Towards A Practical High-Assurance Systems Programming Language

Zilin Chen

A thesis in fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computer Science and Engineering
Faculty of Engineering
The University of New South Wales

September 2022

Acknowledgements

First and foremost, I would like to thank my primary technical supervisor Gabriele Keller for her kind support and guidance, not only during the course of my PhD program, but also throughout my entire academic career. Her supervision continues after she left UNSW.

Christine Rizkallah, to whom I owe many thanks, took over the role of the technical supervisor when Gabi moved to the Netherlands. She has dedicated a lot of her time and consideration to me and to my PhD project. In particular, she keeps the same level of commitment after she left UNSW. I might have forced her to read many papers on topics that she is not necessarily very interested in. Her mentorship, both on my research and on my general well-being, has been invaluable.

I would also like to thank Gernot Heiser, who kindly agreed to be my primary supervisor. Being my supervisor, and dealing with the formality from the university, simply gives him one more thing to worry about beyond his already overly full schedule.

Besides the lovely supervisory panel, I am very grateful to Gabi, Manuel Chakravarty and Liam O'Connor, who offered the functional programming / type theory courses (COMP3141, COMP3161, COMP4181) at UNSW around 2012–2014. Without them and their courses, I would definitely have had a totally different career. They, together with the rest of the old Programming Languages and Systems group at CSE UNSW, had been of great help in my early career, before and during the first half of my PhD, until the group was disassembled. Liam, in particular, is impressive in his technical insights and communication skills, from which I have benefited enormously, since my first undergraduate course on type theories.

The Trustworthy Systems (TS) group, in which I have been a member of for a whole decade, is my second home. Thanks are due to everyone in the group. The brilliant TS leaders, past and current, Gernot, Gerwin Klein, June Andronick and Michael Norrish, have been very considerate and supportive in helping me with the balance between my work and my study. I would also like to thank the people who have worked on the Cogent project, in addition to those I have mentioned above: Sidney Amani, Toby Murray, Yutaka Nagashima, Japheth Lim, Alex Hixon, Partha Susarla, Vincent Jackson, Ambroise Lafont, Craig McLaughlin. The list goes on, and I apologise for not being able to enumerate everyone.

The PhD students, colleagues and friends who forked out from TS, and are now at the University of Melbourne under the supervision of Christine, have been great company and provided invaluable support during the last year of my PhD program. They are Vincent, Louis Cheung, Zhuo (Zoey) Chen and Selene Linares. Thanks are due to them for their kind help, and I truly enjoy the many discussions that we had, both technical and non-technical. They generously offered their time to proofread my thesis drafts. Any mistakes in the thesis, though, are absolutely my fault.

I also want to offer my thanks to the students that I co-supervised. They helped with the design and implementation of various parts of the Cogent framework, and experimented with the ideas that I had but could not find time to implement by myself.

Thanks are also due to my thesis examiners, Prof. Anil Madhavapeddy and Dr. Jonathan Protzenko, who kindly agreed to examine my thesis and provided a lot of constructive feedback on my work.

Outside the academic community, my friends, in particular those who live in Sydney and I can hang out with at times, have been an important part of my life. They are the ones who keep me sane and well at work, and they are always willing to lend me a helping hand when I need.

Last but not least, I sincerely thank my parents and the whole family living in the other hemisphere, for their remote but full-hearted love and support. Special thanks to my mother, who solicitously, repeatedly and unstoppingly wanted to know my timeline to the completion of my PhD. I owe my family a lot, especially for the past ten years, during which time I only visited them very few times and spent little time with them, even during difficult times.

Publications

- 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. Apr. 2016. "Cogent: Verifying High-Assurance File System Implementations." In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188. DOI: 10.1145/2872362.2872404
- Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. Aug. 2016. "A Framework for the Automatic Formal Verification of Refinement from Cogent to C." in: *International Conference on Interactive Theorem Proving*. Nancy, France
- Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Sept. 2016. "Refinement Through Restraint: Bringing Down the Cost of Verification." In: *International Conference on Functional Programming*. Nara, Japan
- Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. Oct. 28, 2017. "The Cogent Case for Property-Based Testing." In: *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Shanghai, China, 1–7. ISBN: 9781450351539. DOI: https://doi.org/10.1145/3144555.3144556
- Zilin Chen. 2017. "COGENT$^{\Uparrow}$: Giving Systems Engineers A Stepping Stone (Extended abstract)." In: *The workshop on Type-Driven Development* (TyDe'17). Oxford, UK. Retrieved April 2019 from https://www.cse.unsw.edu.au/~zilinc/tyde17.pdf
- Liam O'Connor, Zilin Chen, Partha Susarla Ajay, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Nov. 2018. "Bringing Effortless Refinement of Data Layouts to Cogent." In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, Limassol, Cyprus, 134–149. DOI: https://doi.org/10.1007/978-3-030-03418-4_9
- Zilin Chen, Matt Di Meglio, Liam O'Connor, Partha Susarla Ajay, Christine Rizkallah, and Gabriele Keller. Jan. 2019. *A Data Layout Description Language for Cogent*. at PriSC. Lisbon, Portugal
- Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. "Cogent: uniqueness types and certifying compilation." *Journal of Functional Programming*, 31. DOI: 10.1017/S095679682100023X
- Zilin Chen. Sept. 2022. "A Hoare Logic Style Refinement Types Formalisation." In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (TyDe '22). ACM, Ljubljana, Slovenia, 14 pages. DOI: 10.1145/3546196.3550162
- Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. Dec. 2022a. "Property-Based Testing: Climbing the Stairway to Verification." In: *ACM SIGPLAN International Conference on Software Language Engineering* (SLE

2022). ACM, Auckland, New Zealand, 14 pages. DOI: 10.1145/3567512.3567520. **Distinguished Artifact Award.**

- Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Jan. 2023. "DARGENT: A Silver Bullet for Verified Data Layout Refinement." *Proc. ACM Program. Lang.*, 7, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Article 47, 27 pages. DOI: 10.1145/3571240

Abstract

Writing correct and performant low-level systems code is a notoriously demanding job, even for experienced developers. To make the matter worse, formally reasoning about their correctness properties introduces yet another level of complexity to the task. It requires considerable expertise in both systems programming and formal verification. The development can be extremely costly due to the sheer complexity of the systems and the nuances in them, if not assisted with appropriate tools that provide abstraction and automation.

Cogent is designed to alleviate the burden on developers when writing and verifying systems code. It is a high-level functional language with a certifying compiler, which automatically proves the correctness of the compiled code and also provides a purely functional abstraction of the low-level program to the developer. Equational reasoning techniques can then be used to prove functional correctness properties of the program on top of this abstract semantics, which is notably less laborious than directly verifying the C code.

To make Cogent a more approachable and effective tool for developing real-world systems, we further strengthen the framework by extending the core language and its ecosystem. Specifically, we enrich the language to allow users to control the memory representation of algebraic data types, while retaining the automatic proof with a data layout refinement calculus. We repurpose existing tools in a novel way and develop an intuitive foreign function interface, which provides users a seamless experience when using Cogent in conjunction with native C. We augment the Cogent ecosystem with a property-based testing framework, which helps developers better understand the impact formal verification has on their programs and enables a progressive approach to producing high-assurance systems. Finally we explore refinement type systems, which we plan to incorporate into Cogent for more expressiveness and better integration of systems programmers with the verification process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are an important part of the modern society, and are tightly connected to everyone's life. Their presence is everywhere, ranging from small, simple devices such as light control panels in homes and offices, to much more sophisticated systems such as rockets and spacecraft. Ideally, these software systems should operate correctly, in accordance with their designs and specifications. However, defects in software inevitably appear during development. For the majority of software systems, it is sufficient to apply traditional quality control measures, such as testing. These measures are capable of detecting some defects, but usually not adequate to guarantee their absence, especially not in a mathematically sound manner. Depending on the application, the remaining defects in software products have a varying level of impact on people's lives. If a light control panel manifests a integer overflow problem every several years and turns off a light by mistake, it generally does not result in a major disruption. In contrast, a similar bug in an airliner can be fatal, and unfortunately such bugs have indeed been reported. For example, Boeing's flagship 787 Dreamliner aircraft could lose all alternating current electrical power if it had been continuously powered for 248 days [Federal Aviation Administration 2018]. This was cause by an internal counter overflow in the generator control units' software. The overflow bug, in this instance, could potentially lead to fatal accidents. To eliminate errors, more rigorous measures must be taken to ensure the functional correctness of software systems in safety-critical applications. Such applications include, but are not limited to, vehicles and facilities in a transport system, power plants controls, medical devices, etc. Having correctness guarantees for these systems is important, not only because the misbehaviour in these systems can cause loss of life, but also because it is crucial to retain the confidence the public has in these systems, and furthermore in the industries where these systems are used.

1.1 Formal Software Verification

Formal verification of software systems [Almeida et al. 2011; Bjørner and Havelund 2014] is a methodology for reasoning about properties of computer programs by presenting mathematical proofs based on a well-founded formal system. Formal verification is a composite of its theory, engineering practice and toolset. It is accomplished by viewing or modelling the program as a mathematical object, and formally proving properties directly of the mathematical model of the system, or showing that the model of the implementation is compliant with a formal mathematical specification.

