Cogent$^{\Uparrow}$: Giving Systems Engineers A Stepping Stone

Extended Abstract

Zilin Chen
UNSW, Australia
zilin.chen@student.unsw.edu.au

Abstract

This paper presents our vision and work in progress on deploying a high-level functional language with a rich and accessible type system for better modelling and verifying systems programs.

CCS Concepts • **Software and its engineering** → *Software verification*; **Abstraction, modeling and modularity**; **Specialized application languages**;

Keywords Cogent, Systems programming, Refinement

1 Background

We have previously presented the Cogent project for reducing formal verification effort by code and proof co-generation [2, 10, 11]. We showed that it is theoretically possible to produce file systems (FSes) with end-to-end verification using the Cogent framework.

Cogent is a restricted purely functional language that we designed to enable both destructive updates, which are essential in systems programming, and equational reasoning, which would ease manual proof of systems' behaviour. The language features a uniqueness type system [13] to mask destructive updates while exposing a functional semantics for equational reasoning. It is restricted in the sense that loops and general recursion are both disallowed. Thus any real world Cogent program needs to be paired with some C code by means of the foreign function interface (FFI), in order to implement a) datatypes and algorithms which rely on sharing writable pointers (thus violating uniqueness); b) impure code; c) looping combinators and iteration schemes; and d) interfaces to Linux kernel code.

Figure 1 summaries the Cogent framework (ignoring the blue components for now, which are our new additions), with all user input indicated with dotted lines.

To develop a (file) system, *systems engineers* must supply both source programs written in Cogent and the complementary C code consisting of abstract data types (ADTs), Linux interfaces and so forth, which is linked with the C code generated by the Cogent compiler. Much of this manually-written C code can be shared between systems (e.g. red-black trees, Linux VFS wrappers).

TyDe'17, September 03, 2017, Oxford, United Kingdom

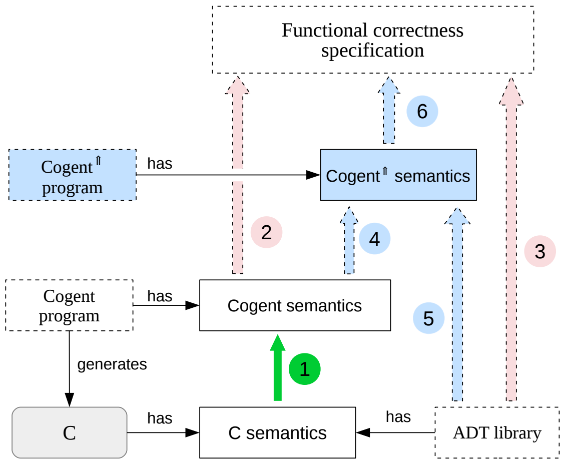

Figure 1. Overview of the Cogent framework

To verify such a system, *proof engineers* define a functional correctness specification in Isabelle/HOL, then manually prove that the Cogent program (pink arrow ②) and the ADT implementations (pink arrow ③) are together a refinement of this abstract specification. The refinement proof that completes the end-to-end verification is generated automatically by the Cogent compiler (green arrow ①).

2 Challenges

For our first version of Cogent, we prioritised the completion of the full compilation and verification pipeline over the fine tuning of usability and performance. This puts us in a position to conduct a series of case studies to pinpoint the weakness of Cogent, and to evaluate whether it is in fact simplifying the goal of developing verified systems. We were able to answer the latter in the affirmative [2], and identified a number of language features and optimisations necessary to address the former. We also identified three more fundamental challenges we need to tackle:

Reliance on Testing Testing is essential even in the presence of formal verification. Formal proof is technically complicated and labour intensive. Effective testing has a great chance of avoiding fruitless attempts to prove broken definitions. But systems code is intrinsically difficult to test and to debug, especially in the Cogent ecosystem, with which the programmers are less familiar. Therefore it would be delightful if Cogent programs are correct by construction and excessive manual testing can be eliminated.

Design for Verification A correct implementation unnecessarily means that it can be formally verified. Systems must be designed in a modular fashion that is amenable to verification. Exactly what designs are easy to verify, however, is not obvious to systems engineers, as they generally lack knowledge or experience in formal verification. In our case studies, we designed a verification-friendly FS from scratch, as well as ports of existing Linux FSes with architecture that mirrored the original C code. [1] Systems designed in these two strategies manifest distinct characteristics in verification: while we were able to prove properties of the former [1], the latter in general does not make use of any of Cogent's advantages for verification, which makes it as difficult as verifying the C code on which it was based.

Level of Abstraction As we explained in Section 1, Cogent heavily depends on its FFI to C, which usually makes it interspersed with a significant amount of C code. For performance reasons and to comply with interfaces used in native C implementations, our ADT library (and wrapper code) is distinctly imperative in flavour. For instance, instead of combinators such as `map` and `fold`, the interface provides unwieldy looping functions which allow for early exit, accumulated state, or other lower-level concerns. Therefore, the unfortunately low-level interface, combined with the uniqueness type system, forces the programmer to think on a lower level than one would expect from an ordinary functional language.

3 Cogent$^{\Uparrow}$ Language

To address the aforementioned challenges, I propose Cogent$^{\Uparrow}$ language, a higher-level general-purpose extension to Cogent, which will play an important role in our framework (blue components in Figure 1).

