Certified Parsing of Binary Data

Pierre-Évariste Dagand
Cnrs/Lip6/Inria – Whisper

Abstract

This internship offers to design and implement a formally-verified format language in the Coq proof assistant. The rôle of a format language is to describe the structure of binary data. From a format description, one obtains a parser – taking binary data to high-level data-structures – and a serializer – mapping back those data-structures to binary data. The parser is correct if it is the left-inverse of the serializer. We shall strive to provide a mechanically-checked proof of correctness of the parsers/serializers obtained from our format language. This internship will be held at LIP6 (UPMC, Paris).

The foundation of our computerized civilization is built from *infrastructure software*: hypervisors, operating systems, servers, *etc.* Such software implements complex policies to ensure a fair and secure access to computing resources. This access is mediated by low-level mechanisms directly manipulating the devices. As a result, writing infrastructure code is tedious and error-prone. It is crucial to abstract away the low-level details by providing high-level abstractions. Domain-specific languages (DSLs) are a means to that end [Muller et al., 2000].

A typical DSL is the Devil language for hardware access [Mérillon et al., 2000]. An OS programmer describes the register set of a hardware device in the high-level Devil language, which is then compiled into a library providing C functions to read and write values from the device registers. In doing so, Devil frees the programmer from having to write extensive bit-manipulation macros or inline functions to map between the values the OS code deals with, and the bit-representation used by the hardware: Devil generates code to do this automatically.

This internship is concerned with the *parsing of binary data.* Parsing is ubiquitous in infrastructure code: drivers must parse the output of devices, network stacks must parse packets to process them, applications must parse binary documents (PDF, png, *etc.*) to display them. A bug in these intricate parsing codes can lead to crashes or, worse, security vulnerabilities. A network packet sniffer might crash upon processing certain packets [CVE details, 2014c]. A PDF viewer might execute arbitrary code upon opening a carefully crafted document [CVE details, 2014a]. A web browser might accept forged RSA signatures upon receiving a carefully crafted certificate [CVE details, 2014b].

To tackle this issue, we propose to design and implement a *format language* [Burgy et al., 2011, Levillain, 2014, Bangert and Zeldovich, 2014]: a domain-specific language, akin to a parser generator [Johnson, 1979], offering specialized abstractions for parsing and serializing binary data. Conceptually, the purpose of a format language is to relate an external format – a lump of binary data – to an internal representation – a semantic object – in some programming language. The external format may involve checksum computation, redundant information, or even compression. The internal representation is typically a data-structure in the target programming language, such as C, OCaml or Coq. Formally, the relation between binary data and its semantic domain is witnessed by a parsing function that is the left-inverse of a serialization function, which maps semantic values back to an external format. This amounts to a round-trip property: serializing composed with parsing yields the identity function on (semantic) values.

By seamlessly integrating programs and proofs, dependently-typed languages, such as Coq [The Coq Development Team], Agda [Norell, 2007], or Idris [Brady, 2013], enable the development of a provably-correct *and* extensible format language [Morrisett et al., 2012]. Executable code can be obtained by extraction to OCaml, or code generation to Cminor [Leroy, 2009] or even directly to x86 [Kennedy et al., 2013] (whose semantics have all been formalized in Coq). We also benefit from a rich semantic domain that amounts, at the very least, to Martin-Löf type theory with inductive and coinductive types. While usual interpretations of formats depend on the target language's data-structures, we are here offered the opportunity to develop a genuine denotational semantics. For example, we have given a compositional interpretation [Kennedy et al., 2013] of the x86 binary opcodes (represented as tuples of Booleans) into a Coq data-type (representing the instruction set, including its maze of addressing modes).

In this internship, we shall design, implement, and verify a compositional format language. Concretely, we are aiming at:

- developing a format DSL embedded in the Coq proof assistant;
- taking as input binary data in the form of Coq tuples of Booleans;
- producing internal representations in the form of Coq types;
- illustrating stream transformations, such as checksum and offset computation;
- supporting extraction of the parser and serializer to OCaml.

The team: This internship will be held at LIP6 (UPMC, Paris) in the Whisper team (INRIA) headed by Gilles Muller. It will be supervised by Pierre-Évariste Dagand. The Whisper team is exploring avenues for collaboration with the Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI): this project could become the cornerstone of a stimulating cooperation revolving around the compositional development of certified software.

Student's profile: We are looking for a student with experience in an interactive theorem prover (Coq, or Isabelle) or a dependently-typed programming language (Agda, or Idris). A motivated student with a strong background in functional programming (OCaml, or Haskell) could certainly learn to use Coq along the way [Pierce et al., 2014, Chlipala, 2011]. Acquaintance with the C programming language and the Unix environment (GNU/Linux, or BSD) is recommended. Having a knack for low-level programming is highly appreciated.

References

J. Bangert and N. Zeldovich. Nail: A practical tool for parsing and generating data formats. In *OSDI*, pages 615–628, Oct. 2014.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 2013.

L. Burgy, L. Réveillère, J. L. Lawall, and G. Muller. Zebu: A language-based approach for network protocol message processing. *IEEE Trans. Software Eng.*, 37(4):575–591, 2011.

A. Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2011. URL `http://adam.chlipala.net/cpdt/`.

CVE details. Acrobat vulnerabilities, 2014a. URL `http://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-497/`.

CVE details. Mozilla NSS vulnerability, 2014b. URL `http://www.cvedetails.com/cve/CVE-2014-1568/`.

CVE details. Wireshark dissectors vulnerabilities, 2014c. URL `http://www.cvedetails.com/vulnerability-list/vendor_id-4861/Wireshark.html`.

S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. 1979.

A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: The world's best macro assembler? In *PPDP*, pages 13–24, 2013.

X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

O. Levillain. Parsifal: a pragmatic solution to the binary parsing problem. In *LangSec Workshop at IEEE Security & Privacy*, May 2014.

F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *OSDI*, pages 17–30, Oct. 2000.

G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*, pages 395–404, 2012.

G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Mérillon, and L. Réveillère. Towards robust OSes for appliances: A new approach based on domain-specific languages. In *SIGOPS European Workshop*, pages 19–24, 2000.

U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. *Software Foundations.* Electronic textbook, 2014. URL `http://www.cis.upenn.edu/~bcpierce/sf`.

The Coq Development Team. *The Coq Proof Assistant Reference Manual.* URL `http://coq.inria.fr/coq/refman`.