Interactive theorem proving [Ouimet and Lundqvist 2007; Ringer et al. 2019], among all formal verification methods, is arguably the most thorough and rigorous means to ensure the correctness of programs. It is typically done in a *proof assistant* (or sometimes interchangeably called a *theorem prover*) software, in which the user can write program specifications, define functions and theorems and construct mathematically proofs in a formal language. The formal language is defined on top of a sound formal system, such as higher-order logic (HOL) or Martin-Löf's intuitionistic type theory [Martin-Löf 1972, 1984]. The proofs are mechanically checked by the proof assistant according to the formal rules in its underlying formal system. To construct the proofs, proof assistants provide the user with varying degrees of automation, but they rely on user input to make progress when the automation falls short—after all, there is usually no decision procedure for the validity of theorems in these formal systems. Commonly used proof assistants include Isabelle/HOL [Nipkow, Paulson, et al. 2002], Coq [Bertot and Castéran 2004], Agda [Norell 2009, 2007] and HOL [M. J. C. Gordon and Melham 1993], just to name a few.

Formally proving the correctness of software system gives very strong guarantees, but it comes at a price: enormous amount of work in proof engineering [Andronick et al. 2012; Matichuk et al. 2015]. Formal verification technologies typically require very specialised expertise from the developers [Bowen and Stavridou 1993], which is a scarce and expensive resource. Interactive theorem proving is even more so, and mandates heavy user engagement in the course of software development.

Large systems are often built in a hierarchical manner. The low-level infrastructure at the bottom layer is typically smaller, more stable, and is shared across multiple high-level modules, which are more feature-rich and fast-evolving. The bottom layers are usually a good place to effectively spend the scarce resources for formal verification, and their verification can have a far-reaching effect. If any defects in them are exploited, it compromises the correctness of all the dependent modules and the overall system. Verified low-level systems grant a solid foundation for any subsequent formal reasoning about the correctness of other higher-level components in the system.

In a software system, the lowest level of the software stack is usually the operating system (OS). At the core of an OS sits the kernel, which is the most critical part of the whole OS. With a microkernel architecture, which is commonly advocated for its better component isolation and security characteristics [Biggs et al. 2018], the layered structure of the whole system is more obvious: The kernel is the bottom layer, running the most essential mechanisms of the OS; other system services are stacked on top of the kernel, running in user space. The OS kernel and services must be dependable, so that other applications can run on top of the OS safely and securely. Low-level systems are intrinsically complicated to develop, and are also difficult to get right. There are usually more nuances in their designs and implementation strategies due to the special role that they play, which entails their unique performance characteristics and the interactions with hardware and other systems. The main source of conceptual complexity in systems code is the need to deal with real hardware details, as opposed to some abstract representations. Microkernels are good examples of typical low-level code. Albeit their relatively small codebase, they are immensely complicated to build, as almost everything in them is intertwined in exchange for minimalism and high performance [Heiser and Elphinstone 2016]. The nature of systems code gives us a dichotomy: On one hand, because they are tricky to program and hence more prone to bugs, they are in desperate need of verification; on the other hand, the low-level nature makes them more challenging to model and to verify. Luckily, the research community has recognised the demand for verified systems programs and has attended to solving this problem. In the past decades, fruitful research results are seen in formal software verification, especially of systems software, from OS kernels [Gu et al. 2016; Klein, Elphinstone, et al. 2009; Nelson et al. 2017], to OS services such as device drivers [Alkassar and Hillebrand 2008; Amani, Chubb, et al. 2012; Hao Chen et al. 2016; Duan and Regehr 2010], file systems [Amani 2016; Haogang Chen et al. 2015], network protocols [Cluzel et al. 2021], to compilers [Bourke et al. 2017; Kumar et al. 2014; Leroy 2009], to utility software [Chlipala 2015; Protzenko, Parno, et al. 2020; Zaostrovnykh et al. 2017].

1.2 Program Verification Approaches

Researchers and practitioners from the industry have been constantly seeking for opportunities to develop better tools to reduce the effort needed when conducting formal verification. Programming language research provides an answer. There have been numerous successful attempts to use high-level programming languages equipped with powerful type systems to produce low-level code with program synthesis techniques. The core idea is to allow the developers to operate on a more abstract ground, which is easier to reason about, and the proved properties can be transported to the low-level implementation easily. These high-level languages should be powerful enough for the users to perform low-level tasks efficiently without needing them to attend to every aspect of the low-level details, which is a major source of software bugs. For languages aimed at low-level programming, they ideally should allow the users to fine-tune some aspects of the implementation that they are mostly concerned with, such as memory management and data layout.

Another key factor to successful adoption of formal verification is its scalability, especially when we move the practice of formal verification up the hierarchy of software modules. This is because, as we move further towards the application layer, the number of systems and the amount of code becomes more abundant. A microkernel of an OS can be as small as a few thousand lines of C code. Other components of an OS, such as file systems and device driver, can be much larger, and an operating system may require multiple of these components. For instance, modern Linux supports several dozens file systems, and the number of device drivers it supports is hardly countable. If the effort required to manually verify a small OS kernel is barely affordable, it will certainly not be the case for the above-mentioned OS components. It is reported that it cost 11 person-years to develop and verify the seL4 microkernel, which comprised of less than 9 thousand lines of C code. This number does not include the effort spent on the proof infrastructures, tools and libraries [Klein, Elphinstone, et al. 2009].

The research into high-level programming languages for low-level systems programming has seen some fruitful results. For example, F* [F* 2022; Swamy et al. 2016] is a general-purpose functional language equipped with a very powerful dependent type system with effects. F* is designed for program verification, and it leverages the power of Satisfiability Modulo Theories (SMT) solvers, symbolic computation and proof tactics to increase the level of automation when constructing proofs. F* is not only a single programming language, but it is also a framework for embedding domain-specific languages (DSLs). For instance, Low* [Protzenko, Zinzindohoué, et al. 2017] is an embedded domain-specific language (EDSL) tailored for low-level systems programming. The language features fine-grained control over memory management, which is of paramount importance for low-level programming tasks. A custom compiler backend is attached to Low* to generate C code. EverParse [Ramananandro et al. 2019] is another embedded language that specialises in parsing binary formats. Domain-specific knowledge and proof techniques are leveraged to further ease the verification process. F* programs can be extracted to a range of target languages, including OCaml, C and assembly. It enables the reasoning about functional correctness and security properties of realistic applications.

CakeML [CakeML n.d.; Kumar et al. 2014] is an ML-style functional programming language that is mechanically verified end-to-end. Programmers can choose to express the intended behaviours of their programs in higher-order logic (HOL), and a language synthesis tool will generate CakeML syntax tree accordingly. It can be subsequently compiled to binary code on several supported architectures. Although low-level programming is not necessarily the best application domain that CakeML is designed for, some ongoing research is exploring the possibility of reusing CakeML, or parts of its verified compiler pipeline for low-level programming challenges. Nonetheless, CakeML is another very good example of how high-level programming languages can be used to facilitate the reasoning of programs on an abstract level.

ATS [Danish and Xi 2014; Xi 2017] is a multi-paradigm, feature-rich, and efficient programming language that unifies implementations with formal specifications by virtue of its expressive type system. It is equipped with a dependent type system and also a linear type system, making it very suitable for low-level programming, where the performance, in terms of both time and space, is key. The employment of the linear type system also enables multithreaded programming with ATS in a safe manner.

Fiat [Delaware, Pit-Claudel, et al. 2015] is a library for the Coq proof assistant [Bertot and

Castéran 2004; Coq n.d.], allowing developers to declaratively define the specification of SQL-style query structures. It supports iterative refinement of the specification to efficient functional programs, with high degree of automation. Each refinement step produces a certificate that can be mechanically checked by Coq. Fiat specifically has a focus on synthesising efficient abstract data types from the specification.

The list of examples of how programming languages can be leveraged to aid formal verification does not end, and we do not intend to, and cannot, exhaust it. What we have seen, however, is that the community acknowledges the potential that programming languages have in contributing to successful adoption of formal verification in programming practices—even though the detailed strategies in the implementation vary—and the benefits have been indeed convincingly demonstrated.

1.3 The Cogent Framework

Our answer to the formal verification challenge in low-level systems programming is Cogent [Klein, Andronick, Keller, et al. 2017] (Chapter 2). Cogent is a high-level functional language for systems programming, and is a tool for reducing the cost of formal verification. This is achieved by means of code-proof co-generation—the Cogent's certifying compiler generates C code along with a proof that certifies the correctness of the generated code. By design, Cogent is not a general-purpose language for all types of systems code. By virtue of the type system that Cogent uses, Cogent is specifically purposed for systems programs that do not heavily rely on memory aliasing (i.e. sharing) or can be implemented in an alternative design with minimal aliasing without degrading their performance significantly [Amani 2016]. For this reason, Cogent has been primarily used for developing file systems and device drivers. Microkernels, on the other hand, are typically not suitable candidates, because aliasing is so key to their performance and they can hardly be modularised.

