,
- and correctness by sound implementation methodology.

With our language, the protocol stack implementors now can declaratively specify and construct a complex protocol stack, using simple and explicit high-level abstractions, and automatically generate highly portable C code for various architectures from this source.

The contributions are:

- A formal notation for the specification of packet formats using dependent types. In our notation, the programmer can specify physical organization, dependencies among field contents and constraints over the values of some fields (Paper II).
- A dual compilation of packet type descriptions, which generates an internal data representation and a packet processing library of parsing and marshaling operations (Paper II).

- A framework for implementing protocol stacks by integrating other code into packet descriptions (Chapter 3).
- A contribution to the functional programming language community in the form of a library for processing ad-hoc data formats. This is a generalization of our language for describing packets (Paper I).

1.2 Outline of the thesis

The remainder of this thesis is structured as follows: Chapter 2 explains the background for our work. We briefly characterize embedded systems and the specific challenges involved in developing their software. Our particular area of interest, network software and protocol development, is discussed in more detail. Then we give background on domain-specific (embedded) languages, and also discuss related work in DSLs for network software. Chapter 3 presents the domain-specific embedded language IPS which we have developed. We describe details about the protocol stack specification in IPS: notations and meaning of the language constructs for specifying protocols and their relationship in a layered hierarchy. We discuss the design decisions we have taken and also include the preliminary evaluation results. Chapter 4 concludes with a summary and directions for future work.

Chapter 2

Background and Motivation

2.1 Embedded software

An embedded system is a combination of computer circuitry and software that is built into a product for purposes such as control, monitoring and communication. Whereas embedded systems of the past were usually realized mostly in hardware, nowadays advances in chip technology have made it possible to program complex and pervasive software. From portable wireless devices like sensor nodes to large fixed installations like controlling systems for large chemical plants, embedded software appears everywhere. It has taken over what mechanical and dedicated electronic systems used to do in the past. It has thus become an application area of increasing importance [Lee00].

Embedded software is often developed under constraints, since embedded systems are usually resource constrained. For example, wireless sensor nodes only have a few kilobytes of RAM and code space, and they use batteries. Their software has to be designed for low energy consumption and low memory usage. In addition, embedded software development often has extended correctness and safety requirements: an embedded system might be controlling a machine that is expected to run continuously for months or even years without human intervention. The software of such systems has to be developed and tested or verified much more carefully than general-purpose computer software would require.

Embedded network software is becoming interesting for research and business as networks of embedded systems are emerging as platforms for a growing number of applications. Embedded network software, most notoriously communication protocol stacks for embedded systems, are an ever-ongoing area of interest: many companies are re-implementing well-known infrastructure protocols on new emerging embedded hardware and software architectures; new protocols are designed and implemented in response to new application requirements and revising protocol specifications in industry standards. Thus, implementing protocol stacks for embedded systems is an ongoing process. The increasingly complex tasks along with the limited resource usage imposed by hardware or the environment make embedded network software harder to implement than desktop variants. In consequence, programmers have to deal with too many implementation-level details besides the programming logic which is their normal and original focus. Therefore, it is worth researching on how to make embedded network software implementations both easier and more likely to be correct within the resource limits.

2.2 Network software

Network software is typically organized according to a layered architecture to make design, implementation, evaluation and reuse easier. Complex functionality of network software is divided into smaller individual pieces, called *protocols*, which can be hierarchically layered into a *protocol stack* to realize more complex functionality.

2.2.1 Protocols

A protocol is defined by its specification which can be broken into two conceptual parts, called *packet specification* and *protocol logic specification.*

A *packet specification* describes messages that computers on the network use for communication, which includes the information to be sent to the receiving computer, e.g., the address of that computer, and how the message is coded, e.g., the first byte of the message is used to store the sender's address, the second one is used to store the receiver's address. The packet specification describes the physical organization of a message in terms of the *packet format*, usually presented in figures, and specifies dependencies among field contents as well as constraints over the values of some fields, usually by informal explanations written in a natural language. Fig. 2.1 shows the original TCP packet format description [RFC] as an example, reproduced in the usual text-only style of the RFCs in the networking community. Physical packet layout is specified as bit-length for all header fields in the figure. For example, the `source port number` takes 16 bits. Additional constraints and fields dependencies are specified in the accompanying text. For example, constraint on field `Reserved` is:

> "Reserved for future use. Must be zero."

Dependency between field `ACK control` and field `Acknowledgment Number` is:

> "If the `ACK control` bit is set `this field` contains the value of
> the next sequence number the sender of the segment is expecting
> to receive."

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                        TCP Header Format
```

Figure 2.1: TCP packet specification (from RFC793)

According to this informally described packet specifications, protocol implementors write packet processing code that includes parsing and marshaling packets. Writing packet processing code is not hard, but rather tedious, time-consuming and error-prone. The incoming packets are dealt with as sequences of bits that have to be interpreted according to the specification. The outgoing packets have to be aligned to header fields, at the same time converting between byte order in the network device and the processor. This is most frequently done in the C programming language, using offsets, bit masks, dedicated functions and other low-level operations, and this lack of abstraction is a common source of errors. For example, Figure 2.2 shows an abstracted form of the packet processing code (`uip.c`) from uIP [Dun07]. The code first computes the TCP/IP packet header by mapping a TCP/IP header structure `uip_tcpip_hdr` to the raw buffer storing incoming packets. Then it checks the size of the packet. If its size is less than the size reported in its IP header, this packet is assumed as a corrupted packet in transit which has to be dropped. Then it subtracts the IP datagram offset to get the `fragment flag`. To calculate the checksum of TCP header, it uses `htons` explicitly converted from host byte order to network byte order.

It is even worse that code fragments like this might occur more or less anywhere in the code implementing the whole protocol stack. This makes it difficult to trace the implementation from the specification, which is nevertheless highly structured. All this makes the program hard to read by anyone