Cogent$^{\Uparrow}$ mainly differs from Cogent in the following aspects. Firstly, it supports general recursion and recursive datatypes. Secondly, it relaxes the uniqueness constraints of Cogent, which hides the capability details. They together make it possible to prototype a system entirely in Cogent$^{\Uparrow}$. This is in contrast to Cogent, where low-level concerns may invite users to mirror the structure of the native C code, unnoticeably. Finally, Cogent$^{\Uparrow}$ features a dependent type system together with other handy type extensions which allow precise properties to be stated in the types.

The power of dependent type systems is widely acknowledged, however we cannot neglect the steep learning curve and the unfriendly user interface they pose on mainstream programmers. For example, Lindley and McBride [9] show the pains of using dependent types in Haskell. Even in a natively dependently typed language, e.g. Idris [4], it is still quite involved to manually construct proof terms, which are vulnerable to changes.

[1]Mirroring the structure of the C code was not intended.

To alleviate this burden on systems programmers, Cogent$^{\Uparrow}$ is equipped with an SMT-based proof search mechanism, which allows users to concentrate on the algorithms of their programs during prototyping. LiquidHaskell [12] is a successful example in this spirit, and F* applies the idea further to writing and verifying low-level programs such as cryptographic algorithms [3].

With these features, Cogent$^{\Uparrow}$ leads to a workflow which better leverages the Cogent framework. As first step in the development process, systems programmers can prototype fully in Cogent$^{\Uparrow}$ (with the exception of calls into the kernel) and execute the program. Cogent$^{\Uparrow}$'s rich type system diminishes the need for testing, and QuickCheck [5] style automatic testing can be deployed to further reduce manual effort (see Section 4). The functional nature of Cogent$^{\Uparrow}$ language encourages programmers to design the systems with abstraction and modularity in mind. Cogent$^{\Uparrow}$'s semantic is closer to the abstract correctness specification, merely differing in datatypes and algorithmic aspects, where the refinement proof (arrow ⑥ in Figure 1) can be conducted semi-automatically and systematically [6, 8]. Based on the Cogent$^{\Uparrow}$ implementation, the programmers then gradually refine (arrow ④ and ⑤) the implementation to Cogent or to C, taking capabilities and other low-level concerns into account, to reach acceptable performance.

Having our own Cogent$^{\Uparrow}$ has advantages over using existing languages in the wild. As Cogent$^{\Uparrow}$ will be incrementally rewritten to Cogent, it has to be capable of connecting to Cogent seamlessly. Since Cogent$^{\Uparrow}$ is an extension of Cogent, language interoperability comes nearly for free (modulo type system difference). Additionally, much of the toolchain for Cogent, such as the FFI and the code generators, is readily available and can be shared with minimal adaptation.

4 Property-based Testing

Applying QuickCheck (or in general, property-based testing) to systems is not new (e.g. [7]), but the presence of formal verification gives rise to more opportunities. The properties that we have specified for formal verification can be reused for property-based testing. It allows us to test before we attempt the costly one-step manual proof between the functional correctness specification and Cogent / ADTs (arrow ② and ③ in Figure 1). Importantly, it does not incur extra system design overhead to adapt to property-based testing.

In practice, our abstract specification is not executable hence unsuitable for testing; we can nevertheless reach as far as the Cogent$^{\Uparrow}$ level [2], leaving arrow ⑥ as a straightforward refinement proof. A separate paper is in submission, detailing our observation and experiments on this theme.

[2]As a proof of concept, we use Haskell in our experiments instead of Cogent$^{\Uparrow}$ so that we can use the Haskell QuickCheck package as-is.

References

[1] Sidney Amani. 2016. *A Methodology for Trustworthy File Systems*. PhD Thesis. CSE, UNSW, Sydney, Australia.

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *ASPLOS*. Atlanta, GA, USA, 175–188.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Cătălin Hriţcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Verified Low-Level Programming Embedded in F*. arXiv:1703.00053. (Feb 2017). http://arxiv.org/abs/1703.00053

[4] Edwin C. Brady. 2011. IDRIS — Systems Programming Meets Full Dependent Types. In *2011 PLPV*. New York, NY, USA, 43–54. http://doi.acm.org/10.1145/1929529.1929536

[5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *5th ICFP*. New York, NY, USA, 268–279. http://doi.acm.org/10.1145/351240.351266

[6] Willem-Paul de Roever and Kai Engelhardt. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47. United Kingdom.

[7] Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. New York, NY, USA, 455–468. http://doi.acm.org/10.1145/2500365.2500574

[8] Peter Lammich. 2013. Automatic Data Refinement. In *4th ITP*. LNCS, Vol. 7998. 84–99.

[9] S. Lindley and C. McBride. 2013. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *2013 ACM SIGPLAN Symp. Haskell*. New York, NY, USA, 81–92. http://doi.acm.org/10.1145/2503778.2503786

[10] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement Through Restraint: Bringing Down the Cost of Verification. In *ICFP*. Nara, Japan.

[11] The COGENT Team. 2017. COGENT Homepage. (2017). http://ts.data61.csiro.au/projects/TS/cogent.pml.

[12] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. In *2014 ACM SIGPLAN Symp. Haskell*. New York, NY, USA, 39–51. http://doi.acm.org/10.1145/2633357.2633366

[13] Philip Wadler. 1990. Linear types can change the world!. In *Programming Concepts and Methods*.