The design of Cogent is heavily influenced by the experience that the team has previously acquired in formally verifying the seL4 microkernel [Klein, Andronick, Keller, et al. 2017; Klein, Derrin, et al. 2009]. The verification of seL4 is achieved by the joint effort from systems engineers and proof engineers, two groups of developers with very distinct technical background. The design of seL4 was shaped iteratively around a Haskell prototype, which served as a communication protocol between the two groups of developers. This was also greatly assisted by developers who were knowledgable in both fields. In the seL4 development, the Haskell prototype and the C implementation of the kernel were not formally, nor mechanically connected. When systems developers attempted to optimise the C implementation or to add new features, it often resulted in the Haskell prototype becoming out-of-sync with the actual C implementation, and some effort would be required to update the prototype to catch up.

Cogent, in its ecosystem, serves a similar purpose to the Haskell prototype as seen in the seL4 development. But in this instance, the semantics of the language is guaranteed to be an abstraction of the C code, thanks to Cogent's certifying compiler. We use Cogent to serve as a middle ground for both systems programmers and proof engineers. Admittedly, the connection between the Cogent program and the high-level functional correctness proof needs to be maintained manually. This typically results in a workflow where developers first design and develop the system in Cogent, and once they are confident with the features and performance of the program, they hand the program to the proof engineers for verification.

1.4 Evaluation

To assess any verification-oriented programming languages, it is important that we put them in the right context. It is important that such a language can deliver the expected results in the formal verification of software systems. It should not be neglected though, when putting these languages in action, they necessarily involve human factors, which include effectiveness, efficiency, workflow, maintainability, and general user experience. We summarise them as the *usability* problem of a language-aided verification framework. Specifically, we further dissect the usability problem into the following aspects, which will be the benchmarks we use to assess Cogent.

Expressiveness—whether the language is powerful enough for the users to fulfil their tasks. Cogent partly serves as a specification language for low-level system code. When the program synthesiser cannot effectively derive code, we need to enrich the specification so that the compiler has enough knowledge about how to synthesise code.

Performance—how does the resulting program perform when it is deployed. When programming in a high-level language, there are inevitably gaps in the performance of the generated low-level target code when compared with code manually written in a low-level language. To fill the gap, the users should be able to more directly intervene in the code generation algorithm for the performance critical portions of the program, if the compiler itself is unable to determine the optimal compilation strategy. This is similar to the plethora of optimisation flags found in many compilers.

Framework—does the language give enough guidance to the developers so that they tend to make the right design decisions. A framework should suggest an appropriate workflow to the developers. When the users follow this workflow, they should be able to achieve their end goal with little back-tracking in their development, if any. The features that the framework provides can have a big impact on how users proceed with their development. When the framework falls short on certain features, the users will inevitably try to navigate through the obstacles to get their work done, veering off course. Having the right framework ensures that the users use the framework as intended, and obtain the most benefit out of the framework. Also, the framework should set the stage for developers to communicate effectively, both among themselves and with the language.

Tooling—does the language provide the necessary tools to facilitate an effortless development experience. The programming language and the ecosystem should provide users with user-friendly tools. These tools should be intuitive to understand and learn, simple to use, and effective in what they are intended for. When the tools are inadequately effective or efficient, the users may try to exploit shortcuts to get things done in an inappropriate manner, or worse, be deterred from using the language.

In the case of Cogent, the high-level question that we would like to ask then is: what is missing from the Cogent language and its verification framework for it to be effectively used in real-world applications. By experimenting Cogent in several case studies, we acquired more insights about the Cogent ecosystem, and gained experience with its application in realistic development scenarios. We identified a few shortcomings of the prior Cogent language and its verification framework:

- The Cogent language does not grant users fine-grained control over the low-level memory layout of data types, and the default layout chosen by the compiler is not in general suitable for implementing binary-compatible, efficient systems programs. Users often have to adapt the design of their programs to match how the compiler chooses to represent data, resulting in inadvertent deviation from the reference design of the system. Also, the default layout that the compiler picks creates a representational gap between the Cogent portion of the system and the ambient program with which the Cogent portion interfaces. This leads to costly data conversion and copying, hindering the overall performance and trustworthiness of the system. We address this problem in Chapter 3.

- The Cogent language is restricted, in the sense that it doesn't natively support language features that are widely used in realistic programming tasks, such as recursion, loops, and aliasing. Therefore, programming with Cogent heavily relies on its interface to C code, when extra expressiveness is sought after. It is crucial that the language interface between Cogent and C is simple, and is intuitive to use, so that it is not a technical obstacle in Cogent's ecosystem and it does not ruin the user experience with Cogent. This issue is examined and tackled in Chapter 4.

- Cogent greatly reduces the effort required in formal verification. It automates the refinement proof of the compilation process from Cogent to C code. However, it leaves parts of the overall system to be manually verified. They include the high-level functional correctness of the Cogent code, and that of the manually written C code that is needed to complement the expressiveness of Cogent, as we just mentioned above. The verification of the correctness of the Cogent program with respect to the specification is an post-hoc effort. However, Cogent itself in many cases fails to give developers any indication on whether the functional correctness proof can indeed be constructed in a satisfactory manner, if at all. Providing developers with early feedback on the verifiability of systems under construction is key to a successful deployment of the Cogent framework. We present our attempt at addressing this problem in Chapter 5.

- As a communication medium between systems developers and verification experts, Cogent faithfully expresses the operational behaviours of a system. But the language alone is not expressive enough for developers to write down the properties about their programs, which is another useful method to document their design and implementation, and to communicate with the verification team. Some other verification-oriented programming languages provide more expressive type systems, such as dependent types, and other languages have support for program annotations, in the form of assertions about language objects and program states. Examples of such languages include Dafny [Leino 2010], ESC/Haskell [Xu et al. 2009]. Chapter 6 is an exploration in this direction.

1.5 Contributions and Roadmap

This thesis is devoted to answering these questions and to addressing the limitations of Cogent. It utilises a breadth of technologies to augment the Cogent framework from different facets. Each addition to the overall system, moreover, carries ideas that are applicable independent of Cogent. They can be used on their own in other contexts individually. The rest of this thesis is structured as follows:

- In Chapter 2, we set up the background knowledge about the Cogent framework. After a quick tutorial on Cogent, we introduce the Cogent language, its compilation process, and the formal verification framework. We defer background information that is only required by an individual chapter to that relevant chapter.

- Chapter 3 is devoted to the Dargent extension of Cogent. Dargent is a data layout description language, and is an extension to Cogent's refinement-based formal verification framework. It allows programmers to dictate the memory layouts of high-level algebraic datatypes, giving systems programmers more expressiveness and flexibility in using a high-level language for low-level tasks. Despite a strong resemblance to data description languages (or sometimes called data marshallers, encoders or parsers), Dargent is in fact fundamentally different from these languages, as we will see throughout the chapter. The Dargent extension is key to enabling Cogent programmers write high-performance, binary-compliant systems code, without sacrificing the abstraction Cogent provides. Dargent demonstrates how a layout-refinement calculus can be incorporated in a high-level languages. We believe it is an essential feature that not only COGENT, but other real-world systems programming languages should be equipped with.

- COGENT programming is highly reliant on the interoperation with C code. Chapter 4 presents antiquoted-C, our foreign function interface (FFI) solution to alleviate the pain in language interoperability. Specifically, because of some specific features of COGENT's type system, the traditional name-mangling techniques in FFIs fall apart. The antiquoted-C takes a totally different approach to FFI design, which is intuitive for users, and is also cost-effective for language developers. We give our recipe for its construction and envision its application in other languages.

- One of the key reasons to adopt the COGENT approach is to benefit from its semi-automatic formal verification. Although a significant portion of the proof is fully automatic, manual effort is still needed in parts of the verification. This means that user expertise and development cycles are required to obtain a fully verified system. Chapter 5 presents the augmentation to the COGENT framework with a property-based testing (PBT) mechanism. This form of testing, apart from the typical functionality of finding bugs that all testing has, can also be used as an important intermediate step towards a fully verified system. Instead of testing properties about the concrete implementation of the system directly, we test the refinement relation between the implementation and an abstract executable specification. This specification is very similar to the one that will be used in the formal proofs. The testing framework thus is a lightweight replica of the corresponding refinement proof. It can guide developers to good systems designs that are amenable for formal verification. It also allows for an incremental approach to fully verified systems.

- Refinement types are another lightweight tool that can be used by systems programmers to express their intentions in the design of systems. It has some great potential when integrated into COGENT's verification framework. Chapter 6 digresses from COGENT, and explores some more theoretical results about refinement types. Specifically, we formalise a simple refinement type system in Agda, and interpret the language in a shallow manner in Agda. We formulate refinement types in terms of Hoare-triples, which significantly simplifies the formalisation. We present some initial results on using backward reasoning, the weakest-precondition predicate transformer as a typechecking algorithm, which aggregates all proof obligations about subtyping (i.e. logical entailments), which can be solved separately, either by automatic theorem provers, or manually by proof engineers.

- We conclude the thesis in Chapter 7 with more directions for future work outlined.

The source code associated with this thesis, and more broadly, with the whole COGENT project, is publicly available online in the project's GitHub repository [The COGENT team 2023]. A snapshot of the development presented in each chapter, if available, is given in the respective chapter.