```
#define BUF ((struct uip_tcpip_hdr *)&uip_buf[UIP_LLH_LEN])
... ...
  if((BUF->len[0] << 8) + BUF->len[1] <= uip_len) {
  ... ...
  } else {
    UIP_LOG("ip: packet shorter than reported in IP header.");
    goto drop;
  }
... ...
  /* Check the fragment flag. */
  if((BUF->ipoffset[0] & 0x3f) != 0 ||
     BUF->ipoffset[1] != 0) {
... ...
  /* Sum TCP header and data. */
  sum = chksum(sum, &uip_buf[UIP_IPH_LEN + UIP_LLH_LEN],
       upper_layer_len);
  return (sum == 0) ? 0xffff : htons(sum);
```

Figure 2.2: Code extract from uIP: *uip.c* (abstracted)

other than the original authors, hard to modify in case of slight modifications in packet specifications, and hard to check and recover errors.

Modern approaches to address these challenges are *Data Description Languages* (DDLs) based on type theory, such as DATASCRIPT [Bac02], PACKETTYPES [MC00] and PADS [FG05]: high-level languages for describing and processing domain-specific data – in our case, the binary data associated with protocols. They allow programmers to describe the physical layout of data and its semantic properties, e.g., range constraints on values, in a format specification. This specification is used to automatically generate data processing implementations as well as other tools, e.g., printing libraries, in different target languages. Since in DDLs the low-level implementation details are hidden from the programmers, they are able to produce reliable code. Also, as the format specification for data in these languages is intuitive and expressive, it can serve as high-level documentation that is more readable and maintainable than conventional packet processing code written in low-level languages. Another benefit is the built-in consistency of all related code, i.e., changes have to be made only once in the source specification, and the corresponding derived implementations will be changed automatically.

The second aspect of a protocol specification is the *protocol logic specification* which defines rules of operations for governing interactions between communicating peers, e.g., authentication, error detection and congestion control. The rules include how a protocol interacts with adjacent protocols, and how the protocol interacts with its peer protocol in the same layer. For example, in RFC793[RFC],

> "The TCP/user interface provides for calls made by the user on the TCP to OPEN or CLOSE a connection, to SEND or RECEIVE data, or to obtain STATUS about a connection."

is the first kind of specification. And an example of the second kind is:

> "The "three-way handshake" is the procedure used to establish a connection. This procedure normally is initiated by one TCP and responded to by another TCP."

Conceptually, protocol logic can be represented with a state machine, in terms of a set of states and actions. It changes states either by self initiated actions, e.g., sending messages, setting timers, or by reacting to stimuli, e.g., responding to interface calls, receiving messages and timer events. For example, when an application makes a connection request, the TCP sender state machine initializes by picking an unused initial sequence number, goes to the so-called SYN-SENT state, and sends a SYN message. The states are implemented by local states, e.g., a sequence number of a TCP connection. The actions are implemented as handlers for processing events passed from adjacent layers (e.g., a timeout-triggered retransmission), and utility functions that handlers may call (e.g., fragment and later reassemble packets).

2.2.2 Layering

Protocols are elegantly designed in layers where each protocol explicitly declares some assumptions made about other protocols and performs some particular function independently of others. Protocols are then successively layered into a protocol stack to build more complex functionality. The hierarchy of protocols need not be linear. A given protocol may support multiple higher-layer protocols and may use several communication services provided by lower-layer protocols. Hence the protocols on a host form a *protocol graph*. For example, TCP/IP is normally considered to be a 4-layer system, as shown in Figure2.3 [Ste94]. The `link` layer at the bottom includes the device driver in the operating system and the corresponding network interface card in the computer, which handle all the hardware details of physically interfacing with the media, e.g., cable, radio, etc. The `network` layer operates on packets moved around between peers in the network. The `transport` layer provides a flow of data between two end-points (hosts), for the application layer above. The `application` layer handles the details of the particular applications.

To maximize efficiency, or to meet the constraints imposed by hardware limits, the programmers implement the protocol stack in a monolithic way. For each protocol the assumptions about the existence and the details of the design of related protocols in the stack are made and the programming logic is deeply woven into the fabric of the whole implementation. The IP implementation uIP [Dun07] for embedded systems is an example for implementing

Application	Telnet, FTP, e-mail, etc.
Transport	TCP, UDP
Network	IP, ICMP, IGMP
Link	device driver and interface card

Figure 2.3: The four layers of the TCP/IP protocol suite

TCP/IP as a *monolithic stack*, i.e. without any layer structure. The code is extremely tightly coupled and certain mechanisms in the interface between the application and the stack are removed to minimize the resulting code size. However, this approach quickly turns out to be impractical, because of growing interdependencies of performance and functionality aspects, and as a result of the inherent non-scalability of such programming. Moreover, programming in this way is exceedingly difficult even for experienced programmers.

In *layered protocol stack* implementations, distinct layers are responsible for different facets of the communication. Protocols can be implemented in a modular fashion to reduce complexity and to make reuse and configuration possible. Individual layers are easier to both write and verify than complex, monolithic stacks. Since the dependencies among protocols are explicit, it is easy to build special-purpose protocol stacks by assembling existing layers, while it is often difficult to tailor a monolithic stack to specific application requirements without rewriting large portions. In a layered protocol stack implementation, any unneeded layer can simply be removed to reduce the surplus functionality and tailor the networking parts to application needs. For example, for applications that do not require reliable transmission of messages, TCP can be removed.

Well-understood, the price for these advantages when following the modular layer structure in the implementation is a considerably increased overhead in protocol processing. Protocol boundaries in the stack will impose a number of separate function calls, and inhibit direct access to other layer's internal data structures. The challenge is to combine the modularity of layered protocol stacks with the good performance of monolithic protocol stacks. The solutions which have been proposed to this problem always take one of two approaches.

System-based approaches provide systematic software architectures for constructing protocol stacks, in order to clearly express the modular structures and protocol layer composition. Following such a system-based optimization approach will produce a well-structured implementation, while enabling a few particular optimizations which are specific to protocol stacks. For instance, Integrated Layer Processing[CT90] advocates rewriting of existing protocol stacks such that the processing done by the different layers is pipelined, avoiding message copies every time a layer boundary is crossed. X-kernel[HP91] uses a particular thread concept: an initial up-call and one reserved thread per message. Chameleon [DOH07] is an architecture for organizing protocol stacks in sensor networks, which aims at isolating low-level packet processing from other aspects of protocol stack implementation. Headers are removed and attributes for further package processing are introduced.

Language-based approaches go one step further and use compilation techniques to reduce the performance penalty for protocol layering. Using a compiler is an advantage, because it can automatically perform protocol-oriented optimizations based on specific common behaviors of protocols. For example, the last action taken in a message output function of a protocol is always to invoke the next lower layer's output function. When the lower output function returns, the original output function is done and also returns. Responding to this behavior, Morpheus' *short-circuit return* optimization [AP93] saves one jump assembler instruction per protocol layer, i.e., the output functions with no further work are bypassed in the sequence of procedure returns. For the lower output function, there is no need to jump back to the current output function after it finishes it task, instead it should jump back to the current one's caller. Another example is that protocol stack processing in some cases consists of frequently executed code segments. For instance, one message output function might always call another message output function in the next lower layer. Promela++ [BMvE98] offers programmer-annotated abstractions for explicitly specifying this kind of execution path as a *fast path*, for a more efficient composition of multiple protocol layers. According to *fast path*, procedure calls can be inlined to form a tightly packed single function which removes almost all call overheads. It also enables further optimizations by the C compiler since inlining allows the C compiler to optimize across function boundaries using standard techniques, such as copy propagation and constant folding.

That said, it is clear that compilation techniques and language-centric development should be employed in order to produce reliable and efficient code for the domain of network software. Taking this language-based approach further means to create an entire language that is especially tailored to the needs of protocol stack implementations, a *Domain-Specific Language.*

2.3 Domain-specific languages

A *Domain-Specific Language* (DSL) [Hud97, MHS03] is a programming language that closely models a particular application domain or problem of knowledge or expertise, where the concepts are tied to the constructs of the language. Since the program written in a DSL is at the level of abstraction of the application domain, it is expressive, concise and self-explanatory. It can serve as high-level documentation which is easier to write, read and maintain than an equivalent program written in a general purpose language. And the users will more likely be domain experts, rather than skilled programmers. Domain specific languages have long been considered as an approach to software engineering [Hud96]:

> I have believed for a very long time that *abstraction* is the most important factor in writing good software. As programming language researchers we design, and as software engineers we are trained to use, a variety of abstraction mechanisms: abstract data types, higher-order functions, ... we might ask what is the "ideal" abstraction for a particular application is. In my opinion, it is a programming language that is designed precisely for that application: one in which a person can quickly and effectively develop a complete software system. It is not general at all; it should capture precisely the semantics of the application domain — no more and no less. In my opinion, a domain-specific language is the "ultimate abstraction".

2.3.1 Why a domain-specific language?

We are going to discuss advantages of domain-specific languages, with a particular focus on network software and protocol stack implementation.

First, a DSL offers an appropriate suite of high-level constructs and abstractions with natural vocabulary for manipulating specific concepts in domain expertise (in our case of network protocols, buffers, timers, and packets are such concepts, rarely supported by general purpose languages). Consequently, programs are more expressive, and also more straightforward, i.e. easier to read and develop.

Second, a DSL offers guidelines and built-in functionality for constructing programs. The support routines can either be an integral part of a language, as language primitives, or, in some cases, not visible at the source code level but instead automatically applied where needed. The compiler generates implementation details by both converting the explicitly specified code and exploring the hidden common program patterns, avoiding technical boilerplate code. For example, a compiler for a protocol implementation language might be able to generate the appropriate locking transparently for multiprocessing, so that protocol source code is independent of the degree and style of multiprocessing. DSLs realize a clear and elegant way to write better programs with less code, and less code also means less programming space for errors.

Third, DSLs are a perfect medium to enforce constraints, because they increase the amount of context available to the compiler. The entities satisfying some constraints have common behaviors, which effectively raises the abstraction level. The compiler can automatically supply more of the code and data structures as well as make implementation decisions that the programmer would otherwise have to specify. In the protocol development domain, we can often find generic functions shared by different protocols. For example, protocols fragment and reassemble packets: any TCP segment whose size is greater than the Maximum Segment Size(MSS) needs to be fragmented into smaller pieces, and reassembled upon receipt. IP does exactly the same, using the Maximum Transmission Unit(MTU) to limit the size of its IP datagrams. This commonality can be realized by types in a DSL, i.e., a protocol of type *Fragmentable* has to specify its *maximum fragment size*. The compiler can check that the implementation actually conform to types, as well as use utility functions to fragment and reassemble implicitly.

Fourth, DSLs can restrict the design space to contribute to a good implementation discipline. General-purpose languages, in contrast, would allow programmers to use their own algorithm in place of a support routine and thereby circumvent domain-specific constraints on the data structures. And of course, all kinds of subtle mistakes can happen when doing so. For instance, the out-of-bounds array access, a typical programming error, is likely to happen when manually extracting erroneous values out of a packet field in the buffer, due to the flexibility on accessing memory. A DSL avoids it by design, because it will not offer any explicit low-level memory access or pointer arithmetic.

Fifth, the domain-specific constructs, along with the enforced design philosophy, put the compiler in a better position to make optimizations that are specific to the domain of compilation. As we have already mentioned earlier in Section 2.2.2, protocol-oriented optimizations can be achieved automatically. In contrast, conventional languages and their optimization techniques only offer limited opportunities related to protocol stacks and protocol-specific optimizations in general can only be achieved manually.

Sixth, DSLs can guarantee syntactically and semantically unambiguous formal descriptions of the domain. It enables properties of programs to be verifiable at the domain level, which highlights the power of a DSL linked with formal methods. For example, if the protocol developer is allowed to specify the control flow of a protocol layer in a formal semantic model, the specification can be verified by a validator against program-specified safety requirements. In particular, the programmer can specify that the protocol never deadlocks by requiring that some particular state in the protocol is reached infinitely often [BMvE98].

2.3.2 DSLs for protocol stack implementations

During the past decade, DSLs have been studied extensively in the protocol development domain. Some of them decompose complex protocols into modules and enforce such design philosophy using language constructs for either ease of programming [KKM99] or reusability and optimizations [AP93]. Others have focused on both verification and code generation. They primarily aim at making correctness verification as easy as implementation. The typical approach to this, taken by TAP [Mcg04], Teapot [CRL96] and Promela++ [BMvE98], is to have two execution models: one for model checking, one for generating executable code. TAP [Mcg04] is effective to describe asynchronous message-passing network protocols. Teapot [CRL96] has been designed for writing cache coherence protocols. Both of them heavily specialize for one particular protocol category and ignore the protocol construction handling. Promela++ [BMvE98] is a more closely related work, which provides explicit language mechanisms to encapsulate and compose protocol layers where the adjacent layers communicate neatly using FIFO message queues. However, the downside of this scheme is a time-consuming context switch between the communicating processes which could overburden the runtime memory. Promela++ does not support some necessary primitives, e.g., timers and memory allocation, thus its source code has to include blocks of C code when needed.

2.3.3 Domain-specific embedded languages

It is fairly difficult and time-consuming to design and implement a DSL from scratch. Designing the grammar, writing a parser, code generator or interpreter, are all challenging tasks. Much of the startup cost of creating a new DSL goes to its non-domain specific parts, e.g., variables and arithmetic types, which are necessary for almost all programming languages. Building on this base, the domain of interest can be further tailored. During the procedure of development, a DSL tends to grow, since more and more new modules, procedures, and data structures will be added. Consequently, all kinds of difficulties associated with these evolutions will be invoked. For example, each time new functionality is added in the DSL, the new syntax might be added as well followed by modifying the parser and so on. Moreover, in many cases, a DSL is designed quickly, started with modest goals, but after a long evolution, ends up as a complex general purpose language.

A *Domain-Specific Embedded Language* (DSEL) is a shortcut to create a DSL [Hud97]. It embeds domain-specific concepts and features into an existing programming language (a host language which is normally general-purpose and fully fledged), to take advantage of its implementation and tools. In more detail, it reuses design decisions as well as syntax and semantics for non-domain specific aspects from the host language; it defines specific abstract data types and operators in the host language. The host language with new added functions is extended to the new language where a problem in a domain can be described with new constructs. Therefore it has all the power of the host language. Since a DSEL only needs to focus on domain-specific issues and is able to use the host's tools, e.g., type checker and compiler, implementation of a DSEL can be carried out much faster than a full DSL. In addition, DSELs effectively realize the DSL methodology. Domain terms can be directly translated into code in the DSEL, which give a similar look-and-feel of special syntax as a DSL does. And the correct semantics is made as the domain experts expect from the language, because new functionalities are added to the DSEL such as functions and procedures, which are seamlessly integrated into the former part of the language. This makes extensions for a DSEL easier than for a stand-alone DSL, and leads to reduced maintenance costs.

To our knowledge, so far there is no research apart from ours which targets DSELs on network software development.

2.4 Summary

Based on our survey of both network protocols and programming language design, we believe that exploiting the capabilities of domain-specific languages will allow straightforward and correct-by-constrain designed implementation of network protocol stacks. It will provide powerful program development support, meet the challenges and realize infrastructure automatically. Optimally, this language should be robust, expressive, and easily usable, so that it can be integrated into the protocol development and standardization processes.

Chapter 3

Implementation of Protocol Stacks

This chapter covers four topics. We first introduce the domain-specific language IPS that we have developed (Section 3.1) which provides a framework for implementing protocols and protocol stacks with overlay features. Then we discuss the major design decisions of IPS in Section 3.2. The preliminary evaluation is shown in Section 3.3 which underlines IPS's practical usability. Finally, we discuss limitations of IPS. We illustrate all of these with Rime [DOH07], a lightweight layered communication stack for sensor networks sketched in Figure 3.1. Since its primitives are designed in a strictly layered fashion, it is suitable to be implemented in our language.

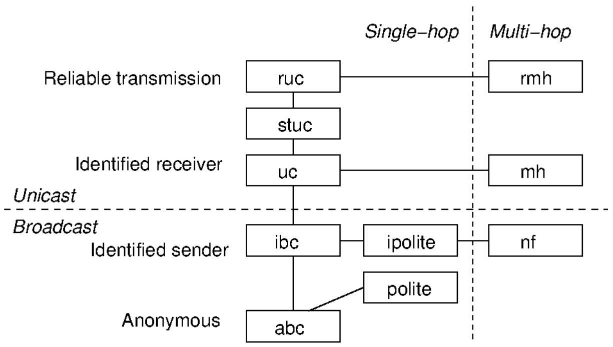

Figure 3.1: The communication primitives in the Rime stack

3.1 IPS specifications

We have developed the domain-specific language IPS (Implementation of Protocol Stacks) which employs a framework for describing protocols and overlaying them into protocol stacks. Further, protocol stacks can be combined into protocol graphs in an concise and structured way. IPS has been designed to be a small language that captures several core features of protocol stack implementation:

- packets can be specified by packet formats including physical layout, dependency of fields and semantic constraints,
- implementation details belonging to different protocols can be described in isolation,
- packet processing is dealt with separately from protocol logic,
- assumptions made about related protocols can be declared explicitly,
- individual protocols can be overlaid to build protocol stacks,
- protocol stacks can be collected together to build protocol graphs for the host computer.

The resulting descriptions are modular and easy to trace back to the protocol stack specifications.

3.1.1 Protocols

IPS supports protocol development by providing a high level of abstraction. The implementations of protocols are concise in the sense that protocols are expressed with few statements and declarations, reducing verbosity by hiding implementation details and making programs easier to read, understand and debug. The programmer has to provide a few implementation details to specify the simplest possible protocol:

- *name* is the name of the protocol,
- *packet* is the packet format specification,
- *bufferin* is the buffer where the incoming packets are stored,
- *bufferout* is the buffer where the outgoing packets are stored,
- *send* is the sending function,
- *receive* is the receiving function.

Figure 3.2 shows the implementation in IPS of the protocol `ibc` (identified best-effort single-hop broadcast) which is used to identify the sender in the Rime stack. Its specification, quoted from [DOH07], is as follows:

> All packets, both outgoing and incoming, are stored in a single buffer, called the Rime buffer... ... The ibc primitive adds the single-hop sender address as a packet attribute to outgoing packets. All Rime primitives that need the identity of the sender in the outgoing packets use the ibc primitive, either directly or indirectly through any of the other communication primitives that are based on the ibc primitive.

```
ibc :: Protocol
ibc = protocol{
  name = "ibc",
  packet = header0:header1:payload,
  bufferin = "rimebuf",
  bufferout = "rimebuf",
  send = ibcsend,
  receive = end
}
   where header0 = int 0 2 |* constraint
         constraint x = (x==*0)||*(x==*1)
         header1 = int 1 16
         localAddr = cterm "rimeaddr_node_addr.u16[0]"
         ibcsend = (ifE (upperProtocolIs "uc")
                      {-then-} (header0 =* 1)
                      {-else-} (header0 =* 0))
                   : header1 =* localAddr
                   : gotoSap 0
                   : end
```

Figure 3.2: Ibc protocol implementation in IPS

`Ibc` has type `Protocol`. The definition of `ibc` is introduced by the keyword `protocol` followed by comma-separated component definitions enclosed by curly brackets. `Name` is the identity of a protocol used to be recognized by others. For example, `ibc` implementation has protocol name ‘‘`ibc`’’. Packets are one of the fundamental protocol abstractions and thus an integral part of the language (specified by `packet`). A packet is a sequence of header fields followed by a payload. For example, an `ibc` packet has a header including two fields (`header0` and `header1`), and a payload (`payload`). The packet format can be described both syntactically and semantically. That is, the header fields can specify both physical organization, dependencies among field contents and constraints over the values of the fields. All this is explained in more detail in the appended Paper II. For example, the first header field `header0` is a field which has type `int`, id `0` and size `2`(bits). It is used to route incoming packets to the upper encapsulated protocols, the `2` bits allow for 4 possible protocols to be located on top of `ibc`. `Header1` has type `int`, id `1` and size `16`(bits) which is used to specify the address of the packet sender. `Constraint` is a function used to impose a constraint on the content of a field. In our example, the value of `header0` should be either `0` or `1` because in the Rime stack `ibc` only has two upper protocols so far. The location of the incoming and outgoing packet is specified by `bufferin` and `bufferout` respectively. For example, a single buffer `rimebuf` is used for both incoming and outgoing packets to reduce memory footprint, and should be consistent over a whole protocol stack (as explained later in Section 3.1.2). A protocol shows how to transmit its packets via its lower-layer protocols by specifying a `send` function, and how to pass its receiving packets to the upper-layer protocols by specifying a `receive` function. In `send`, if an outgoing packet comes from the upper protocol `‘‘uc’’`, `header0` is assigned `1`, otherwise `0`. The `send` also adds sender information to the transmission by assigning the sender node's local address `localAddr` to field `header1` for outgoing packets stored as the first element of a C array as `rimeaddr_node_addr.u16[0]` which is prefixed by the keyword `cterm`. Then the packet is passed to the next lower layer through `gotoSap 0`. `Sap` (service access point) is the interface between two adjacent protocols. It specifies that the upper protocol is connected to the lower protocol by an indexed logical channel, e.g., in the case of `ibc`, `0` is the index of its lower-layer protocol `abc`. Here we can see that IPS offers a seamless model for thinking about packets: packets can be specified by packet formats and the fields of a packet can be referred in the operations directly. When a packet is received by the `ibc`, it immediately passes the packet to the upper layer. In other words, the receiving function `receive` does nothing and consists only of the mandatory `end` keyword. The decisions regarding which upper-layer protocol should be used will be explained Section 3.1.2.

The basic protocol type `Protocol` can be extended to new protocol types where new state information is added by using additional declarations and the protocol behaviors are extended by additional procedure code. For example, a protocol providing reliable transmission must have separate execution of retransmission and timers to invoke the retransmission regularly. In the Rime stack, the protocol `stuc` (stubborn single-hop) gives this kind of service which repeatly sends a packet to a single-hop neighbor. As Figure 3.3 shows, `stuc` has protocol type `RetransmissionProtocol` and it has to specify a timer `retransmissiontimer` with id `0` and the expire interval `200`(ms) as well as a retransmission function `retransmission` with parameters `parameters` and function body `functionbody`.

IPS allows code reuse by overwriting part of the specification. For example, in the Rime stack, the protocol `ipolite` (identified polite single-hop broadcasts) works in the same way as the protocol `polite` but located on the top of `ibc` to identify the sender. It can be realized by reusing the existing `polite` implementation and only overwriting the protocol `name` as shown in Figure 3.4.