Chapter 2

Cogent: A Programming Language for Systems Programming

This chapter draws on the work presented in the following publications:

- Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Sept. 2016. "Refinement Through Restraint: Bringing Down the Cost of Verification." In: *International Conference on Functional Programming*. Nara, Japan
- 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. Apr. 2016. "Cogent: Verifying High-Assurance File System Implementations." In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188. DOI: 10.1145/2872362.2872404
- Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. Aug. 2016. "A Framework for the Automatic Formal Verification of Refinement from Cogent to C." in: *International Conference on Interactive Theorem Proving*. Nancy, France
- Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. "Cogent: uniqueness types and certifying compilation." *Journal of Functional Programming*, 31. DOI: 10.1017/S0956796821000023X

The author of this thesis contributed to the design of the Cogent language, together with the first author of the ICFP'16 paper, who is the primary contributor. The author made significant contribution to the design and implementation of the Cogent compiler. Another major contribution the author made is to bring Cogent to the development of realistic systems programs, assisting the systems experts—the first and second authors of the ASPLOS'16 publication—with developing two file systems using the Cogent framework. The author also contributed to the formal verification aspect of the work, but played a less central role.

We devote this chapter to an introduction of Cogent, the main research platform on which this thesis is based. In this chapter, we first give an overview to the Cogent framework (Section 2.1). Then we introduce the Cogent language with a quick tutorial (Section 2.2). Next, we discuss Cogent's type system (Section 2.3), which is important for understanding the development presented in the rest of this thesis. Towards the end of this chapter, we outline Cogent's compilation pipeline (Section 2.4) and its verification pipeline (Section 2.5), which are particularly relevant to the work presented in Chapter 4 and Chapter 5 respectively.

2.1 Overview of Cogent

In an attempt to developing and formally verifying file systems [Amani 2016], researchers saw a need for a methodology which allows them to achieve it productively. Contrary to an operating system's microkernel, such as seL4 [Klein, Elphinstone, et al. 2009], file systems are not only possibly larger in size, but also, more importantly, dozens of them may be needed for one operating system to serve different users. That is why manually verifying one single file system is often not adequate, and manually verifying dozens of them is not practical.

Keller et al. [2013] then came to the conclusion that a high-level functional language was a good choice to fulfill the task of (partly) automating the verification of file systems code. In a nutshell, the systems programmers write file systems code in a functional language, which is capable of co-generating efficient C code, along with a proof that the generated code is correct with respect to the source language's semantics. Not only that the high-level functional language is easier to program with, but also, more crucially, it is a lot easier to reason about in an interactive theorem prover, such as Isabelle/HOL [Nipkow, Paulson, et al. 2002], since *equational reasoning* techniques can be utilised. This hides the low-level tedium such as reasoning about pointers and memory locations commonly seen in traditional C verification.

The high-level functional language that we arrived at was Cogent. It is a total, higher-order, polymorphic, purely functional programming language. The Cogent compiler co-generates low-level C code, as well as a machine-checked certificate proving the correctness of the compiled C code. Cogent differs from mainstream functional languages (e.g. Haskell, ML) in a few aspects.

First of all, since Cogent is a language tailored for writing and reasoning about low-level systems code (initially file systems code, and later extended to other domains of systems programming), and in such applications manual memory management is paramount for performance concerns, all heap memory management has to be done by the user explicitly, as in C. This saves Cogent from having a garbage collection mechanism, which simplifies the verification of the compilation, and also importantly, gives developers more control over the run-time performance of the compiled Cogent programs.

Secondly, Cogent is equipped with a *uniqueness type system* [Baker 1992; Barendsen and

Smetsers 1993; Bernardy, Boespflug, et al. 2017; de Vries, Plasmeijer, et al. 2007, 2008; Ennals et al. 2004; Harrington 2006; Hofmann 2000; Jung, Jourdan, et al. 2017; Li et al. 2022; Marshall et al. 2022; Odersky 1992; Wadler 1990; Walker 2005], which is key to enabling a seamless semantic transition from COGENT's purely functional semantics to C's imperative semantics. Along with the semantic shift, it also lends itself to a principled compilation strategy to efficient low-level C code, by means of performing in-place updates [Bernardy, Boespflug, et al. 2017, Section 6.3].

Thirdly, COGENT is a pure and restricted language. Unlike HASKELL, COGENT does not have primitives like the `IO` monad, which allows for interaction with the "world" while retaining the purity of the language. Any effectful computations in COGENT bottom out at foreign function calls into C. These impure operations include I/O, memory (de)allocation, etc. Additionally, for the simplicity of the automatic proof generation, COGENT does not natively support recursive data types, recursion or loops.

Instead, COGENT is bundled with a standard library of common abstract data types (ADTs), such as arrays, linked lists and red-black trees, and the corresponding operations on them. This library consists of many data types and functions that cannot be natively implemented in COGENT. Because the library is primarily implemented in C, we also sometimes refer to it as the *C library*, even it contains a small amount of COGENT code. Users can extend this C library with more data structures and functions that can be reused by other applications. For instance, during the experiments on implementing file systems using COGENT [Amani, Hixon, et al. 2016], the library was extended to include some key data structures from the virtual file systems switch (VFS) layer. Apart from the C library, users can always opt for implementing a small amount of their code directly in C, when COGENT is not expressive enough to complete a specific task or some manual optimisation needs to be done on the C level. COGENT programs can access this part of the code via a *foreign function interface* (FFI) between COGENT and C.

On the verification front, the purely functional semantics of COGENT and its *certifying compiler* reduce the effort needed to prove end-to-end correctness properties about the compiled programs. Specifically, what we want to establish is a notion of *refinement* [R. J. R. Back 1988; Morgan 1990; Roever and Engelhardt 1998] between the C implementation and a *functional correctness specification* of the system that the user wrote in Isabelle/HOL. The refinement relation states that the generated C code correctly implements of the specification (see Figure 2.1). The end-to-end proof is divided into two halves by a compiler generated functional semantics of the COGENT source program, which is shallowly embedded in Isabelle/HOL. The *C refinement proof* (the lower half) from COGENT's program semantics down to C is fully automatic, thanks to COGENT's certifying compiler. Instead of verifying the correctness of the compiler, we prove the correctness of each compiled C program by *translation validation* [Pnueli et al. 1998]. The *functional correctness proof* (i.e. the upper half of the end-to-end proof) from the semantics of the submitted COGENT source program to the functional correctness specification remains a manual effort, but it is eased through equational reasoning techniques by virtue of the purely functional

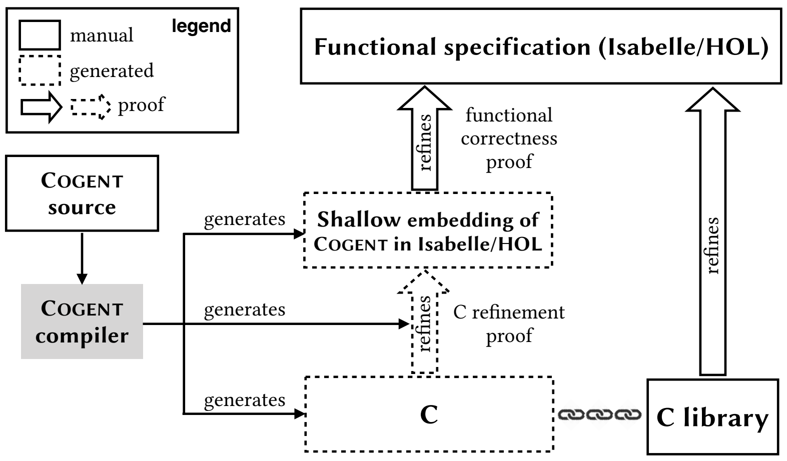


Figure 2.1: The verification framework of Cogent

nature of both the Cogent program and the specification. With the end-to-end proof, more abstract properties can be proved on top of the functional specification. Any property of the functional specification can be projected onto the C code. Cogent and its certifying compiler greatly reduce the effort required to prove the correctness of low-level code, as seen in projects like the seL4 verification [Klein, Elphinstone, et al. 2009; Klein, Sewell, et al. 2010], in which the manual verification is done directly on the C language level.

The correctness of the C library has to be manually verified, and this proof can indeed by tough. Thankfully, the C library is meant to be shared across many systems, by all Cogent users. Therefore, the effort invested in proving each library module is an one-off effort, and it will be amortised over time. The work by Cheung et al. [2022] showcases the verification of the `WordArray` module, which defines common operations on arrays of words.

Cogent has been used to implement two real-world file systems: BilbyFs [Amani 2016] and the Linux ext2 file system. As a proof of concept, the functional correctness of two key file system operations in BilbyFs—`sync`, which flushes the content in the buffer to the physical storage medium, and `iget`, which looks up an inode from the physical medium—were formally verified [Amani 2016; Amani, Hixon, et al. 2016]. Later, Cogent was also used to implement and verify device drivers, some of which will be mentioned in Chapter 3. From the case studies, it can be seen that Cogent's verification methodology indeed reduces the effort of formally verifying operating systems code, compared to verification that is directly carried out on C.

Interim summary The verified microkernel seL4 [Klein, Elphinstone, et al. 2009] sets a baseline and a guideline for low-level systems code verification with the Isabelle/HOL verification toolchain. It uses the three-layer architecture, Isabelle/HOL functional correctness specification – Haskell executable specification – C code, to decompose and modularise the verification task. Each step of the verification is a refinement proof established manually, and the two refinement proofs are composed to form the justification for the correctness of the concrete C implementation with respect to the abstract Isabelle/HOL specification. The Cogent project was an incarnation of an attempt to verify file systems at scale, and a language-based approach was chosen. Amani [2016]'s PhD dissertation gives a full account of the modular design of a verifiable flash file system BilbyFs, its two independent implementations in C and in Cogent, and the formal development of BilbyFs based on the Cogent verification framework. The thesis can be interpreted in two ways: It is an umbrella project that defined the scope of Cogent and used Cogent as a tool to approach file system verification; it is also the largest case study of the Cogent language, which provided the language designers with feedback from the user's perspective and validated the verification methodology of Cogent. Almost in parallel with Amani [2016]'s work, O'Connor [2019b]'s PhD dissertation documents the design and formal semantics of the core calculus of the Cogent language, with a focus on its uniqueness type system, its type inference engine and the semantic shift from a purely functional semantics to an imperative one. These two pieces of work, together, advanced seL4's verification story by using Cogent as the intermediate level between the Isabelle/HOL specification and C, and by generating C code and its refinement proof from the high-level language Cogent, which was a leap in developer productivity for both systems programmers and verification engineers. This thesis builds on previous work, enriching the Cogent language and its verification framework with highly demanded language features and better infrastructure for end users, while preserving the abstraction boundary provided by Cogent.

2.2 Cogent at a Glance

One of the most outstanding features of Cogent, apart from its certifying compiler, is its uniqueness type system [Baker 1992; Barendsen and Smetsers 1993; Bernardy, Boespflug, et al. 2017; de Vries, Plasmeijer, et al. 2007, 2008; Ennals et al. 2004; Harrington 2006; Hofmann 2000; Jung, Jourdan, et al. 2017; Li et al. 2022; Marshall et al. 2022; Odersky 1992; Wadler 1990; Walker 2005]. As an opening example, we showcase a toy file system implementation which gives a taste of Cogent and demonstrates the key language features.

Figure 2.2 contains the complete Cogent code for the toy file system's implementation.[1] The file system has a flat storage structure and does not support directories found in most real-world file systems. All it can store are regular files (analogous to Linux's regular file type). In the file system, files are identified by an integer identifier, and the contents of a file is represented as an array of bytes. The file system allows three operations on files: write, read and remove. The underlying data structure that stores the files is a binary search tree (BST), with the key being the file identifier. Alongside the storage, we keep a snapshot of the current state of the file system; we call it a *summary*. That summary consists of the number of files currently stored (similar to

[1]The real Cogent compiler does not accept Unicode characters as shown in the code. In this thesis we use Unicode symbols rather than their ASCII counterparts for better cosmetics.

```
include <gum/common.common.cogent>  -- for Maybe and Either
include <gum/common/wordarray.cogent>
-- length: ∀(a :< DSE). (WordArray a)! → U32

type BST k v
insert : ∀ (k :< DSE, v). (v, BST k v) → (Bool, BST k v)
lookup : ∀ (k :< DSE, v). (k, (BST k v)!) → Maybe v!
delete : ∀ (k :< DSE, v). (k, BST k v) → Either (BST k v) (BST k v, v)

type Summary = { count : U8, used : U32, last : Bool }
type Disk = BST U8 File
type File = { id : U8, data : WordArray U8 }
type Operation = < Write File | Read U8 | Remove U8 >

free : ∀ (t :< E). t → ()

go : (Operation, Summary, Disk) → (Summary, Disk)
go (op, sum, d) =
  op | Read id →
         let sum = (lookup[U8, File] (id, d)
                    | Just f  → sum {last = True}
                    | Nothing → sum {last = False}) !d
         in (sum, d)
     | Write f →
         let size = length (f.data) !f
         and (res, d) = insert (f, d)
         and True ⇐ res ▷ False → (sum {last = False}, d)
         and sum {count = c, used = u} = sum
         in (sum {count = c+1, used = u+size, last = True} , d)
     | Remove id →
         delete (id, d)
         | Right (d, f) →
             let sum {count, used} = sum
             and size = length (f.data) !f
             and _ = free f
             in (sum {count = count - 1, used = used - size, last = True}, d)
         | Left d → (sum {last = False}, d)
```

Figure 2.2: A simple example of a COGENT program

`ls -l | wc -l` in Linux), the total size of storage used in the file system (`du -s`), and a Boolean flag indicating whether the last file system operation has succeeded or not (Linux's return code).

On lines 1 and 2, we `include` some library files. The first one brings the types `Maybe t` and `Either a b` into scope. These two types have their standard meanings, as in many mainstream functional languages such as Haskell. In Cogent, they are defined as follows, unsurprisingly:

```
type Maybe t = < Nothing | Just t >
type Either a b = < Left a | Right b >
```

They are called *variant types* in Cogent, which are also commonly called *sum types*, *tagged unions* or *disjoint unions* in other contexts.

Line 2 includes a word array library, which implements an array type whose element must be words (unsigned words of 1, 2, 4, or 8 bytes long). This is the data type that we use to represent the file contents. The word array library is implemented in a variant of C, and it is invoked in Cogent via the C foreign function interface, which we shall discuss in more detail in Chapter 4. An excerpt of the word array implementation from Cogent's standard library will be given at the beginning of Section 4.6. For the time being, however, it is safe to think of it as a regular C implementation operating on the following `WordArray` type (with the elements instantiated to 8-bit word type `u8`):

```
struct WordArray_u8 {
        int len;
        u8* values;
};
```

The C struct consists of an integer value for the length of the array and a pointer to an ordinary C byte array to store the words. In most cases, the array is allocated dynamically in heap memory.

A word array `length` function (line 3 in Figure 2.2, commented out as it has been defined in the library) will be needed to calculate the size of the file. This function is polymorphic on the element type `a`. `a :< DSE` is a constraint on what instances `a` can take. `DSE` are the three permissions that a type can be granted. They stand for Droppable (or Discardable), Shareable and Escapable respectively. The type constraint `a :< DSE` grants `a` all three permissions, meaning that `a` can only be instantiated with types (e.g. words) that have all three permissions (i.e. unrestricted). When a type cannot be *shared* or *dropped*, we call it a *linear* type [Girard 1987; Wadler 1990]. In other words, any object of a linear type can only be used *exactly once*. When a type is not subject to this restriction, we call it a *non-linear* type. The Escapable permission controls a type's mutability, namely whether it is *writable* or *readonly*: *escapable* roughly corresponds to writeable. We will come back to the uniqueness type system with a more formal notion in Section 2.3.

On lines 5–8 we define an *abstract type* `BST k v`, where `k` is the BST's key type and `v` is the value type. Following the type are two *abstract functions*. An abstract function in Cogent is a function whose type signature is given in Cogent, while the definition of the function is provided in C. In the three interface functions, we again declare the key type to be unrestricted. These functions' signatures are standard. We need to note though, that all three functions can fail: The `insert` function may fail because the key to be inserted is already in the BST, and the other two functions may fail when the queried key does not exist. When the insertion and removal operations fail, the functions are still required to return the (unchanged) file store; this is due to the uniqueness requirements imposed by the type system. The uniqueness requirements are exempted in the `lookup` function. Because the function only needs to inspect the BST without modifying it, it only requires a readonly reference to the BST, denoted by the `!` symbol in the type `(BST k v)!`. A readonly type can be freely discarded (or dually, shared), and hence we do not have to return it in the `lookup` function. The restriction on readonly linear object is that there must not be any other writable references to the same memory region when any readonly references are present. This restriction is needed to retain referential transparency of the language.

The uniqueness type system governs how resources are used and how objects are referenced, which is of paramount importance when compiling the purely functional language into efficient C code. Even for abstract functions, we can still reason about the behaviours of the function according to its type. The `insert` function takes a linear value `v` and a linear BST, and returns as result a Boolean flag for signalling errors along with the potentially modified BST. Given the fact that the input `v` is not returned, we know that in the fail case where the new value is not added to the tree, the input object must be freed by the `insert` function. The `lookup` function returns a readonly copy of the object in question. From this information, it can be adduced that this function returns a readonly alias to the element that is sought after, as this is the only sensible definition of the function given the types. The `remove` function returns a linear object of type `v` —the removed element—along with the updated BST in the success case (viz. `Right` of the `Either` type). We know from the type system that the returned object is uniquely referenced, and hence we can later deallocate the memory for it.

We define some types for the file system on lines 10–13. The `Summary` and `File` types are *record* types, which have the normal meaning as in other languages. The `Disk` type is a *type synonym* to the abstract type `BST U8 File`. On line 15, we declare an abstract function `free` that works for all writable linear types, which is suggested by `t :< E`. Again, as Cogent is a restricted pure language, this function has to be defined in C. The remainder of the code defines the main file operation function `go`, which updates the file store and the summary as per the input operation.

The `go` function heavily relies on pattern matching, which is key to any real-world functional languages. Pattern matching in Cogent is represented by the scrutinee expression followed by a series of aligned vertical bars. Indentation is part of Cogent's syntax, similar to Haskell and Python. We pattern match on the variant `op`, which is the file system operation to be performed.

In the `Read` case, we invoke the `lookup` function, explicitly applying it to its type arguments `U8` and `File`. In most cases, types arguments can be omitted and the compiler is able to infer them.