```
stuc :: RetransmissionProtocol
stuc = retransmissionprotocol{
  name = "stuc",
  bufferin = "rimebuf",
  bufferout = "rimebuf",
  packet = header0:payload,
  send = stucsend,
  receive = stucreceive,
  retransmissiontimer = stuctimer,
  retransmission = function0
}
   where ... ...
         stuctimer  = timer 0 200
         function0  = function 0 parameters functionbody
         ... ...
```

Figure 3.3: Stuc protocol implementation in IPS

```
import Polite(polite)
ipolite :: Protocol
ipolite = polite{
  name = "ipolite"
}
```

Figure 3.4: Ipolite protocol implementation in IPS

3.1.2 Protocol stacks

In IPS, any new protocol type, e.g., `RetransmissionProtocol`, is derived from `Protocol` which means that any protocol automatically satisfies `Protocol`. Therefore, all the protocol abstractions have a uniform protocol interface provided by `Protocol`, and can be composed as building-blocks. As a protocol stack is the combination of different protocols at various layers, IPS provides a basic combinator `<|>` to build protocol stacks. It provides a way to overlay protocols in a strictly linear way, as a proper "stack". Introduced by the keyword `stack`, a number of protocols are enumerated to form a top-down stack. For example, with the definitions of `abc`, `ibc` and `polite`, we can now proceed to the description of a protocol stack. In the example in Fig 3.5, we overlay `ipolite` on top of `ibc`, and add a third protocol `abc` below where the `abc` protocol is the basic protocol of the Rime stack.

Aside from the protocols, a protocol stack specification includes *constraints* and an index of *service access point* specified as arguments with each protocol. The encapsulation relationship between protocols is specified in the *constraints* on field values of their packets. It is used to multiplex incoming packets to the corresponding upper-layer protocols. For example, `ibc` has been overlaid with `ipolite`, the first header field of `ibc`, `header ibc 0`, should have value `0`. If this field of an incoming `ibc` packet is `0`, it will be passed to `ipolite`. And as

```
stack1 = stack (protocol ipolite [] -1)
               <|>
               (protocol ibc [(header ibc 0)|* (\x->x==*0)] 0)
               <|>
               (protocol abc [(header abc 0)|* (\x->x==*0)] 0)
```

Figure 3.5: A protocol stack implementation in IPS

mentioned before, *service access point* (sap) is the logical channel connect to protocols which should keep consistent in both protocol implementation and protocol stack composition. For example, `abc` is the lower-layer protocol of `ibc` connected by `ibc`'s sap 0. In the `ibc` implementation in Figure 3.2, `gotoSap 0` means passing packets to `abc`.

The `ipolite` is located in the topmost layer: no constraints (by `[]`) and no service access points apply(by an arbitrary negative number -1). The `ibc` below `polite` specifies a constraint that the first header value of the incoming packet (`header0` earlier) must be `0`. And `ibc` is connected to `polite` by service access point `0`. Below `ibc`, we find the basic protocol `abc`, with a similar constraint, and using service access point `0` of `ibc`. In the example of Rime, the total protocol stack is organised in a tree rooted at `abc`. This means, every protocol in the stack has exactly one lower peer, and thus does not need more than one service access point.

3.1.3 Protocol graphs

IPS provides a combinator `<->` to combine protocol stacks together to express protocol graphs. Suppose that we have defined another protocol stack `stack2` by composing `polite` and `abc`. With the previously given explanations for `stack1`, the specification of this 2-layer `stack2` for `abc` and `polite` should be straightforward: `polite` is connected to `abc` by the service access point `0`. Then, we use our combinator `<->` to merge the two stacks together into the protocol graph shown which is the tree-shaped protocol hierarchy in the subset of Rime.

```
stack2 = stack  (protocol polite [] -1)
                <|>
                (protocol abc [(header abc 0)|* (\x->x==*1)] 0)