Unfortunately, the type inference algorithm is incomplete and in some cases, a little explicit type information from the users is needed to guide the type inference engine. This is typically the case when a type variable only appears in its `!`-ed readonly form. As the `!` type operation is not injective, the users need to tell the compiler what instance the type variable should take. On lines 21 and 22, we set the `last` flag according to the result of the `lookup` function call. In the binding of the `sum`, we **let**! (reads "let bang") the variable `d` with the `!d` syntax. The variable `d` represents a writable linear object, and we use `!b` to temporarily turn it into a readonly reference, which is expected by the `lookup` function. The **let**! construct temporarily turns the linear object `d` into an unrestricted one, so that it can be discarded in the `lookup` function. This mechanism is simliar to immutable borrows in Rust [Klabnik and Nichols 2022, § 4.2]. The write permission on `d` is recovered outside of the **let**!-binding, as soon as the !-ed object goes out of scope. Cogent's linearity analysis is rather coarse-grained. To ensure the soundness of the type system, it always takes the safer course of action. It puts a bold requirement on the language that readonly objects must not escape from a **let**!-binding, even though it is safe in some cases. It therefore does not allow us to write the code on lines 20–23 in a seemingly equivalent way:

```
let res = lookup[U8, File] (id, d) !d
 in res | Just f  → (sum {last = True }, d)
        | Nothing → (sum {last = False}, d)
```

Here, `res` contains a readonly reference in the `Just` case and is thus disallowed.

In the `Write` case, we first need to compute the size of the file to be written, and then call the `insert` function. It is worth mentioning that we cannot swap the order of these two expressions: if we called `insert` first, then the linear file `f` would be *consumed* by the function, and we would then lose it permanently. This is due to the linear usage of linear objects—any (non-readonly) linear object can only be used *exactly* once. We give a name `d` to the function's return value (line 26). Since `d` is linear, the resultant `d` does not shadow the argument `d`. This is why we do not need to, and very often prefer not assign fresh variable names to linear objects that are updated in-place—keeping the same name is a good indication of how the variables got updated. Line 27 uses a biased pattern matching syntax, inspired by Idris [Brady 2011, 2013], to avoid excessive cascading indentation levels. The syntax translates easily to the equivalent and more traditional syntax in the following way:

$$\begin{array}{l} \textbf{let}\ p_2 \Leftarrow e \rhd p_1 \to e_1 \\ \textbf{in}\ e_2 \end{array} \quad \text{translates to} \quad \begin{array}{l} e \mid p_1 \to e_1 \\ \ \ \mid p_2 \to e_2 \end{array}$$

It greatly improves the aesthetic of the code when e_1 is short, which is commonly the case for error handling. On lines 28–29, we first "take" out the `count` and `used` fields from the file store and bind their values to `c` and `u` respectively, and then "put" the updated values back into the store. This is how we destructively update fields of a record, such as `Disk`.

The `Remove` case is largely similar to `Write`. We use regular pattern matching syntax for comparison. Line 33 shows the use of field-puns: when the binder's name is the same as the field name, we can omit the = sign and the binder on the right hand side. The same applies for a put operation $r\{\mathsf{f} = e\}$: when the expression e is a single variable f, whose name is the same as the field name f, the put can be shortened to $r\{\mathsf{f}\}$.

2.3 Cogent's Uniqueness Type System

Similar to the Rust language [Rust n.d.], Cogent is equipped with a uniqueness type system that ensures memory safety and eases verification. Cogent's type system allows imperative-style destructive updates, while retaining a purely functional semantics. The type system eliminates the need for a garbage collector, enabling the compiler to generate more efficient, predictable C code, making the language suitable for systems programming tasks.

The uniqueness types come at a cost: it is impossible in pure Cogent to implement data structures which, even temporarily, rely on sharing. Instead, such data structures have to be implemented in C, verified separately, and imported as ADTs through a FFI that requires the uniqueness type constraints to be satisfied at the interface level.

Uniqueness type systems ensure that each *linear* object in memory is uniquely referenced. Consequently, updates to these objects in a purely functional language can be compiled as in-place destructive updates, without the need for copying. In Cogent, we call the type of objects that are subject to the uniqueness restrictions *linear types*, and the rest *non-linear types*.[2] Roughly speaking, linear objects either reside in the heap, or contain pointers to other heap-objects. Cogent's verification framework depends on the AutoCorres library [Greenaway 2015; Greenaway et al. 2014], which does not support stack pointers, therefore all pointers address heap memory. In short: a linear object is behind a pointer and/or contains pointers.

Cogent *primitive* types include the unsigned n-bit integer types, `U8`, `U16`, `U32` and `U64`, and booleans (`Bool`). For the work that we will describe in Chapter 3, we extended the language with custom-sized unsigned integers. These integer types range from `U1` to `U63`. We call these non-standard integer types *non-word-size integers*, contrary to the *word-size integers* of `U8`, `U16`, `U32` and `U64`. The Cogent language defines primitive arithmetic, logical and bitwise operators only on word-size integers. The only operations allowed on non-word-size integers are **cast** and **truncate**, which converts an integer to a wider or a narrower integer. Primitive types in Cogent are all unboxed and non-linear.

Algebraic data types include record and variant types. *Type synonyms* to other types can be defined but such types have a *structural* interpretation, meaning the synonym is identical to the type it is aliasing. In general, two types in Cogent are identical if they have the same

[2]The terminology is somewhat intermingled unfortunately, partly due to historical reasons. Strictly speaking, we should call them unique types and non-unique types, which align with our (correct) use of the term uniqueness type system. The concepts of uniqueness and linearity are closely related yet different; their relationship has not been systematically studied until recently [Marshall et al. 2022]. For the purpose of this thesis, the distinction between them is largely an orthogonal problem. We recommend that interested readers consult Marshall et al. [2022]'s work for more details and references on that topic.

structure, after fully expanding any type synonyms. Furthermore, Cogent supports declaring abstract types with their definitions provided in C. For the development of Dargent (Chapter 3), we also extended Cogent with array types. Array types are indexed by their lengths, which are restricted to constant integers, for a tractable type checking algorithm.

The Cogent type system distinguishes between *boxed* and *unboxed* types through a *sigil* annotation on the type. In the surface language, a prefix # type-operator turns a type into its unboxed form. A boxed type (sigil ⓑ) resides in the heap and is accessed by reference through its unique pointer. An unboxed type (sigil ⓤ) is accessed by-value and either resides in the stack or is inlined inside a larger data structure in the heap. In the latter case, when the object is accessed, it will be copied by value to the stack. All boxed types are by definition linear while the converse is not true.

Records, arrays and abstract types can be either boxed or unboxed. Primitive types and variant types, however, can only be unboxed. For this reason, primitives and variants are not accompanied by a sigil. For a boxed type, Cogent allows further fine-grained control of accessibility: a boxed sigil can either be a writable boxed sigil (ⓦ) or a readonly one (ⓡ). When a type has a readonly sigil, even though it is behind a pointer or contains pointers, it becomes non-linear.

In Figure 2.3, we show some key typing rules for records and variants, and use them to explain the operations allowed on these types. For a fuller formalisation of the Cogent language, readers are recommended to consult O'Connor [2019b]'s PhD dissertation. In the typing rules, Δ is a kinding context, and Γ is a typing context. The judgement $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ is for splitting the typing context. This is needed because Cogent's type system is substructural, and linear objects cannot be freely duplicated or dropped in the context.

For record types, as Cogent does not support native heap-memory (de)allocation, only unboxed records can be constructed with primitive operations. In the Struct rule, the expression initialises several fields of an unboxed record with their initial values. In a record type, each field is annotated with its usage $\mathtt{u}$, which can be either *present* (°) or *taken* (•). When an unboxed record is constructed, all the initialised fields are present. The **take** operation is accomplished with a pattern matching in the surface language, such as `r {f1 = e1, f2 = e2}`. Multiple fields can be taken simultaneously in the surface language, but in the core calculus, a **take** operator only operates on one field at a time. It is used to access fields in a non-readonly record. Because of the uniqueness type system, when such a field is accessed, it transfers the ownership of that field from the record to a new binder. In the typing rule Take, after the field f is taken, its usage changes from present to taken. The new binder y now owns the field. Dually, a **put** operation returns the ownership of a field back to a record. Its surface syntax looks the same as that of a **take**, but it is an expression rather than a pattern. When a record is non-linear, fields can be directly accessed via the more conventional member access operation (written like `e.f` in the surface language), whose typing is formalised in the Member rule. The non-linearity of the

$$\boxed{\Delta;\Gamma \vdash e : \tau}$$

$$\frac{\Delta;\Gamma \vdash \overline{e_i : \tau_i}}{\Delta;\Gamma \vdash \#\{\overline{\mathrm{f_i} = e_i}\} : \{\overline{\mathrm{f}_i^{\circ} : \tau_i}\}\ \textcircled{u}}\textsc{Struct} \qquad \frac{\begin{array}{c}\Delta \vdash \{\overline{\mathrm{f}_i^{\circ} : \tau_i}, \mathrm{f}^{\bullet} : \tau\}\ s\ \textbf{Share} \\ \Delta;\Gamma \vdash e : \{\overline{\mathrm{f}_i^{\circ} : \tau_i}, \mathrm{f}^{\bullet} : \tau\}\ s\end{array}}{\Delta;\Gamma \vdash e.\mathrm{f} : \tau}\textsc{Member}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : \{\overline{\mathrm{f}_i^{\mathrm{u}} : \tau_i}, \mathrm{f}^{\circ} : \rho\}\ s \qquad s \neq \textcircled{r} \\ \Delta; x : \{\overline{\mathrm{f}_i^{\mathrm{u}} : \tau_i}, \mathrm{f}^{\bullet} : \rho\}\ s, y : \rho, \Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \textbf{take}\ x\ \{\mathrm{f} = y\} = e_1\ \textbf{in}\ e_2 : \tau}\textsc{Take} \qquad \frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : \{\overline{\mathrm{f}_i^{\mathrm{u}} : \tau_i}, \mathrm{f}^{\bullet} : \tau\}\ s \\ s \neq \textcircled{r} \qquad \Delta;\Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \textbf{put}\ e_1.\mathrm{f} \mathrel{:=} e_2 : \tau}\textsc{Put}$$

$$\frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathrm{A}\ e : \langle \mathrm{A}^{\circ}\ \tau, \overline{\mathrm{A}_i^{\bullet}\ \tau_i}\rangle}\textsc{VCon}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : \langle \mathrm{A}^{\circ}\ \rho, \overline{\mathrm{A}_i^{\mathrm{u}}\ \tau_i}\rangle \\ \Delta; x : \rho, \Gamma_2 \vdash e_2 : \tau \qquad \Delta; y : \langle \mathrm{A}^{\bullet}\ \rho, \overline{\mathrm{A}_i^{\mathrm{u}}\ \tau_i}\rangle, \Gamma_2 \vdash e_3 : \tau\end{array}}{\Delta;\Gamma \vdash \textbf{case}\ e_1\ \textbf{of}\ \mathrm{A}\ x.e_2\ \textbf{else}\ y.e_3 : \tau}\textsc{Case}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash e_1 : \langle \mathrm{A}^{\circ}\ \rho, \overline{\mathrm{A}_i^{\bullet}\ \tau_i}\rangle \qquad \Delta; x : \rho, \Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \textbf{esac}\ e_1\ \textbf{of}\ \mathrm{A}\ x.\,e_2 : \tau}\textsc{Esac}$$

(lists are represented by $\overline{\text{overlines}}$)

Figure 2.3: Typing rules for records and variants

record is represented by the **Share** judgement.

Variants are always unboxed. Once a variant is constructed (see the VCon rule), the given constructor is marked as present, and the usages of all other constructors are marked as taken. These absent alternatives can be arbitrarily chosen, therefore type annotations need to be given by the user to guide COGENT's type inference engine. Pattern matches in the surface language will be desugared to a sequence of **case**-**of** (rule CASE) expressions and a terminating **esac**-**of** (rule ESAC) expression. The former takes two continuations: one for the matched constructor and the other for the remaining variant. When a constructor is matched, we mark it as taken ($^{\bullet}$) in the remaining variant. The **esac**-**of** only handles variants that have one remaining unmatched constructor.

The COGENT compiler used a pre-defined code generation algorithm to compile data types to C. A record type in COGENT was mapped to a C struct, with the fields laid out in the order in which they are declared in the type. The mapping for variant types, on the other hand, was less direct: COGENT's verification tool chain does not support C unions, so variants are also

$$\boxed{\Delta \vdash \tau_1 \sqsubseteq \tau_2}$$

$$\frac{\text{for each } i : \Delta \vdash \tau_i \textbf{ Drop} \qquad \Delta \vdash \tau_i \sqsubseteq \rho_i \\ \text{for each } j : \Delta \vdash \tau_j \sqsubseteq \rho_j}{\Delta \vdash \{\overline{\mathrm{f}_i^{\circ} : \tau_i}, \overline{\mathrm{f}_j^{\mathrm{u}} : \tau_j}\}\, s \sqsubseteq \{\overline{\mathrm{f}_i^{\bullet} : \rho_i}, \overline{\mathrm{f}_j^{\mathrm{u}} : \rho_j}\}\, s}\textsc{RecSub}$$

$$\frac{\text{for each } i : \Delta \vdash \tau_i \sqsubseteq \rho_i \\ \text{for each } j : \Delta \vdash \tau_j \sqsubseteq \rho_j}{\Delta \vdash \langle\overline{\mathrm{A}_i^{\bullet}\ \tau_i}, \overline{\mathrm{A}_j^{\mathrm{u}}\ \tau_j}\rangle \sqsubseteq \langle\overline{\mathrm{A}_i^{\circ}\ \rho_i}, \overline{\mathrm{A}_j^{\mathrm{u}}\ \rho_j}\rangle}\textsc{VarSub}$$

Figure 2.4: Subtyping rules for records and variants

represented as structs in C, containing a field for a tag, and a field for each alternative's payload.

2.3.1 A Remark on Subtyping

COGENT's type system supports subtyping, which deserves further elaboration. Figure 2.4 presents the subtyping judgement for records and variants. The notion of subtyping is closely related to the uniqueness type system that COGENT employs, or more precisely, about the usage of each part of the record or the variant. As we have seen earlier, the usage of a field in denoted by its *takenness* in a record. When it is taken, it means that the ownership has been transferred from the record to another object. The notion of takenness for variant types denotes whether an alternative has been pattern matched or not. In both cases, when two types form a subtyping relation, they only differ in the takenness of their parts. The takenness of fields and alternatives is a static property of types, and it does not affect the generated C code: The taken parts of the type are present in the corresponding C type, except that the C program never accesses them. As a result, types that form a subtyping relation must have the same underlying representation. This is the insight we acquired while experimenting with different subtyping schemes. In particular, COGENT does not employ width-subtyping for composite types, and integer types of different widths do not form any subtyping relation either.

In COGENT, **take**, **put** and pattern matching are all operations that convert objects between subtypes and supertypes. If different C presentations were used, the conversion would have to be accomplished by costly data copying. This is because the generated C code will be verified and reinterpreting memory (e.g. via a C cast) is in general prohibited by the verification framework. By sharing the same memory representation, we essentially restrict subtyping to the static semantics, and no dynamic operations are needed in the runtime, which results in better runtime performance. The performance gain is particularly evident for pattern matches. Since only one constructor is matched at a time in the core calculus, when we exhaust a large variant type, it is achieved by a cascade of matches. If each match incurred a copy, the overhead would cumulate.

Requiring the same representation among subtypes also simplifies the C code generation and the automatic C refinement proof.

2.4 Compilation

The Cogent compilation pipeline is depicted in Figure 2.5. The compiler is developed in a fully modular fashion, which is important for component reuse. Similar to various other programming languages, Cogent has a surface language and a core language. After the source Cogent program text is parsed and typechecked, it is desugared into the core language, which is then followed by a series of core-to-core AST transformations, each with a typechecking phase for sanity checking. The normaliser turns the core AST into A-normal form [Sabry and Matthias Felleisen 1992], which is significantly more verbose. Then the AST is monomorphised by generating specialisations of the original polymorphic functions in accordance with the type arguments used in the program, which are determined by a call-graph analysis. At this stage, the AST can become quite lengthy, if a large number of instances of a polymorphic function are needed. Luckily there is no need for programmers to inspect the intermediate representations (IR) or the verbose target C code, which is obtained by compiling the monomorphised Cogent AST to a C AST and pretty-printing it to a file. From each core AST representation, the compiler can generate Isabelle/HOL embeddings, which we introduce next.

Remarks—readability of generated C code. One of the design philosophies that Cogent's designers have is that the target code does not need to be inspected by human programmers. The Cogent compiler typically produces a large volume of very verbose C code, due to the A-normalisation and monomorphisation phases. This design decision is opposite to some other high-level verification languages, such as Low* in the F* toolchain [Protzenko, Zinzindohoué, et al. 2017], which puts a strong emphasis on producing readable C code, so that domain experts can further inspect and optimise the generated C code. For Cogent, the only occasion that might require developers to directly interact with the generated C code is during debugging; no optimisation or interfacing is done on the C code level. Some extensions to Cogent have been explored to lift debugging to the Cogent level [Warn 2021]. As we shall see in later chapters, the extensions and improvements described in this thesis do not attempt to improve the readability of the C code; we rest upon the do-not-read-C assumption. **End of remarks.**

2.5 Verification

Apart from being a programming language, Cogent is also a verification framework realised in Isabelle/HOL, based on the concept of *certifying compilation*. As we have introduced earlier, compiling a Cogent program results in multiple components:

(1) a C program,

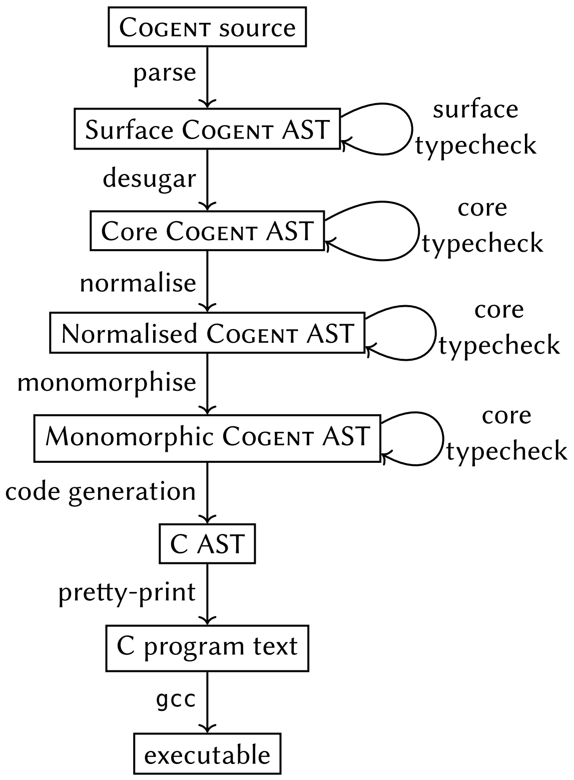


Figure 2.5: The Cogent compilation pipeline

(2) an Isabelle/HOL *shallow embedding* of the Cogent program,

(3) an Isabelle proof of refinement between the C program and the Isabelle shallow embedding.