rimestack = stack1 <-> stack2
```

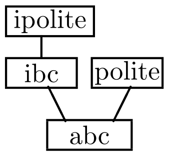

Figure 3.6: A protocol graph implementation in IPS

Care must be taken that all protocols in one protocol stack(graph) use the same protocol buffers for raw data, i.e., *bufferin* and *bufferout*. This is ensured by the combinators for stack construction and stack merging. Whenever two protocols with non-matching buffer names are used, the user receives an informative error message instead of generated code. Instead of this behavior, it would be possible to redefine the combinators. When building a stack, the buffer in all protocols which are going to be overlaid could be redefined to a given name. When combining two stacks, the name used in the first stack could become the common buffer name for the combined stack. However, for the sake of clarity, we have chosen to only check that property rather than to enforce it by modifying the protocol specifications. No automatic renaming behind the scenes should obscure the functionality.

3.2 Discussion of major design decisions

3.2.1 Using Haskell as a host language

We have chosen the functional language Haskell as our host language for embedding IPS because of a number of advantages it has over other languages. Its first and major advantage is that it supports higher-order functions. In consequence, a DSEL in Haskell can be thought of as a higher-order algebraic structure, a first-class value that has the appearance of special syntax. Second, Haskell has lightweight syntax, a property which directly affects the readability of a DSEL implemented in it. If the syntax of the host language is too heavyweight, for instance containing an excessive amount of keywords or parentheses, the embedded DSL will inherit this and possibly require preprocessing or extending the compiler to make code more readable [MHS03]. This, however, somehow eliminates the initial advantages of working with a host language at all. Third, the monad concept in Haskell allows one to cleanly define any desired control structure in a procedural programming style. Fourth, its static type system allows very sophisticated constraints to be placed on the use of DSEL components and their relationships with other parts of the language. Moreover, the type mismatching can be detected at compile time. Haskell has previously been successfully used to design a range of DSELs for different domains: to mention a few prominent representants here, Lava [BCSS98] for hardware design, Paradise [AMS08] for generating excel sheets, LexiFi [JES00] for evaluating financial contracts, Cryptol [Lew07] for code encryption and decryption, and WASH [Thi05] for building web applications.

3.2.2 Using C as a target language

Functional programming languages like Haskell are not normally associated with the development of system software like network protocol stacks, mainly due to the lower performance and lack of support for system programming. IPS programs are not interpreted, instead they are compiled into highly portable non-architecture specific C-implementations, which will be compiled further into machine code for different platforms in a later step. We have chosen to generate C as the target language because C is widely used for system programming tasks as protocol implementations, due to its high performance and tight relationship with operating systems development. Especially important for embedded systems is interoperability with hardware: excellent C compilers are available for almost any hardware and computer architecture. A standards-compliant C program can be compiled for a very wide variety of computer platforms and operating systems with little or no change to its source code.

3.2.3 A modular layered architecture

As we have described previously, protocol specifications are usually written in a modular fashion, specifying a whole stack of layered protocols. This reduces complexity and makes reuse and configuration possible. It should be straightforward for a protocol stack implementation to contain code in the same fashion as the specification, and would render code easy to read, understand and trace back for debugging and maintaining. IPS enforces a strict layered and fully modular structure where any module can be replaced. The basic building-blocks are protocols implemented as individual modules which present a uniform interface allowing nearly arbitrary composition. These separately developed modules are then overlaid to produce the whole protocol stack. However, modularity is one of the chief villains in attempting to obtain good performance due to the large overhead involved in interfacing between modules. Thus, we generate code from modules instead of running them directly. With protocol-oriented knowledge, the compiler could employ cross-layer compiler optimizations.

3.2.4 Using the static type system of the host language

Using static types, the consistency of protocol specifications can be checked at compile time where the compiler can detect logical mistakes in both single protocol implementation and protocol overlay. Furthermore, type classes can be used to model particular protocol types, which constitute a constraint on protocol implementations. Each protocol has a particular protocol type, for example, a retransmission protocol `stuc` has type `RetransmissionProtocol`. And `stuc` has to have a defined function `retransmission`. Then the type system can help generate predictable code parts for this particular protocol type. In the case of retransmission, the compiler additionally generates code for the retransmission service, placing the user-defined `retransmissiontimer` and `retransmission` functions into generic code parts. The type system also helps separate common framework code in a different module. Using type classes, `RetransmissionProtocol` and other protocol variants are implemented as polymorphic instances of a `ProtocolType` class. The code parts common to all protocols are handled by general class functions. `Stuc` and `ibc`, having two different types, can be overlaid as `Protocols` in this way.

3.2.5 Separation of packet processing and protocol logic

Protocols have to process packets. Packet processing functions are actually independent pieces of code which do not have much interaction with the rest of the implementation of protocols. Briefly, an incoming packet in the raw buffer is interpreted according to a packet format and the extracted values for headers will be used by the protocol implementation later; an outgoing packet is structured by aligning header fields which are calculated by protocol logic implementation and are then placed in the hardware buffer. Packet processing can be separated from protocol logic[DOH07, MC00] where all management of packets is dealt with in one place. It avoids the low-level details of packet headers to spread over all the source code. It mimics the way to describe a protocol in a protocol specification where packet formats are specified by tables and protocol logic is specified in running text.

This separation of concerns also solves one of the problems of implementing modular layered protocol stacks: cross-layer information-sharing. That is, making information within one layer available to another layer of the stack. This is almost impossible in the layered architecture, because protocols are developed individually and packet headers that belong to a particular layer are removed after they have been processed. However, in some cases, the removed information from the lower layers is required. For example, the TCP has to access data from its lower layer protocol IP to compute its checksum. Due to the separation of packet processing from protocol logic, once a packet has been parsed, the extracted values for header fields stay as they are. They can be used when needed instead of being removed as the protocol stack processes the packet.

3.2.6 Automatic packet processing from packet descriptions

Packets are semistructured data which can be specified formally by using dependent types, e.g., the physical organization, dependencies among field contents and constraints over the values. Packet specifications contain both syntactic and semantic properties of packet formats, e.g., the detailed physical layout for each field, its allowed values, constraints and dependencies. In general, packet formats and specifications are independent of any given machine's architecture.

We take an approach similar to data description languages, which automatically generate data processing code from data format descriptions. In our case, all packet processing is contained in the automatically generated library from packet descriptions. It enables to build up strong intuitions from high-level perspective which can be straightforwardly mapped from packet figure and explanation in protocol specification. Instead of developing a stand-alone data description language, we embedded it into Haskell. Thus, the description of a packet format is a Haskell term which can be referred seamlessly by protocol logic implementation later on. Since some standard tasks common to all protocol implementations are fully automated, our approach liberates protocol stack implementation from low-level data manipulation related to the wire format of packet, and thus substantially reduces the complexity of such implementations.

Furthermore, the global code generation enables bit-sized header fields packing between layers possible in modular layered protocol stacks. It is a frequent requirement for embedded networks, especially important for sensor networks where protocols are very lightweight and normally only have a few header fields with a small number of bits, e.g., flag fields, type fields, etc. Since each protocol is developed individually, short bit-sized header fields belonging to different layers have to be held in separate byte-sized fields which makes header compression generally impossible. However, reducing header size to a minimum is important for restricting the size of packets within the limit of particular platforms as well as reducing the energy cost for sending and receiving packets. By this approach, the packet description for each protocol can be specified within each module, and the generated packet processing code can be treated in a monolithic way, which makes automatic packet compression possible.

3.3 Preliminary evaluation

The primary metric of a domain specific language is ease of programming. It measures how well a DSL fits the problem domain from the point of view of supporting software development. In the case of protocol stack implementation, a DSL should offer meaningful and intuitive abstractions to support a straightforward mapping between protocol specification and implementation. This measurement is a qualitative metric and is therefore hard to quantify, but we can get some idea about how similar the protocol stack specification and implementation are by looking at the examples shown in Section 3.1. A DSL should also relieve the programmer from making and expressing low-level design decision by hiding the intricacies of low-level details using high-level abstractions. We measure it by lines of code in Section 3.3.1.

We also measure the code footprint of generated code in Section 3.3.2, a property which is especially important for memory-constrained embedded systems. The entire memory footprint at runtime will also depend on the dynamically allocatable buffers. However, the latter is determined by implementation decisions of programmers, and therefore its measurement is out of our range.

Another important metric is the performance of the generated code from source code of a DSL. It is well-known that latency and throughput are the typical measurements of runtime performance for a protocol implementation. At the protocol level, the time to transfer a complete packet involves the time taken for processing a packet, e.g., how long does an outgoing packet take for the first bit to send (latency), plus how long does it take for the remaining bits to send (the length of the packet divided by the throughput). As the throughput mainly depends on the hardware, we only measure the performance of latency in Section 3.3.4. The time taken for processing a packet is composed of two parts: time to process the header and time to process the data. The size of packet headers matters for the energy consumption, as larger packet headers require more transmission and reception time and thus have higher energy consumption. It is important for resource constrained embedded systems where packet headers are commonly compressed before being sent out. Therefore we compare the packet header sizes in different implementations in Section 3.3.3. As the amount of data depends on the particular application, we completely leave it out from our measurement.

We wish to demonstrate that IPS is able to significantly reduce the implementation complexity of protocol stacks, while showing acceptable resource requirements and performance, when compared to the hand-crafted code written by specialists in protocol stack implementation. To keep the example manageable, we only show the single-hop part of the Rime stack.

3.3.1 Lines of code

One of the design goals of IPS is to facilitate the implementation of protocol stacks. To substantiate that IPS reduces the complexity of protocol stack implementation, we compare the Rime implementation written in IPS with the corresponding original implementation written in C. We use the lines of code used to implement the protocol stacks as an approximation of the implementation complexity. Of course, this measure is affected by aspects not inherent to the programming language, but rather due to programming style and conciseness. Results have to be interpreted with care and minor differences would not indicate differences in the complexity which we intend to measure here. However, our results are clear enough to show a general trend and obvious differences.

Table 3.1 lists the lines of code in the Rime implementation in IPS and in C. For Rime implemented in C, we count statements as lines of code, thus excluding comments and header files. Accordingly, the numbers for the implementation in IPS count IPS statements, excluding comments and module specification statements. We see that the numbers for IPS are considerably smaller than the ones for Rime's original implementation, generally less than 25 % of the lines of C code. In addition, we see that the code for `ipolite` is extremely short, because `ipolite` does not add substantially new function-

Table 3.1: Lines of code for Rime implementations

Lines of code	C implementation	IPS implementation	Ratio
abc	65	16	0.25
ibc	88	18	0.20
uc	91	22	0.24
stuc+ruc	97 + 72	39	0.23
polite	89	20	0.22
ipolite	89	3	0.03

ality. In fact, `ipolite` is identical to the previously-defined protocol `polite`, only situated in other place in the protocol graph. While the C implementation duplicates a considerable amount of code, IPS can reuse the previous code and only overwrite the protocol name as shown previously.

3.3.2 Code footprint

Table 3.2 lists the code memory footprint of the Rime implementations in hand-crafted C code and IPS, both compiled for the COOJA simulator [OAJ+06]. We see that the code footprint resulting from our implementation is bigger than the one from the hand-crafted implementation. The reason for this is that the hand-crafted implementation uses packet attributes instead of packet headers, and a separated transformation module to transform packet attributes into packets with headers and vice-versa. Thus, all the packet processing code is separated from the proper Rime stack implementation. In our implementation, the packet processing C code resulting from our code generation is integrated with the protocol logic implementation, which increases the code footprint for each protocol.

Table 3.2: Static memory footprint of Rime implementations

Code footprint (Bytes)	C implementation	IPS implementation
abc	4408	6852
ibc	5192	7408
uc	5492	7148
stuc+ruc	9160+8380	19322
polite	7312	8896
ipolite	7928	8896

3.3.3 Size of packet headers

In IPS-generated code, all fields from different layers are presented in forms of field handles specifying physical locations. The generated packet processing library is bit-oriented rather than byte-oriented, i.e., parsing and marshaling functions can extract and construct packet fields crossing byte boundaries, and quantities can be aligned arbitrarily. This mechanism allows IPS to compress small header fields into the same byte, even when they belong to different protocols. This bit-oriented implementation saves considerable memory by efficiently using all available bits in a transmitted byte, and considerably reduces the size of the transmitted header, as we will exemplify by the protocols abc, ibc, uc and ruc.

`abc`	2-bit	Packet type flag
`ibc`	2-bit	Packet type flag
	16-bit	Sender address
`uc`	2-bit	Packet type flag
	16-bit	Receiver address
`ruc`	2-bit	Packet type flag
	5-bit	Packet ID
	5-bit	Retransmission counter
	⋮	⋮

Figure 3.7: Header fields of selected Rime protocols

Lowest protocol in the stack (and thus at the beginning) is `abc`, which encodes its packet type (the routing to the upper layers) in one header fields requiring `2` bits. The `ibc` above equally requires one `2-bit` field for the packet type, and adds an address field, the length of which we specified as `16` bits. `Ibc`'s layered protocol `uc` again has two header fields which require `2` and `16` bits individually. More header fields follow in the upper layers of `ruc/stuc`: another packet type flag (`2` bits), a packet ID (`5` bits), a retransmission counter (`5` bits), and more fields which we do not discuss further here. With byte alignment for each protocol stack layer, these headers would have to be located as shown in Figure 3.8. Even more empty "padding" space is required when the target hardware uses word-alignment and a word size of 16, 32 or 64 bits.

Figure 3.9 shows how IPS can align the header fields for different protocols without any padding requirements. The header fields we have shown for an `abc/ibc/uc/ruc` packet take 50 bits and are efficiently compressed into 7 bytes, whereas the byte-aligned version requires 9 bytes.

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T  |padding    |T  | Sender    | ABC/IBC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   address         |padding    | IBC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T  |    Receiver address       | UC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   |padding    |T  |Packet ID| | UC/RUC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Retrans|  ...                  | RUC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.8: Non-packed layered header of the Rime stack (fields byte-aligned).

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T  |T  |  Sender address       | ABC/IBC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       |T  |  Receiver address | UC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           |T  |Packet ID| Retr| RUC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|ran|                           | RUC
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.9: Bit-oriented Rime packet header generated by IPS (no alignment required).

3.3.4 Performance

We measure and compare the runtime performance of the Rime implementation generated by IPS and the original Rime implementation by experiments. The application scenario is a sensor network with 25 sensor nodes in a square lattice for measuring temperature. One node acts as a sink node which collects all temperature information(4 bytes of application data) measured in the network. Each node runs a Rime protocol stack, exchanging packets with its single-hop neighbor by using the single-hop unicast protocol stack of Rime.

We simulate our test scenario by COOJA [OAJ$^+$06]. To make sure all the experiments work on the same input, we use 10000 already-collected temperature values. We measure the amount of time that is needed to process a packet for sending/receiving, by taking timestamps before calling and after returning from the Rime stack. We carried out all experiments on the same computer. Each experiment was repeated 3 times and the lowest value was taken.

Figure 3.10 and Figure 3.11 report the estimated average time for sending/receiving a packet, and how they develop over time in the test setup for sending

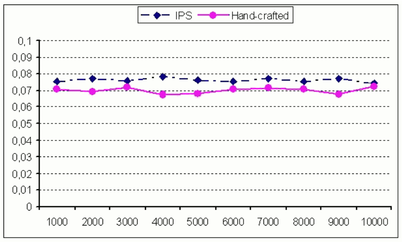

Figure 3.10: Average time for sending uc packets

receiving

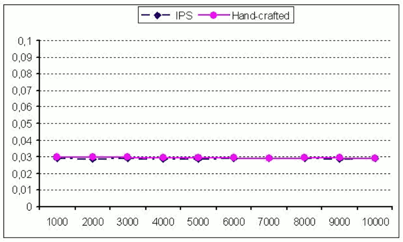

Figure 3.11: Average time for receiving uc packets

both the IPS and hand-crafted implementation. We see that the execution time resulting from the generated and the hand-crafted code are approximately the same. This is not surprising, since the C code resulting from our code generation uses techniques similar to the hand-crafted implementation, i.e., bit-oriented packing to reduce the size of header. While our measurements are slightly slower, the reduction is not significant.

3.4 Scope and limitations

We have decided to delimit the scope of the computations by addressing asynchronous message-passing protocols on single processor distributed real-time embedded systems – synchronous communication is not addressed in this research.

So far, the IPS does not comprise the runtime system. Some of the abstractions, i.e., timer, event and thread, heavily rely on particular features of the Contiki runtime system which is our primary testing and development platform. We plan to generalize from Contiki [Dun07] and develop an appropriate runtime system for IPS in the future.

Chapter 4

Conclusions and Future Work

4.1 Summary

This thesis investigates a language-based approach to network programming in embedded systems. We design and implement a domain specific language to facilitate protocol stack implementation for embedded systems.

We confine our attention to protocol stack specifications and automatic code generation. Our prototype language IPS captures the core features for the implementation of protocol stacks. It provides high-level notations and abstractions and hides the intricacies of low-level details, e.g. packets, buffers, timers and so on. IPS thereby provides a unified framework for specifying protocols, composing protocols to form protocol stacks and building protocol graphs. The high-level specification allows programmers to concentrate on the functionality of protocols rather than on the details of how to structure and optimize code for packet processing and protocol logic. It enables flexible composition of modules, making it possible to construct a new protocol stack by reusing existing modules instead of rewriting code.

So far, our primary working area is to automatically derive code for packet processing from packet format descriptions. We have developed a stand-alone tool which generates C libraries for packet processing from packet format descriptions (presented in the appended Paper II). Subsequently, we have integrated this packet format description language seamlessly as part of the language IPS, where a packet format description can be dually compiled into a packet processing library and an internal data representation which can readily be used for protocol logic implementation. IPS uses the same packet description mechanism and code generation, but we have streamlined the implementation by using a technique of "embedded compilation" [EFdM03]. Embedded compilation exhibits two essential advantages over stand-alone implementations with parser and code generator: ease of implementation and increased type safety. We have embedded a data description language in Haskell for processing ad hoc data formats (presented in the appended Paper I), which is a generalization of our language for describing packets.

The internal compilation of IPS into C code allows the programmers to specify protocol stacks in a high-level language while ensuring good performance and high portability. As the preliminary experimental evaluation shows, IPS is able to achieve the high efficiency of compiled code in terms of energy consumption and memory usage.

4.2 Future work

We believe that this work has taken some important steps towards a DSL to ease the development and the deployment of protocol stacks for embedded systems.

We would like to explore more about how a suitable runtime system can be used to address resource usage. In the case that the stack implementation is located as part of an existing operating system (OS), we would like to employ the existing OS primitives like timers and threads. For example, so far IPS heavily relies on particular runtime primitives of Contiki [Dun07], i.e., protothreads and timers. Decoupling IPS from Contiki can be achieved by giving different options at compile time to employ different runtime primitives. For cases without a (suitable) OS on the target platform, we plan to develop our own specialized runtime system as a loadable kernel module. It should provide a suitable concurrency model to easily handle multiple threads, an efficient and flexible memory management, and other operating system facilities needed by protocol implementation, like timers. The concurrency model could be similar to TinyTimber [LEAN08], i.e., lightweight and architecture-independent, which can widely adapt to variations in network conditions. For memory management, explicit dynamic memory management could be a good option which has less runtime cost compared to garbage collection in terms of processor overhead and additional memory requirement. This approach is flexible and efficient but unsafe. For instance, the programmer is allowed to return memory to the system using a *free* call, which leads to memory leaks if the programmer forgets to free it, or to dangling pointers if memory is freed too soon. IPS can avoid this kind of situation by using constructs on top of these unsafe levels, and expand to them in a way that we believe is particularly suitable for a protocol stack runtime system. At the source code level, IPS does not give the programmer right to explicitly request and release memory. In the corresponding generated code, according to the state of predictable context (e.g., a communication session is ended or not), the memory (e.g., the packet pool for one session) can be maintained explicitly by automatically generated IPS runtime system code.

We would like to investigate different protocol oriented compilation techniques to address the protocol stack as a whole, in order to reduce the performance penalty for layering, and to produce more reliable and efficient code. We are most interested in cross-layer compiler optimizations. For example, [HR97] shows three techniques which can be automated in a compiler with the help of annotations of the language: optimizing some computations, compressing protocol headers and delaying processing. So far, we used bit-oriented packet processing to handle the inefficiency caused by the abstraction barriers between layers. We would like to explore the other two techniques as well. As we have embedded our language in Haskell, rewriting rules in the Glasgow Haskell Compiler (GHC) [JTH01] are another direction that we are going to study. Rewriting, enriching GHC compilation with domain specific knowledge, offers a powerful way to optimize the program.

We intend to implement a series of practical protocols in the very near future, to explore the expressiveness of IPS, and to guide further improvements and extensions towards a more full-fledged language. For an individual protocol, its state machine and valid state transitions deserve to be studied, which are essential to guarantee the correctness of the implementation. So far in IPS, the overlay for a protocol stack focuses on packet handling which can be compiled into a dataflow engine. We would like to explore more features of overlays [LCH$^+$05] to express protocol stacks concisely and logically, i.e., the interrelated possible events and states.

Finally, we will formally present syntax and semantics of essential IPS abstractions.

References

[AMS08] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage dsl embedded in haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 225–228, New York, NY, USA, 2008. ACM.

[AP93] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.

[Bac02] Godmar Back. Datascript - a specification and scripting language for binary data. In *GPCE*, pages 66–77, 2002.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ACM International Conference on Functional Programming*, 1998.

[BMvE98] Anindya Basu, J. Gregory Morrisett, and Thorsten von Eicken. Promela++: A language for constructing correct and efficient protocols. In *INFOCOM*, pages 455–462, 1998.

[CHR$^+$03] C. Consel, H. Hamdi, L. Reveillere, L. Singaravelu, H. Yu, and C.Pu. Spidle: A DSL approach to specifying streaming applications. In *Second International Conference on Generative Programming and Component Engineering*, 2003.

[CRL96] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *PLDI '96*, pages 237–248, New York, NY, USA, 1996. ACM Press.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90*, pages 200–208, New York, NY, USA, 1990. ACM Press.

[DOH07] Adam Dunkels, Fredrik Österlind, and Zhitao He. An adaptive communication architecture for wireless sensor networks. In *SenSys*

'07: Proceedings of the 5th international conference on Embedded networked sensor systems, pages 335–349, New York, NY, USA, 2007. ACM.

[Dun07] Adam Dunkels. *A Language-based Approach to Protocol Programming Memory-Constrained Networked Embedded Systems.* PhD thesis, Swedish Institute of Computer Science, 2007.

[EFdM03] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[FG05] Kathleen Fisher and Robert Gruber. Pads: a domain-specific language for processing ad hoc data. *SIGPLAN Not.*, 40(6):295–304, 2005.

[FMW06] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *SIGPLAN Not.*, 41(1):2–15, 2006.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

[HR97] Mark Hayden and Robbert Van Renesse. Optimizing layered communication protocols. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 169, Washington, DC, USA, 1997. IEEE Computer Society.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[Hud97] Paul Hudak. Modular domain specific languages and tools. Technical report, Department of Computer Science, Yale University, 1997.

[JES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 280–292, New York, NY, USA, 2000. ACM.

[JTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.

[KKM99] E. Kohler, M.F. Kaashoek, and D.R. Montgomery. A readable tcp in the Prolac protocol language. In *ACM SIGCOMM '99 Conference*, 1999.

[LCH$^+$05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[LEAN08] Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Nordlander. Tinytimber, reactive objects in c for real-time embedded systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 1382–1385, New York, NY, USA, 2008. ACM.

[Lee00] Edward A. Lee. What's ahead for embedded software? *Computer*, pages 18–26, 2000.

[Lew07] Jeff Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *FMSE '07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 41–41, New York, NY, USA, 2007. ACM.

[MC00] Peter J. McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, New York, NY, USA, 2000. ACM.

[Mcg04] Tommy Marcus Mcguire. *Correct Implementation of Network Protocols.* PhD thesis, The University of Texas at Austin, 2004. Supervisor-Mohamed G. Gouda.

[MHS03] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. Technical report, Stichting Centrum voor Wiskunde en Informatica, 2003.

[OAJ$^+$06] Fredrik Österlind, Dunkels Adam, Eriksson Joakim, Finne Niclas, and Voigt Thiemo. Cross-level sensor network simulation with cooja. In *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.

[RFC] Rfc793. http://www.faqs.org/rfcs/rfc793.html. [Online; accessed 18-March-2009].

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley Professional Computing Series, 1994.

[Thi05] Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Interet Technol.*, 5(1):1–46, 2005.

Paper I

A Library for Processing Ad hoc Data in Haskell

Embedding a Data Description Language

Yan Wang and Verónica Gaspes

Proc. of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)
Hatfield, United Kingdom, September 2008

A Library for Processing Ad hoc Data in Haskell

Embedding a Data Description Language

Yan Wang and Verónica Gaspes

Halmstad University
{yan.wang,veronica.gaspes}@hh.se

Abstract. Ad hoc data formats, i.e. semistructured non-standard data formats, are pervasive in many domains that need software tools — bioinformatics, demographic surveys, geophysics and network software are just a few. Building tools becomes easier if parsing and other standard input-output processing can be automated. Modern approaches for dealing with ad hoc data formats consist of domain specific languages based on type systems. Compilers for these languages generate data structures and parsing functions in a target programming language in which tools and applications are then written. We present a monadic library in Haskell that implements a data description language. Using our library, Haskell programmers have access to data description primitives that can be used for parsing and that can be integrated with other libraries and application programs without the need of yet another compiler.

1 Introduction

Imagine your favorite online travel agency dealing with data from many sources: airlines, train companies, car rental companies, hotels and maybe some more. It has to understand all these formats for which parsers and converters have to be programmed. Moreover, as the agency comes to serve more companies this work has to be done again for new data formats. It is most likely that these companies have legacy data formats that have not been upgraded to XML or other standardized formats for which tools exist. Furthermore, these data formats frequently evolve over time, leave some fields unused or use them for new purposes etc, making some of the data seem erroneous according to the latest version of the format. This scenario is not at all unique. In bioinformatics, demographic applications, geophysics applications, network traffic monitoring, web servers logs, etc, most of the data formats are *ad hoc*, i.e., semistructured non-standard data formats. Typical tools programmed for ad hoc data sources of this kind include generating reports, exporting data to other formats, collecting statistics and reporting errors in input data.

We came across a similar problem when designing a domain specific language for the implementation of protocol stacks. Packet formats are often described in packet specifications. The physical organization, the dependencies among field contents and the constraints over the values of some fields are provided using a combination of figures and explanations. In protocol implementations there are no traces of this. Instead, references to fragments of a buffer are interspersed all over the code implementing a protocol. These fragments are converted to and from network format; bit operations are used to determine the value of the fields that need to be checked, etc.

Parser generators are very good at describing data formats specified by context free grammars. However, for ad hoc data they have been found lacking. In most ad hoc formats some of the fields depend on the values of other fields. Using parser generators these dependencies are usually dealt with in the actions specified together with the grammar rules and are thus not part of the format specification. Also, parser generators support error handling, but this becomes part of the parser. In the case of ad hoc formats it is desirable to leave error handling to other tools as it is often the case that erroneous data has to be processed anyway. Finally, ad hoc formats are needed also for binary data.

Modern approaches for describing ad hoc data formats are data description languages[12,3,5]. These are domain specific languages with constructs based on dependent type systems. Compilers for these languages generate libraries in some target programming language. These libraries include types, parsing functions and tools like pretty printing that can be used to build more advanced tools. This is very good: one description is used to generate several utilities automatically. On the other hand, extensions with new constructs and the creation of more tools require in most of the cases modifications to the compiler – the exception being PADS/ML for which a generic mechanism is provided [4].

This paper describes a library implementing a data description language embedded in Haskell that facilitates dealing with ad hoc data formats. With our approach, both the data description and the rest of the tools are Haskell programs. The library is based on the Data Description Calculus (DDC) [6], which we briefly present in Section 2. In Section 3 we present our DDC library, constructed in the style of monadic parser combinators [9]. In Section 4 we show how to integrate a data description with other tools by using pretty printing as an example. In Section 5 we explore performance, and in Section 6 we discuss related work, before the final conclusions. The contributions of this work are as follows:

- A rich collection of parser combinators for ad hoc data formats. Both physical layout, field dependencies and semantic constraints can be expressed.
- Built-in error detection and collection that does not halt the parsing process. This can be used for generating precise error messages and statistics, including positions and causes, both syntactical and semantic.
- Ways of extending the library with new primitive parsers, using type classes.
- Suggestions for extending the library with additional tools.

2 The Data Description Calculus

Recently, the essentials of data description languages have been formalized in the Data Description Calculus (DDC) [6]. It uses types from a polymorphic, dependent type theory to describe various forms of ad hoc data. What we report in this paper is an implementation of this calculus as an embedded language in Haskell. In order to motivate our choices later on, we briefly present the calculus. We explain only a few constructs by giving small examples and explaining the semantic functions for these cases.

In DDC, data formats are described using types according to the syntax shown in Figure 1. Base types $C(e)$ are variations of types of the target language[1], for example strings or integers, parameterized by an expression e in the target language. The expression is used to specify simple properties of a data field, such as its length in number of characters or the number of bits in the representation. This expression can be a variable that can be used to refer to the value of a previous field in the data format. For imposing more semantic constraints on a data field there is the constraint type $\{x : \tau \mid e\}$ where the expression e from the target language can refer to x, the value of the data field. For putting together fields there is the dependent sum type $\Sigma x : \tau_1.\tau_2$. The variable x can be used in τ_2 to refer to the value of the first field. In this way the value of a field can influence some aspect of another one, for example its length: $\Sigma x{:}Int.String(x)$.

$$\tau = C(e) \mid \{x : \tau \mid e\} \mid \Sigma x{:}\tau.\tau \mid \tau + \tau \mid \tau \& \tau \mid \texttt{compute}(e) \mid \texttt{absorb}(\tau) \mid \ldots$$

Fig. 1. The syntax of DDC

For each type τ the semantics prescribes three interpretations:

1. $[\![\tau]\!]_{\mathrm{P}}$ maps τ into a parsing function in the target language. The parsing functions return pairs $([\![\tau]\!]_{\mathrm{REP}},[\![\tau]\!]_{\mathrm{PD}})$.
2. $[\![\tau]\!]_{\mathrm{REP}}$ maps τ into a type in the target language for the in-memory representation of the data.
3. $[\![\tau]\!]_{\mathrm{PD}}$ maps τ into a type in the target language for a parse descriptor where error information can be collected during parsing. This meta data is interesting when parsing ad hoc data because usually sources have to be dealt with in spite of being erroneous and in some cases applications are even meant for the collection and analysis of errors.

Some short illustrations should help clarify how to use the constructs of DDC and the role of the three interpretations.

– $C(e)$, with a type C and an expression e of the target language, can be used to describe physical layouts. For example, strings of a given length $String(20)$; or ending in a particular character $String(';')$. Likewise, it can describe integers that use a number of digits $Int(3)$; or are represented by a number of bits $BitInt(8)$. For these types, $[\![\ ]\!]_{\mathrm{REP}}$ is an atomic type in the target language, like `String` or `Integer`. The function $[\![\ ]\!]_{\mathrm{P}}$ takes care of the constraint provided by the expression during parsing.
– $\Sigma x : \tau_1.\tau_2$ is the construct for dependent pairs. It can be used to constrain fields with the values of previous fields, as in $\Sigma x{:}BitInt(8).String(x)$. The parsing function provided by $[\![\ ]\!]_{\mathrm{P}}$ deals with the two fields in sequence using

[1] The target language is the language for which tools are generated, it is a parameter for the calculus.

$[\![\tau_1]\!]_{\mathrm{P}}$ first, and then using the result of type $[\![\tau_1]\!]_{\mathrm{REP}}$ when the remaining input is parsed according to $[\![\tau_2]\!]_{\mathrm{P}}$. $[\![\ ]\!]_{\mathrm{REP}}$ is the pair $([\![\tau_1]\!]_{\mathrm{REP}}, [\![\tau_2]\!]_{\mathrm{REP}})$.
- $\{x\!:\!\tau|e\}$ expresses constraints on a field, for example $\{x\!:\!Int(3)|x > 38\}$. The constraint itself is given as a boolean expression e in the target language that depends on values x of type $[\![\tau]\!]_{\mathrm{REP}}$. The parsing function parses according to $[\![\tau]\!]_{\mathrm{P}}$ and then checks whether the calculated value of type $[\![\tau]\!]_{\mathrm{REP}}$ satisfies e. The parse descriptor $[\![\tau]\!]_{\mathrm{PD}}$ indicates whether the value satisfies the condition or not.
- `compute`$(e : \sigma)$ can be used to output a value in the parsing function (the value of e) without a corresponding field in the input.
- `absorb`(τ) deals with data that is important for parsing, for example a separator, but becomes uninteresting as output. The parser parses according to $[\![\tau]\!]_{\mathrm{P}}$ and discards the output.

As Fisher et al. argue in [6], there are many parallels between DDC and parser combinators [9]. For example there is a clear relation between the parsing function for Σ types and the monadic sequence combinator. We exploit this in our library using rich monadic combinators for parsing and generating error information. However, DDC constructs focus on ad hoc formats and describe data formats using types rather than grammars, so the combinators are rather different. Another difference is the inclusion of parser descriptors in the result of parsing instead of the generation of errors.

3 A Haskell Embedding

We present an embedding of the DDC in Haskell as a library of ad hoc data parser combinators. Our combinators follow closely the structure of the type constructors of DDC. Our embedding works by implementing the interpretation of DDC types as parsing functions (see Section 2 for an informal explanation). Both the representation type and the parse descriptor associated to each DDC type are implemented in the return type of our combinators. By embedding the constructors of DDC in this manner, our library is easily extensible: ad hoc data descriptions are ordinary Haskell terms, and we have full access to existing Haskell libraries.

We identify each type τ in the calculus with the parsing function $[\![\tau]\!]_{\mathrm{P}}$ introduced in Section 2. Parsing functions compute values of the representation type $[\![\tau]\!]_{\mathrm{REP}}$ and in doing so also update meta data corresponding to the parse descriptor $[\![\tau]\!]_{\mathrm{PD}}$. Therefore, we have introduced a data type representing a parser for type τ:

```
AdhocParser t a
```

In this type, `t` is the type of tokens in the input source and `a` is the type of the result. The type of parse descriptors is not a parameter and is described in more detail in Section 3.4.

Briefly, our library consists of a collection of primitive parsers and parser combinators with a monadic interface that correspond to the combinators for constructing types in DDC. We first give a flavor of our library using an example.

3.1 An Introductory Example

The following is a description of a network address format using our library:

```
webserver =
  orp ipaddress dnsname
ipaddress =
  countp (charequal ".")
         bottom
         (int_range 0 255)
         4
dnsname =
  seqp (charequal ".")
       (charequal " ")
       (stringendsat (\x -> x=="."||x==" "))
       false
```

A network address is either an IP address, or a DNS name. An IP address is a sequence of 4 integers separated by `"."`. Each integer should lie between 0 and 255. Similarly, a DNS name is a sequence of strings, separated by `"."` and ended by `" "`. The sample records could for instance be *"www.hh.se ... ..."* or *"194.47.12.29 ... ..."*.

The two alternatives are put together using `orp`. `Ipaddress` is a sequence with a fixed length `4` which is specified using `countp`. The elements of the sequence are separated by `"."` (`charequal "."`). There is no terminator (`bottom`) to signal the end of the sequence. Each element is an integer within an interval (`int_range 0 255`). `Dnsname` is a sequence with no predefined length, thus it is specified using a more general combinator `seqp` with the separator `"."`, the terminator `" "`, the description of the elements: strings ended by either `"."` or `" "`, and a trivial boolean-valued function `false`, i.e., `\_ -> False`, to specify no extra termination condition.

3.2 Primitive Parsers

For the primitive DDC types *unit* and *bottom* our library includes two primitive parsers:

```
unit :: AdhocParser t ()
```

that consumes no input and returns nothing. It also produces a parse descriptor indicating no error. The parser for *bottom*

```
bottom :: AdhocParser t ()
```

neither consumes input nor returns a result, but the parse descriptor indicates the error count and the error code.

The implementation of primitive parsers for the base types $C(e)$ depends on both C and e. In DDC, C can be any type from the target language (also Haskell in our case). In order to provide for ways of adding more base types, we have introduced a type class `Basetype`.

```
class Basetype t a where
  readTokens :: [t] -> Maybe a
  fromTokens :: [t] -> (Maybe a, Offset)
```

A pair of types `t` and `a` for the input tokens and the result value respectively, is an instance of `Basetype` if we can transform a chunk of tokens into a value and consume valid tokens as far as possible for building a value.

There are three kinds of expressions (e) which are the most commonly used: () for no extra information, ($e :: Int$) for values described using e tokens and ($e :: t$) for values ending with token e. In our library there is a parser for each of these.

For the base type $C()$, we have defined `base`: it parses an input stream by eagerly reading valid tokens. It requires `t` and `a` to be an instance of `Basetype`.

```
base :: (Basetype t a) => AdhocParser t a
```

For the base type $C(n :: Int)$, we have defined `baselen` that parses data consuming exactly `n` tokens.

```
baselen :: (Basetype t a) => Int -> AdhocParser t a
```

For the base type $C(e :: t)$, we have defined `baseend` that consumes a sequence of tokens until the terminating token `e`.

```
baseend :: (Basetype t a, Eq t) => t -> AdhocParser t a
```

We can easily use `readTokens` and `fromTokens` to built other variants with different kinds of e for $C(e)$. For example, the parser

```
baseendsat :: (Basetype t a) => (t -> Bool) -> AdhocParser t a
```

consumes a sequence of tokens until the terminating token satisfies the supposed condition.

With the primitive parsers for base types, it is straightforward to define parsers for Haskell types. After instantiating `BaseType` with the type of tokens `t` and the desired type of values `a`, only the signatures for parsers have to be provided. We illustrate this by implementing parsers for $Int()$, $Int(n)$ and $Int(e)$ in the case of a character input stream. We have to specify how a sequence of `Char` can be translated into a value of type `Int` by implementing `readTokens`, and what is a legal `Char` stream for type `Int` by implementing `fromTokens`:

```
instance Basetype Char Int where
  readTokens [] = Nothing
  readTokens ts = if (all isDigit ts) then Just (read ts)
                                      else Nothing
  fromTokens ts = (readTokens ts', len)
    where ts' = takeWhile isDigit ts
          len = length ts'
```

Finally, we just give a type signature for each parser:

```
int :: AdhocParser Char Int
int = base

intlen :: Int -> AdhocParser Char Int
intlen = baselen

intend :: Char -> AdhocParser Char Int
intend = baseend
```

3.3 Parser Combinators

Parser combinators follow DDC type constructors. However, thanks to higher-order functions and recursive types in Haskell, we need not define combinators for the abstraction type $\lambda x.\tau$, the application type τe, and the fix point type $\mu\alpha.\tau$. We present the implementation of the other types of DDC.

– The dependent sum combinator `sigmap` corresponds to the dependent sum type $\Sigma x : \tau_1.\tau_2$ in DDC.

  ```
  sigmap :: AdhocParser t a ->          -- $[\![\tau_1]\!]_P$
            (a -> AdhocParser t b) ->   -- $[\![\tau_2(x)]\!]_P$
            AdhocParser t (a,b)         -- $[\![\Sigma\ x : \tau_1.\tau_2]\!]_P$
  ```

  `sigmap p q` combines the parsers `p` and `q` sequentially in which `q` may refer to the parsing result of `p`. For example, in DDC we would use $\Sigma x : \mathit{Char}().\mathit{String}(x)$ to describe a sequence of characters started and terminated by a same character. Using our library combinator *sigmap*, we write

  ```
  xstringx = sigmap char stringend
  ```

– The choice combinator `orp` corresponds to the sum type $\tau_1{+}\tau_2$ in DDC, which is used to describe a data source with values of type either τ_1 or alternatively τ_2.

  ```
  orp :: AdhocParser t a ->          -- $[\![\tau_1]\!]_P$
         AdhocParser t b ->          -- $[\![\tau_2]\!]_P$
         AdhocParser t (Either a b)  -- $[\![\tau_1 + \tau_2]\!]_P$
  ```

  `orp p q` first tries `p`. If `p` succeeds, it returns the value of type `Left a`. If `p` fails, it goes back to the starting point of the input stream to apply `q` and returns the value of type `Right b`. For example, in DDC we would use *String(”September”)* + *String(”Sep”)* and with our library we write

  ```
  sep = orp (stringequals "September")
            (stringequals "Sep")
  ```

– The intersection combinator `andp` implements the intersection type $\tau_1 * \tau_2$, which is used to describe data source whose value matches both τ_1 and τ_2.

```
andp :: AdhocParser t a ->   -- $[\![\tau_1]\!]_{\mathrm{P}}$
        AdhocParser t b ->   -- $[\![\tau_2]\!]_{\mathrm{P}}$
        AdhocParser t (a,b)  -- $[\![\tau_1 * \tau_2]\!]_{\mathrm{P}}$
```

`andp p q` is parameterized by two underlying parsers `p` and `q`. It advances the input stream by employing both `p` and `q` from the same starting point and ending at the maximum offsets of two parsers. The final parsed result is a pair built by the two results returned by `p` and `q`. If and only if both parsers succeed on the input stream, the whole parser succeeds. For example,

```
num = andp (intlen 4) (floatlen 6)
```

is a parser that accepts strings starting with four characters to build an integer and six characters to build a float. Assume that the input stream is *”1234.0...”*, it results in *(1234, 1234.0)*. Since the maximum offset of two underlying parsers is 6, the remaining input stream is ”...”.

– The constrain combinator `constrainp` implements the constrained type $\{x\!:\!\tau|e\}$, which is used to impose conditions on data fields. This combinator has type

```
constrainp :: (a -> Bool) ->       -- $e$
              AdhocParser t a ->  -- $[\![\tau]\!]_{\mathrm{P}}$
              AdhocParser t a     -- $[\![\{x\!:\!\tau|e\}]\!]_{\mathrm{P}}$
```

`constrainp f p` is parameterized by a single underlying parser `p` and a boolean-valued function `f`. The parsing result of `p` is checked according to `f`. If the imposed constraint is not satisfied, the semantic error is recorded in the parse descriptor. For example,

```
int_range a b = constrainp (\x -> x>=a && x<=b) int
```

is a parser that accepts strings starting with an integer value produced by the parser `int`. The value should lie between `a` and `b`. If it is outside the range, the result is anyway the integer read and an error code `Err` indicating semantical error is recorded in the parse descriptor.

– The sequence combinator `seqp` implements the array type $\tau\, seq(\tau_s, e, \tau_t)$, which is the type used to describe data source formed as a sequence of fields of type τ.

```
seqp :: AdhocParser t sep ->     -- $[\![\tau_s]\!]_{\mathrm{P}}$
        AdhocParser t term  ->  -- $[\![\tau_t]\!]_{\mathrm{P}}$
        AdhocParser t a  ->      -- $[\![\tau]\!]_{\mathrm{P}}$
        ([a] -> Bool)  ->        -- $e$
        AdhocParser t [a]        -- $[\![\tau\ \texttt{seq}\ (\tau_s,\ e,\ \tau_t)]\!]_{\mathrm{P}}$
```

`seqp ts tt te f` has four parameters: `ts` is the parser for the separator found between elements, `tt` is the parser for the terminator of the sequence, `te` is the parser for each element in the sequence and `f` is a boolean-valued function which examines the parsed sequence to determine whether the sequence has completed before reaching the terminator. In the example,

```
arr = seqp charends ","
           charends "."
           int
           (\ls -> length ls == 10)
```

`","` is the array separator, `"."` is the array terminator, the length of the result array should not be greater than 10 and the elements of array are parsed with `int`. If the given input is a string *"1,2,3,4,5,6,7,8,9,10,11,12"*, resulting value is a list `[1,2,3,4,5,6,7,8,9,10] :: [Int]`. If the input is *"1,2,3,4,5.6,7,8,9,10,11,12"*, the result is a list `[1,2,3,4,5]` instead.
The last argument of the parser is examined after each element is read to see whether the parsing should end: execution becomes very expensive. Because testing for length is so common, we provide a parser `countp` that calculates the length incrementally

```
countp :: AdhocParser t sep ->
          AdhocParser t term  ->
          AdhocParser t a  ->
          Int  ->
          AdhocParser t [a]
```

– The compute combinator `computep` implements the compute type $compute(e)$, which allows us to include a value in the output that does not appear in the data source.

```
computep :: a ->              -- e
            AdhocParser t a  -- [[compute(e)]]_P
```

`computep e` does not do any parsing, but returns the computation result of `e` without parsing errors. In the example,

```
c = computep (2+3)
```

no matter what the input stream is, it results in `5`.
– The absorb combinator `absorbp` implements the type $absorb(\tau)$, which is used to parse and discard the result

```
absorbp :: AdhocParser t a ->  -- [[τ]]_P
           AdhocParser t ()    -- [[absorb(τ)]]_P
```

`absorbp p` simply applies the underlying parser `p`, but ignores the parsed result. In the example

```
xstringx' = absorbp xstringx
```

if `xstringx` parses successfully, the parser `xstringx'` will result in `()`. If `xstringx` fails, `xstringx'` fails as well.
– The scan combinator `scanp` implements the type $scan(\tau)$, which is used to scan the input stream for data that makes the underlying parser succeed.

```
scanp :: Offset ->
         AdhocParser t a ->        -- [[τ]]P
         AdhocParser t (Maybe a)   -- [[scan(τ)]]P
```

`scanp maxoffset p` is parameterized by a predefined maximum scan-offset `maxoffset` and the underlying parser `p`. It attempts to apply `p`, if it fails, it moves on one more token and tries it again until it reaches the scan-offset `maxoffset`. For example,

```
scanxstring = scanp 100 xstringx
```

tries at most 100 times to find a string started and terminated by the same character.

3.4 Implementation

The type `AdhocParser` has been implemented using [9] as a guideline.

```
newtype AdhocParser t a
  = P (([t], PD) -> (Either String a, [t],PD)))
```

Parsers are functions that transform a stream of input tokens of type `[t]` paired with the current parser descriptor of type `PD` into a tuple containing the resulting data representation of type `a`, the remaining unconsumed tokens of type `[t]` and the updated parse descriptor of type `PD`.

`AdhocParser` is actually a combination of the usual `Parser Monad` with the `Error` and `State Monads`. The monadic sequencer `>>=` can be considered as a combinator and the do-notation achieves a programming style mimicking the records or structures provided in some conventional languages. We make `AdhocParser t a` an instance of the `MonadPlus` class for backtracking. We encapsulate the result into the `Either Monad` to handle parsing failure gracefully. We use `Right` to indicate success or success with error and `Left` to indicate failure. We deal with parse descriptors as states and thus make `AdhocParser t a` an instance of the `MonadState` class.

For parse descriptors we use the type `PD`

```
newtype PD = MkPD Int ErrCode Span Body
```

that corresponds closely to how parse descriptors are defined in the DDC:

$$pd = pd\_hdr * pd\_body$$
$$pd\_hdr = int * errcode * span$$

It stores the detected error information for corrupted input streams including an error count of type `Int`, an error code of type `ErrCode`, a span of type `Span`, and a parse descriptor body of type `Body`. The error count for a parse accumulates the error counts from its subcomponents and itself. The error code specifies the degree of success of a parser, i.e., success `(Ok)`, success with errors `(Err)` or failure `(Fail)`. The span is a pair of offsets which indicate the start and end points in the input stream for the current parser. The body for a parse descriptor puts together the descriptors from its underlying parsers. For example, the parse descriptor body for an array has type `Seq Int [PD]` including the number of element errors and parse descriptors of its elements.

```
newtype ErrCode = Ok | Err | Fail
type Span = (Offset, Offset)
data Body = Unit
          | Pair PD PD
          | Or (Either PD PD)
          | Constrain PD
          | Seq Int [PD]
          | Scan (Maybe (Int,PD))
          | Struct [PD]
```

4 Adding Tools

One of the advantages of *embedding* a domain specific language is that terms of the language are just ordinary terms in the host language. In our case, since combinator parsers are written and used within Haskell as well as the rest of the application that uses our library, we are able to use existing Haskell libraries seamlessly. So far we have seen that, by using our library, programmers only need to put effort into describing the formats of their ad hoc data to get parsers for free. In this section, we show how to add another tool by using existing Haskell libraries. However, the instance we will show here is not the only case. For example, it is straightforward to build an error reporter by inspecting the parse descriptor and producing error messages.

4.1 Pretty Printing

A frequent need when dealing ad hoc data is to make sources readable. This can be achieved by displaying the data in a suitable layout. Our basic idea is to process the parsing result using Haskell's pretty-printing library [8]. Since the type of the representation result shows enough information of the built-in structures for ad hoc data, we can convert the representation result to a *pretty document* of type `Doc` with a specific layout according to its type. To do so we introduced a type class `Prettytype`.

```
class Prettytype a where
  pprint :: a -> Doc
```

It is straightforward to make a type an instance of `Prettytype`. We have instantiated some base types. For example, we use a standard document generator `double` to make `Double` an instance.

```
instance Prettytype Double where
  pprint = double
```

We have provided instances for some advanced types, which might result from the ad hoc data parser combinators. For example, the representation result of the parser `seqp` has type `[]`. We use a document combinator `cat` to instance it.

```
instance (Prettytype a) => Prettytype [a] where
  pprint xs = cat (map pprint xs)
```

Users can express their preferred layout by instantiating with custom types. For example,

```
data Date = Date Int Int Int
instance Prettytype Date where
  pprint (Date y m d) = pprint y <>
                        (char '.') <>
                        (pprint $ f m) <>
                        (char '.') <>
                        pprint d
    where f m = case m of
                  1 -> pprint "Jan"
                  2 -> pprint "Feb"
                  ... ...
```

We have defined a function which takes an ad hoc parser and an ad hoc stream as input and generates a *pretty document* for the further processing.

```
adhocpprinter :: (Prettytype a) => AdhocParser t a -> [t] -> Doc
```

5 Practical Considerations

In this section, we compare our library to two other approaches: using a traditional parsing library, Parsec, and using a data description language that generates C code, PADS/C. We wish to demonstrate the utility of our library with respect to three criteria: execution time of the resulting parsing functions, conciseness of the specifications, and the ability to collect error information. For the purpose of our evaluation, we built a realistic test scenario related to the motivating example from our domain of interest, i.e., communication protocols. The application is a packet sniffer used for network troubleshooting and analysis. Its functionality is to intercept and display TCP/IP packets being transmitted or received over the Ethernet network. Its main task is to interpret a raw buffer into a well-formed packet according to the packet format of different protocols and the structure of protocol stack. We implemented the TCP/IP packet parser using Parsec, PADS/C, and our library.

To make sure all the experiments work on the same input, we use the packet capture library pcap [2] to capture and dump raw network traffic from a live connection into logs for further processing. The logs were loaded with different sizes from 0 to 1,000 packets. The size of each packet is between 54 to 1514 bytes. One sample already-captured packet record (hexadecimal digits) looks as follows:

```
005056F975DD000C29EA09AE08060001080006040002000C29EA09AEC0A81780
005056F975DDC0A81702
000C29EA09AE005056F975DD08004500002C454200008006CA78D1558193C0A8
17800050801F103B2CCCD2387D516012FAF065130000020405B40000
```

Data sources described with ad hoc data formats frequently contain errors. In our scenario, for example, errors might be caused by disturbances in the network. To evaluate execution time when processing erroneous data, we built another suite of logs. They are similar to the previous ones but with randomly injected one single bit-flip error per packet.

5.1 Performance

We compared the execution time of a parser built using our library and one using Parsec combinators. We used GHC 6.8.1 with -O2 and carried out all experiments on the same computer running Windows XP with a 1.73GHz processor and 1GB RAM. Each experiment was repeated 3 times and the lowest value was taken.

Figure 2 shows the execution time results where the X-axis shows the size of the log in bytes and the Y-axis shows the execution time in seconds. Our library performs as well as Parsec for the error-free logs. In the case of logs with errors, the Parsec parser runs very fast since it fails at the first unrecoverable error, while our parser continues to work. It is also worth comparing the performance of our parser for logs without errors and for logs with errors. The difference is accounted for by the time needed to construct the parse descriptor.

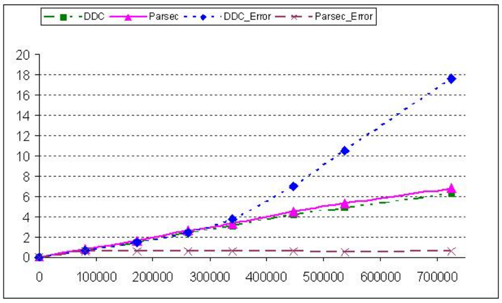

Fig. 2. Execution time results

5.2 Lines of Code

Table 1 shows the lines of code for equivalent Parsec, PADS/C and our own packet parsers. Our parser and the PADS/C description have approximately

Table 1. Lines of code results

	Parsec	PADS/C	Our library
Lines of code	135	98	101

the same size which is more than 30 lines shorter than the Parsec parser. The expected gain in code length is primarily because of the extra code that has to be written in Parsec to handle semantic constraints and physical layout. Table 2 illustrates how we chose to deal with constraints in Parsec and compare it with the PADS/C code and ours. The case we look at is the field for the length of an IP packet header, `ihl`: an integer value described with 4 bits(1 hexadecimal digit in the case of logs) and with a value of at least 5.

– **Physical layout**
  In Parsec, the parser for integer values can not be parameterized to specify physical layout. Thus we have to define a special function `hexDigitN` to build an integer from a chunk of hexadecimal digits. This has to be done for all other cases where the physical layout is needed. In PADS/C, there are primitives for expressing almost all the frequently used data formats, e.g., `Puint8_FW(:1:)` is an unsigned 8-bit integer described in 1 character. In our library, we also cover the most frequent cases, e.g.,`intlen 1` is an integer described in 1 character, and we provide means for extending the library with new primitive parsers including physical layout(see Section 3.2).
– **Constraints**
  In order to deal with semantic constraints in Parsec, we also have to put some effort. Since `GenParser` is a `State Monad`, we first introduce state `MyState` to record the positions of errors, and then update the state during parsing. Of course, more effort would be put in for more precise error reporting. In contrast, both PADS/C and our library include primitives for dealing with constraints.

5.3 Error Detection

Ad hoc data processing needs extensive error handling and good parser error messages. Our parser is able to accurately report syntactical errors i.e., physical layout mismatching as well as semantic errors i.e., constraint violations. Let us reuse the example of Section 5.2. Suppose that there are two contiguous packets containing errors as follows: one has an illegal character on the `ihl` field, and one does not satisfy the condition on this field. Since the syntactical error is unrecoverable, our parser skips the remaining part of the first packet and continues to parse the second one. Meanwhile, it records error information into its parse descriptor. Then we can produce an error message with the precise position and the cause of errors as well as the unexpected input from the data source by inspecting the parse descriptor from the outermost layer to the innermost layer.

Table 2. Code for expressing constraints and physical layout.

Parsec	`type MyState = [(SourcePos, SourcePos)]`
	`hexDigitN :: Int -> GenParser Char MyState Int`
	`hexDigitN 0 = return 0`
	`hexDigitN n =`
	`...`
	`constrain :: (a -> Bool) -> GenParser Char MyState a`
	`                            -> GenParser Char MyState a`
	`constrain f p =`
	`  do pos1 <- getPosition`
	`     p_value <- p`
	`     if (f p_value)`
	`        then return p_value`
	`        else do pos2 <- getPosition`
	`                state1 <- getState`
	`                setState ((pos1,pos2):state1)`
	`                return p_value`
	`ipv4packet =`
	`  do ...`
	`     ihl <- constrain (>=5) (hexDigitN 1)`
	`     ...`

PADS	`Ptypedef Puint8_FW(:1:) ihl_t:`
	`ihl_t x => {x>=5}`

DDC	`ipv4packet =`
	`  do ...`
	`     ihl <- constrainp (>=5) (intlen 1)`
	`     ...`

```
success with error:
<sequence error>
record 1 <baselen type error> <(31,31)> unexpected input:
...09AE08004>>>H<<<00164464...
record 2 <constraint error> <(31,31)> unexpected input:
...75DD08004>>>3<<<0005DC42...
```

Similarly, PADS/C gives:

```
warning: Error [in Puint8_FW_read]: at record 1 at byte 31
         Invalid ASCII character encoding of a number
[record 1]...09AE08004>>>H<<<00164464...
warning: Error [in ihl_t_read]: at record 2 at byte 31:
         Typedef constraint error
[record 2]...75DD08004>>>3<<<0005DC42...
```

In contrast, Parsec halts parsing and discards the remaining part of the input. It can only report the first error:

```
Left (line 1, column 31):
unexpected "H"
expecting hexadecimal digit
```

6 Related Work

Ad hoc data formats have been in the focus of tool developers for a long time. Unix provides a number of utilities that can be pipelined to form filters. Scripting languages are often used to put together parsing utilities. The problem with this kind of approaches is that they do not result in a description of the data format in the program and are thus difficult to keep consistent with implementations and are of no use when changes are made to the data format.

Modern approaches are instead based on domain specific languages used to define the formats. These languages are based on dependent types and are compiled to different kind of processing tools in some target programming languages.

PacketTypes [12] is used to write programmatic descriptions of network protocol packet formats. From these descriptions, a compiler can generate parsers and other tools for packet processing in C. To our knowledge, [12] were the first to use types for packet specifications. There is only one basic type, *bit*, and type constructors for repetition and sequencing. In order to cope with data dependency, fields are allowed to have *attributes* that can be referred to in restriction clauses. There are also ways for overlaying a packet specification in another specification's field — typically in the payload field for layered protocols.

DataScript [3] uses a language of types to describe and manipulate binary data formats and generates libraries in Java. However, it addresses a very specific domain and has been tested to deal with Java byte code. Furthermore, it assumes that the data is error-free, i.e., if an error is detected, parsing needs to halt.

The PADS project [1] has produced a family of data description languages [5] that are used as sources to generate C (PADS/C) and ML (PADS/ML). The languages offer a wider range of basic types, recursive definitions and full blown type dependencies. The compiler generates a type and a parsing function for each type. For base types and type constructors, the target types are fixed. Changing this requires changes to the compiler, as does the addition of new base types. The embedded language we have implemented has clear advantages precisely in this matter. A difficulty with PADS/C is that the compiler has to be modified in order to add tools. This has been recently remedied for PADS/ML [4] taking advantage of the expressiveness of ML's module system.

Parsec [11] is a monadic parser combinator library for Haskell. It is able to process thousands of lines per second making it suitable for industrial-strength purposes. Our library is similar to Parsec in many respects. In both cases, parser combinators are provided to build more advanced parsers, e.g., union, choice, sequence, etc. However our application domain differs from Parsec's that addresses context-sensitive, infinite look-ahead grammars, while we target ad hoc data formats. Our library provides combinators for expressing physical layout, semantic constraints and field dependencies. We also generate parse descriptors that record information about errors in the sources that can be used by the applications.

The Pickler library [10] is a combinator library implemented in Haskell. It builds picklers and unpicklers to convert data from an internal representation (e.g., a Haskell data type) into to external data format (e.g., a stream of bytes) back and forth. There are some similarities with our work. In both cases the programmer describes data formats by composing primitives using combinators, rather than writing tools like parsers and picklers by hand. The pickler combinators tie together the pickling and unpickling actions in a single value, i.e., a value of type a is interpreted to *pickle* and *unpickle* functions respectively, while our DDC library produces an internal representation and a parser descriptor. The primitives and combinators in [10] are analogous to our ad hoc data descriptors. However, since [10] has a separate goal on a different scale, it does neither describe the physical layout and semantic properties nor does it include dependencies.

7 Conclusions and Further Work

We have presented an embedding of the Data Description Calculus in Haskell. It provides us with a domain specific language for describing ad hoc data formats while having access to a complete programming language for building tools and applications. The descriptions are parsing functions with a monadic interface and are thus easy to use together with other libraries in Haskell. We have organized our embedding in such a way that it is easy to extend with new primitive combinators. For doing so we made use of type classes. We have also presented an example showing how to integrate other tools and comparing our embedded language with another domain specific language with a similar semantics.

There are many things we would like to do after testing our library more extensively. We would like to add some useful primitive types that occur frequently in many domains, like dates and IP addresses. We would like to extend it with some standard tools like error statistics, generation of XML schemas, generation of abstract syntax trees, etc.

We are convinced that there is a lot of work to do regarding the implementation. For example, we have only implemented a straightforward version of backtracking and this might need to be revised for efficiency considerations. We are also interested in studying how we could add rewrite rules to GHC to improve execution time.

Beyond, we believe that our library might be an excellent setting for tools that discover the data formats from examples and generate data descriptions automatically [7]. Furthermore, we are intrigued by the connections between the DDC and the theory of formal languages.

Acknowledgment

We want to thank John Hughes for insightful comments and suggestions.

References

1. Pads. http://www.padsproj.org. [Online; accessed 22-June-2008].
2. Pcap. http://www.tcpdump.org/pcap.htm. [Online; accessed 10-Oct-2008].
3. Back, G. Datascript - a specification and scripting language for binary data. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering* (London, UK, 2002), Springer-Verlag, pp. 66–77.
4. Fernández, M. F., Fisher, K., Foster, J. N., Greenberg, M., and Mandelbaum, Y. A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers. In *PADL* (2008), P. Hudak and D. S. Warren, Eds., vol. 4902 of *Lecture Notes in Computer Science*, Springer, pp. 133–149.
5. Fisher, K., and Gruber, R. Pads: a domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 295–304.
6. Fisher, K., Mandelbaum, Y., and Walker, D. The next 700 data description languages. In *POPL '06* (New York, NY, USA, 2006), ACM Press, pp. 2–15.
7. Fisher, K., Walker, D., Zhu, K. Q., and White, P. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL* (2008), G. C. Necula and P. Wadler, Eds., ACM, pp. 421–434.
8. Hughes, J. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (London, UK, 1995), Springer-Verlag, pp. 53–96.
9. Hutton, G., and Meijer, E. Monadic parsing in haskell. *J. Funct. Program 8*, 4 (1998), 437–444.
10. Kennedy, A. J. Pickler combinators. *Journal of Functional Programming 14*, 6 (2004), 727–739.
11. Leijen, D., and Meijer, E. Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.
12. McCann, P. J., and Chandra, S. Packet types: Abstract specification of network protocol messages. *SIGCOMM Comput. Commun. Rev. 30*, 4 (2000), 321–333.

Paper II

A Domain Specific Approach to Network Software Architecture

Assuring Conformance Between Architecture and Code

Yan Wang and Verónica Gaspes

To appear in *Proc. of the 4th International Conference on Digital Telecommunications (ICDT 2009)*
Colmar, France, July 2009

A Domain Specific Approach to Network Software Architecture

Assuring Conformance Between Architecture and Code

Yan Wang and Verónica Gaspes
Halmstad University
CERES
Halmstad, Sweden
yan.wang@hh.se, veronica.gaspes@hh.se

***Abstract*—Network software is typically organized according to a layered architecture that is well understood. However, writing correct and efficient code that conforms with the architecture still remains a problem. To overcome this problem we propose to use a domain specific language based approach. The architectural constraints are captured in a domain specific notation that can be used as a source for automatic program generation. Conformance with the architecture is thus assured by construction. Knowledge from the domain allows us to generate efficient code. In addition, this approach enforces reuse of both code and designs, one of the major concerns in software architecture. In this paper, we illustrate our approach with PADDLE, a tool that generates packet processing code from packet descriptions. To describe packets we use a domain specific language of dependent types that includes packet overlays. From the description we generate C libraries for packet processing that are easy to integrate with other parts of the code. We include an evaluation of our tool.**

Categories and Subject Descriptors

D.1.2 [Programming Techniques]: Automatic Programming-*Program Synthesis*.

Keywords

network software, software architecture, dependent types, program generation.

I. Introduction

Network software is typically organized according to a layered architecture that is well understood. However, writing correct and efficient code that conforms with the architecture still remains a problem [1]. In other domains this problem has been dealt with using automatic code generation from specifications. The most well known case is compiler technology where lexical analyzers are generated automatically from regular expressions [2] and parsers are generated from context free grammars [3]. More recently, the same approach has gained attention in other domains like communication services [4], cryptography [5] and finantial contracts [6]. We propose to use a similar approach for network software: the architectural constraints are expressed in a language that is used as a source for automatic program generation. Conformance with the architecture is thus automatically assured and knowledge of the domain allows us to generate efficient code. Further, other tools, for example for automatic testing and for evaluation of nonfunctional properties, can use our language as a source. One of the major concerns of software architecture is the possibility of reusing not only code but also designs. With our language based approach the designs are encoded in the constructs of the language. Reuse is thus enforced *by construction*.

As part of the layered architecture, protocol specifications include packet specifications. These are written in a highly structured informal notation that describes header fields with lengths, constraints and other properties. On the other hand, programmers that implement network protocols have to deal with packets as sequences of bits that have to be interpreted according to the specification, at the same time converting between byte order in the network device and the processor. This is most frequently done in the C programming language using offsets, bit masks and dedicated functions which are difficult to relate to the specification. Also, code fragments referring to a field or to a constraint on a field can appear more or less anywhere in the code. All this makes it difficult to keep track of the correspondence between the architecture and the implementation and to make modifications to the programs that follow slight modifications in packet specifications.

In this paper we address this problem using a domain specific language based approach. We introduce PADDLE, a tool that generates packet processing code from packet descriptions made in a dedicated language. Packets are described using dependent types in a notation that also includes a construct for packet overlays. From the packet descriptions we generate C libraries for packet processing that are easy to integrate with other parts of the code. The choice of dependent types allows us to deal with semantic constraints on fields and among fields. When using our tool, packet descriptions are kept in one place, can be modified if needed, and the packet processing code is generated automatically. Both our language and its implementation as a tool are based on recent work that formalizes the treatment of ad-hoc data formats [7] that we have adapted to packet processing. Our tool is part of a larger project using the same language based approach to the development of network software.

The contributions of this work are as follows. In Section II we introduce the components of our notation for describing packets, including physical layout, dependency of fields and semantic constraints. We also show the operations for layering packets of a protocol stack. In Section III we show how we generate code for a packet processing library in C. We also discuss some of the characteristics of the generated code. In Section IV we compare the results of using our tool to generate a packet sniffer with an existing sniffer programmed directly in C. The paper concludes with a section on related work and one on conclusions and future work.

II. PADDLE: A PACKET DESCRIPTION LANGUAGE

Our language for describing packets is a language of types that resembles structures in C: header fields are given names and types. However, it is richer than ordinary C structures. The base types can include information on the number of bits occupied by the field. The type of a field can refer to other fields, so that the value of a field can be the number of bits needed for another field. Boolean functions can be used in types to constraint the values that a field might take. To build layers of headers we provide a construct for overlaying packet descriptions. In what follows we present the language in detail.

A. Field Types

A field type is either a primitive type qualified by the amount of bits needed for its representation, a type constrained by a boolean function, an alternative between two types or an array type:

$$\tau \ ::= \ \mathtt{B(e)} \,|\, \tau\{\mathtt{e}\} \,|\, \tau\mathtt{+}\tau \,|\, \tau\mathtt{[e]}$$

a) Base Types: Given that `B` is a primitive type in C and e is an integer expression in C, `B(`e`)` is a field type in PADDLE. A field with this type occupies as many bits as the value of the expression. If no expression is used, the default size of the type `B` is assumed. The expression may refer to the value of previous fields. As an example, a field can have type `int(10)` to indicate that only 10 bits are used in the buffer for a value of type integer for that specific field.

b) Constrained Types: Given that τ is a field type and e is a boolean expression in C, a field with type $\tau\{e\}$ is a field of type τ whose value satisfies the condition e. References to other fields can be made in the boolean expression. As an example, a field can be typed

```
ihl : int(4){ihl>=5}
```

The field `ihl` uses 4 bits in the buffer and its value should be an integer greater or equal than 5.

c) Sum Types: Given that τ and τ' are types, a field with type $\tau+\tau'$ is a field with type τ, alternatively τ'. Using this type we can describe fields of a variety of forms, as for example

```
f : char(4){f=='a'}+int(4){f==97}.
```

d) Array Types: Given that τ is a type and e is an integer expression in C, a field with type τ`[`e`]` is a sequence of length e of elements of type τ. The expression e may refer to other fields. For example, a field with type `int(4)[100]` is a sequence with 100 integer elements, each occupying 4 bits. If the empty `[]` is given, the field can be a sequence of any length.

In PADDLE packets are described putting together fields in records. As an example consider the definition of a packet for IP version 4:

```
ipv4 = {
  version : u8(4){version==4};
  ihl     : u8(4){ihl>=5};
  totallen: u16;
  id      : u16;
  flags   : u8(3);
  fragoff : u16(13);
  ttl     : u8;
  protocol: u8;
  hdchksum: u16;
  srcaddr : u8[4];
  dstaddr : u8[4];
  option  : u8[ihl*4-20];
  payload : u8[totallen-ihl*4];
};
```

where `u8` and `u16` are just abbreviations for `unsigned char` and `unsigned short`.

B. Overlays

In addition to field records, PADDLE provides overlays as a way of describing packets. Overlays are used to encapsulate a packet within another and they can be nested. In this way packet specifications can be made following the layered architecture, in a modular way. Given packets *pname* and *pname'*, a new packet can be defined by placing *pname'* within one field of *pname*:

$$\mathtt{pname.fname\ <\text{-}>\ pname'}\{\mathtt{e}_1\mathtt{,\ ...,\ e}_m\}$$

The overlay includes a list of conditions that have to be satisfied. For example, assuming that the packet type `tcp` has been defined, the overlay that describes `tcp` over `ipv4` is

```
ipv4.payload<->tcp {ipv4.protocol==6}
```

As anticipated, the software architecture, in this case a layered architecture, is encoded in the constructions of the language. More domain specific constraints are also part of the language. In the case of PADDLE this is the fact that packet headers are records of header fields. This language based approach provides us with a source for code generation. The kind of code that is generated is also domain specific. In the case of PADDLE, we generate packet processing libraries, with fragments that can be used to parse packets, to marshal packets and to do some simple processing like filtering. These libraries can then be integrated with the rest of, for example, a protocol implementation or other network software written in C. The resulting program is improved in that there is a localized packet description, making the program more maintainable. The programmer in turn avoids dealing with the low level details involved in the implementation of packet processing. From a software architecture perspective, two central concerns are guaranteed by construction:

- conformity between architecture and code,
- reuse of code and designs.

III. CODE GENERATION

The C programs we generate from packet descriptions include in-memory representations, parsing functions and marshaling functions. The parsing functions are used when receiving a packet while the marshaling functions are used when sending a packet.

A. The representation types

To each packet description we associate a C structure with the same fields as in the packet specification, the *representation type*. The types of the fields in the structure are obtained form PADDLE field types by erasing type dependencies and constraints. The translation from PADDLE field types to C types is as follows:

- A base type `B(e)` is translated to the C type `B`.
- A constrained type $\tau\{e\}$ is translated to the translation of the underlying type τ.
- A sum type τ+τ' is translated to a C union of the translations of τ and τ'.
- An array type τ`[e]` is translated to a C array with fixed size *e* whenever *e* is present and does not contain free variables. Otherwise, it is converted to a pointer.

Overlays are translated to C unions. If at some point in a program we were interested in retrieving the value of a complete packet we would get a value in its representation type. As an example consider the PADDLE packet description fragments

```
ethernet = {
  dstadd : u8[6];
  srcadd : u8[6];
  ptype  : u16;
  payload: u8[];
};

ethernet.payload<->arp
  {ethernet.ptype==0x806};
ethernet.payload<->ipv4
  {ethernet.ptype==0x800};
```

The representation types are

```
typedef union{
  arp *arp;
  ipv4 *ipv4;
  u8 *payload;
}ethernet_payload_u;

typedef struct{
  u8 ethernet_dstadd[6];
  u8 ethernet_srcadd[6];
  u16 ethernet_ptype;
  ethernet_payload_u *ethernet_payload;
}ethernet;
```

However, when parsing or marshaling a packet we are only interested in identifying the fields and checking that they comply with the constraints expressed in the PADDLE description. The packet will reside in some buffer and we will try to avoid copying the contents of the buffer. In order to go through the buffer we introduce a number of auxiliary types that help in the implementation of the parsing and marshaling functions.

To begin with we introduce *field handles* that are used to keep track of the fragment of the buffer where a given field resides. Field handles have type

```
typedef struct{
  char * buffer;
  u_int index;
  u_int offset;
}field_h;
```

including a pointer to the buffer where the packet resides, an index to the buffer bit being read or written, and a count of the number of bits representing this field.

If needed, for example to test whether some condition holds, the value of a field can be easily extracted from the buffer via its handle using a predefined function `FieldRead`.

With these field handles, we can associate a type to each packet description, a *packet handler* type, that is a structure of field handles. For overlays we use unions. Translating a packet description to its packet handle type is straightforward. As an example consider Ethernet packets again. The packet handle has type:

```
typedef union{
  arp_h   *arp_h;
  ipv4_h  *ipv4_h;
  field_h *payload_h;
}ethernet_payload_h_u;

typedef struct{
  field_h *ethernet_dstadd_h;
  field_h *ethernet_srcadd_h;
  field_h *ethernet_ptype_h;
  ethernet_payload_h_u *ethernet_payload_h;
}ethernet_h;
```

B. The parsing function

For each packet description in PADDLE we also generate a parsing function with prototype

```
packet_h *parse_packet(char *buffer,
                       u_int bitIndex);
```

where

- `packet_h` is the packet handler type we have generated for the given packet description. For example, for Ethernet packets it will be `ethernet_h`.
- `buffer` is the memory area where the incoming packet is stored.
- `bitIndex` is an index indicating at which bit parsing should commence.

The code generated for the parsing function depends on the PADDLE type. Some illustrative cases are:

- For fields of a base type an offset has to be moved forward as many bits as required. For example, the field `ptype` of an Ethernet packet is described as `ptype:u16` in PADDLE. This is translated into

```
offset = 16;
p->ethernet_ptype_h =
  FieldMake(buffer,index,offset);
index += offset;
```

where the default bit length of `u16` is `16` and `p->ethernet_ptype_h` is the field handle initialized by a predefined function `FieldMake`.

- For overlays, the conditions have to be evaluated and the corresponding parsing functions have to be called. For example, an Ethernet packet is an Ethernet_ARP packet or an Ethernet_IPv4 packet depending on the value of the field `ethernet_ptype`. The value of `ethernet_ptype` is read from the buffer and this guides further parsing:

```
FieldRead(p->ethernet_ptype_h,
          &ethernet_ptype);
if(ethernet_ptype==ARPType){
  arp_index =
    getIndex(p->ethernet_payload_h
              ->payload_h);
  payload_h->arp_h =
    parse_arp(buffer, arp_index);
};
if(ethernet_ptype==IPv4Type){
  ipv4_index =
    getIndex(p->ethernet_payload_h
              ->payload_h);
  payload_h->ipv4_h =
    parse_arp(buffer, ipv4_index);
};
```

- For constrained field types, the value of the field has to be extracted and the condition has to be tested. For example, if the field `version` of an Ethernet_IPv4 packet has a value other than 4 it should be discarded:

```
FieldRead(p->ipv4_version_h,
          &ipv4_version);
if(!(ipv4_version==ARPType))
  return NULL;
```

If something goes wrong during parsing, the function returns `NULL`.

C. The marshaling function

For each packet description we also generate a marshaling function with prototype

```
int *marshal_packet(
    char *buffer,
    u_int bitIndex,
    packet_inmemory *packet);
```

where

- `buffer` is the memory area where the outgoing packet is stored,
- `bitIndex` is the index indicating at which bit marshaling should commence.
- `packet` is the in-memory representation of the outgoing packet.

The return value of type `int` will be non-negative on success.

For example, the marshaling function for the Ethernet packet has the following prototype:

```
int *marshal_ethernet(
    char *buffer,
    u_int bitIndex,
    ethernet *p);
```

The marshaling function converts an in-memory representation to a sequence of bytes for transmission. It has to do the following tasks.

- Feed the value of each field to the packet buffer using a predefined function `FieldWrite`. For example, the following code is used to write the value of the field `ptype` into the buffer (assuming `u16` in PADDLE):

```
offset = 16;
FieldWrite(buffer,index,offset,
           p->ethernet_ptype);
index += offset;
```

- In case of overlays, choose an adequate marshaling function to proceed. For example, `ethernet_payload` is written into the buffer using

```
if(p->ethernet_ptype==ARPType)
  marshal_arp(buffer,index,
              p->ethernet_payload);
if(p->ethernet_ptype==IPv4Type)
  marshal_arp(buffer,index,
              p->ethernet_payload);
```

where the value of `ethernet_ptype` is checked first to guide the further function calls.

D. The generated code

We have put some effort in generating code that makes efficient use of resources. The parsing functions work directly on the buffer that stores the packet. Only when the value of a field is needed, for example for checking a constraint, do we use a variable in the program to store the field. The parsing and the marshaling functions are bit oriented, instead of byte oriented, meaning that fields can use less than a byte and that fields can cross byte boundaries. With this, headers can be very compact, an important issue in protocols for sensor networks.

IV. A PACKET SNIFFER

We used PADDLE to produce a TCP/IP packet sniffer, a program used to intercept and display TCP/IP packets being received over the Ethernet network. In this section we compare the resulting program with Sniffex, the TCP/IP sniffer that follows with the packet capture library libpcap [8].

Figure 1 shows the execution time results where the *X*-axis shows the number of packets and the *Y*-axis shows the execution time in seconds. The difference in performance is accounted for by the time needed to construct the handle for each field.

It is also interesting to compare the size of sources because it provides a hint on the time needed for programming and on the possibility of understanding the complete program.

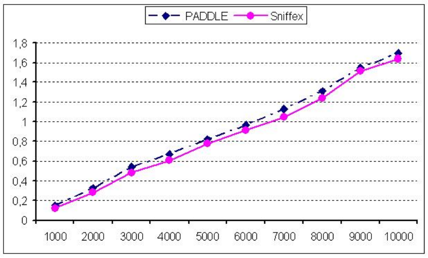

Figure 1. Execution time results

	Lines of code	Generated C code	Executable size
PADDLE	39	297	10720
Sniffex	279	-	9072

Table I
LINES OF CODE RESULTS

Table I shows the lines of code for PADDLE. The lines of code of Sniffex have been adjusted to take into count only the functions that are used in the example. The lines of C code generated from PADDLE include both .h and .c files. We are not so satisfied with the size of the executable and we think there are opportunities for optimizations.

V. RELATED WORK

We are not the first ones to be interested in using high level formal descriptions to generate programs that deal with tedious tasks. In this section we mention some of the earlier work using types as specifications for program generation.

The use of types to describe packets with the purpose of generating packet processing code was introduced in [9]. Types are used externally and a compiler generates parsing functions that can be rapidly adapted to implement packet filters. To our knowledge [9] were the first to use types for packet specifications. There is only one basic type, *bit*, and type constructors for repetition and sequencing. In order to cope with data dependency, fields are allowed to have *attributes* that can be referred to in restriction clauses. There are also ways for overlaying a packet specification in the field of another specification — typically in the payload field for layered protocols. In our work we replace the ad-hoc notion of attributes from [9] with a richer system of dependent types. We also allow for a richer set of basic types instead of just bits. In this way we can deal with more semantic constraints and consistency conditions that are beyond the scope of [9].

The idea of using dependent types for expressing constrains and physical representations was introduced in [10], in the more general setting of ad-hoc data processing. There is, however, no way of expressing overlays. More recently, in [7], the semantics of dependent types as a formalization of ad-hoc data formats was presented. In our work we adapted the semantics for the specific case of network packets.

Types have also been used in [11] to describe and manipulate binary data formats. They address however another domain, has been tested to deal with Java byte code, and does not provide mechanisms for layering data in fields.

For dealing with the contents of payloads, [12] introduces a notation for describing data types in a language independent way and comes with tools to convert these descriptions into binaries that can be used to store or transmit instances of these data structures. In our context it would typically be used to describe the kind of data an application wants to send.

VI. CONCLUSION AND FUTURE WORK

We have shown that taking a domain specific, language based approach we can address two of the main concerns of software architecture:

- conformity between architecture and code,
- reuse of code and designs.

We have illustrated this with PADDLE, a tool to assist in the implementation of protocol stacks, a domain where software is organized in layers. Using our tool packets are described in language of dependent types. In PADDLE types are used to describe both the physical layout of packets and semantic constraints on their fields. These descriptions are then the source for program generation. We generate C code for both interfacing the network to the host formats but also for parsing and writing packets from and to the wire. This processing is bit oriented allowing for very compact packet formats suitable for embedded systems. We think that the descriptions in PADDLE are closely related to packet descriptions in protocol specifications and that the resulting programs are modular and thus easy to assess correct, to maintain and to modify.

We are keen to do more extensive experiments with our tool, both to evaluate performance of the resulting code and to identify limitations that might lead to improvements. In the long term we intend to incorporate the packet types of PADDLE as part of the type system of a domain specific programming language for the implementation of protocol stacks. When for efficiency reasons packets should not be read into a data structure, the program inspects the buffer where the packet is placed by the network adapter. In making our type system internal we still want to be able to deal with the binaries storing packets. We plan to use techniques borrowed from [13] and adapt them to a language with types.

REFERENCES

[1] M. Shaw and P. Clements, “The golden age of software architecture,” *Software, IEEE*, vol. 23, no. 2, pp. 31–39, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1605176
[2] M. E. Lesk and E. Schmidt, “Lex - a lexical analyzer generator,” AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Tech. Rep.
[3] S. C. Johnson, “Yacc: Yet another compiler-compiler,” AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Tech. Rep. 32, 1975.
[4] C. Consel and L. Réveillère, *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, ser. Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004, no. 3016, ch. A DSL Paradigm for Domains of Services: A Study of Communication Services, pp. 165 – 179.
[5] J. Lewis, “Cryptol: specification, implementation and verification of high-grade cryptographic applications,” in *FMSE ’07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*. New York, NY, USA: ACM, 2007, pp. 41–41.

[6] S. P. Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," in *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2000, pp. 280–292.

[7] K. Fisher, Y. Mandelbaum, and D. Walker, "The next 700 data description languages," *SIGPLAN Not.*, vol. 41, no. 1, pp. 2–15, 2006.

[8] "Pcap," http://www.tcpdump.org/pcap3man.html, [Online; accessed 10-Nov-2008].

[9] P. J. McCann and S. Chandra, "Packet types: abstract specification of network protocol messages," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 321–333, 2000.

[10] K. Fisher and R. Gruber, "Pads: a domain-specific language for processing ad hoc data," *SIGPLAN Not.*, vol. 40, no. 6, pp. 295–304, 2005.

[11] G. Back, "Datascript - a specification and scripting language for binary data," in *GPCE*, 2002, pp. 66–77.

[12] CCITT, "Specification of abstract syntax notation one (ASN.1)," International Telegraph and Telephone Consultative Committee, Tech. Rep., 1988, recommendation X.208.

[13] P. Gustafsson and K. Sagonas, "Efficient manipulation of binary data using pattern matching," *Journal of Functional Programming*, vol. 16, no. 1, pp. 35–74, 2006.