The C refinement proof relies on the AutoCorres library [Greenaway 2015; Greenaway et al. 2014] to generate a representation of the C code in Isabelle/HOL. More precisely, the AutoCorres library abstracts the C semantics via the formal language SIMPL [Schirmer 2006] into a monadic embedding in Isabelle/HOL.

This compilation process provides an indirect way of formally verifying properties about the generated C program. The programmer first needs to manually prove the desired properties about the Isabelle/HOL shallow embedding. This manual proof should follow by reasoning *equationally* about the HOL term and applying term rewriting tactics provided by the theorem prover. Then, the automatic refinement proof between the C code and the shallow embedding transports the proven properties to the C program. To summarise, Cogent's verification framework reduces complicated low-level verification on the C program to a simple high-level equational proof.

The refinement proof between the C code and the Isabelle/HOL shallow embedding is composed of a series of language-level proofs and translation validation phases, and it is fully automatic, thanks to Cogent's certifying compiler (see Figure 2.6). When the compiler generates the shallow embedding of the Cogent program in Isabelle/HOL, it also generates a *deep embedding* representing the abstract syntax of the Cogent program. Two semantics are assigned to the deep

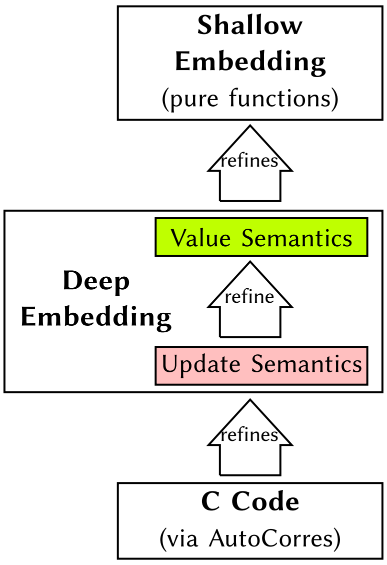


Figure 2.6: The Cogent-C refinement

embedding: a purely functional *value semantics*, and a stateful *update semantics*, with pointers and memory states and in-place field updating. It is proved once-for-all that these two semantics are equivalent for any well-typed Cogent program. As part of the certifying compilation, the compiler produces a refinement proof between the shallow embedding and the deep embedding with value semantics, and a refinement proof between the deep embedding with update semantics and the monadic C embedding obtained from AutoCorres. Chaining the three correspondence lemmas results in a correspondence between the shallow embedding and the C program, stating that the C program is a refinement of the Cogent program's semantics. For more technical details about the C refinement proof, we refer interested readers to our paper [Rizkallah et al. 2016].

With the refinement verification framework that Cogent provides, users can be assured of the correctness of the program as far as the C code level. To ensure that the C code runs correctly on the binary level, the C code could be further compiled with a verified C compiler such as CompCert [Leroy 2009]. However, there is a semantic gap between their C model and that of Schirmer [2006] that we use. For more related work in the area of formalised C semantics, [Krebbers 2015]'s PhD dissertation provides a good overview. The Cogent generated C code also falls into the subset of Sewell et al. [2013]'s `gcc` translation validator, which can be made to compose directly with our compiler certificate.

Chapter 3

Dargent: A Layout Description Language

The material of this chapter is adapted from the following publications:

- Liam O'Connor, Zilin Chen, Partha Susarla Ajay, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Nov. 2018. "Bringing Effortless Refinement of Data Layouts to Cogent." In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation.* Springer, Limassol, Cyprus, 134–149. DOI: https://doi.org/10.1007/978-3-030-03418-4_9
- Zilin Chen, Matt Di Meglio, Liam O'Connor, Partha Susarla Ajay, Christine Rizkallah, and Gabriele Keller. Jan. 2019. *A Data Layout Description Language for Cogent.* at PriSC. Lisbon, Portugal
- Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Jan. 2023. "Dargent: A Silver Bullet for Verified Data Layout Refinement." *Proc. ACM Program. Lang.*, 7, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Article 47, 27 pages. DOI: 10.1145/3571240

The author of this thesis contributed to the design of the Dargent language, the design and implementation of the compiler, the pen-and-paper formalisation of the language, and was involved in discussions about the verification and case studies. The Isabelle/HOL formalisation and proofs presented in Section 3.2.6, Section 3.3.2, and the implementation of the applications presented in Section 3.6 are primarily developed by co-authors of the above publications.

Systems programmers need fine grained control over the memory layout of data structures, both to produce performant code and to comply with well-defined interfaces imposed by existing code, standardised protocols or hardware. Code that manipulates these low-level representations in memory is hard to get right. Traditionally, this problem is addressed by the implementation of tedious marshalling code to convert between compiler-selected data representations and the desired compact data formats. Such marshalling code is error-prone and can lead to a significant runtime overhead due to excessive copying. While there are many languages and systems that address the correctness issue, by automating the generation and, in some cases, the verification of the marshalling code, the performance overhead introduced by the marshalling code remains. In particular for systems code, this overhead can be prohibitive. In this work, we address both the correctness and the performance problems.

We present a data layout description language and data refinement framework, called DARGENT, which allows programmers to declaratively specify how algebraic data types are laid out in memory. Our solution is applied to the COGENT language, but the general ideas behind our solution are applicable to other settings. The DARGENT framework generates C code that manipulates data directly with the desired memory layout, while retaining the formal proof that this generated C code is correct with respect to the COGENT functional semantics. This added expressiveness removes the need for implementing and verifying marshalling code, which eliminates copying, smoothens interoperability with surrounding systems, and increases the trustworthiness of the overall system.

3.1 Introduction

In the realm of software systems, such as device drivers, file systems, and network stacks, precise control over the *data layout* of objects is crucial for compatibility and performance. Specifically, controlling the composition of objects in memory on a bit- and byte-level can avoid the need for translation or *deserialisation* at the boundaries between on-medium and in-memory data which frequently arise from interacting with standardised protocols or hardware.

These systems are often implemented in the C language, in part because it offers low-level features to give fine-grained control over data layout. Unfortunately, to maintain good performance, the C programmer must throw away the conceptual abstraction of the data type, and instead focus on the low-level details of bits and bytes. This low-level code contains many subtle bit-twiddling operations which, apart from being difficult to manually verify, are also tedious and error-prone to implement.

To maintain the higher-level structure of a program without sacrificing performance, we want to use a language with a high-level semantics, but with facilities for specifying the low-level memory layout of heap-allocated objects. Most high-level languages use fixed heap layouts, and COGENT was no exception. Applying such a fixed code generation scheme is no surprise for a typical high-level functional language, whose implementation details are hidden from the language users. But COGENT is not just a functional language, it is also a systems language where the exact low-level representation of data types is relevant to programmers. This fixed code generation algorithm often resulted in suboptimal or undesired representations of COGENT types in C, and users of the COGENT language had to know about the implementation details of the code generator in order to write C code that directly interfaces with COGENT programs. In situations where the Cogent program needs to interoperate with existing C components, say, when developing an operating system component that interfaces with the Linux kernel headers, glue code was required to translate between representations, resulting in development and runtime overhead. Also, problematically, as this glue code depended on the representation choices made by the compiler, future versions of the Cogent compiler could break previously working code due to changes in the code generation scheme.

In this chapter, we present Dargent, a language for describing data layouts of high-level algebraic datatypes, along with a data refinement framework for *automatically verifying* the correctness of the compiled C code with respect to the layout descriptions. We build on the Cogent language and refinement framework (Chapter 2).

While there is a very long line of prior work on *data description languages* [G. Back 2002; Bangert and Zeldovich 2014; Fisher and Gruber 2005; Geest and Swierstra 2017; Kohler et al. 1999; Madhavapeddy et al. 2007; McCann and Chandra 2000; Ramananandro et al. 2019; Slind 2021; Wang and Gaspes 2011; Ye and Delaware 2019], and the data layout descriptions used in Dargent do indeed look similar to those used in such languages, there is a fundamental difference: These languages are designed for synthesising data (de)serialisation functions (also sometimes referred to as data marshalling/unmarshalling functions, encoders/decoders, or parsers and pretty-printers for low-level data), which convert data stored in a low-level, sequential format in some storage medium to a high-level, structured representation in memory and vice versa. They are primarily used for the interaction and communication between different programming languages (e.g. a foreign function interface) or systems (e.g. data transmission over the network). In this context, code to transform back-and-forth between the two representation is still necessary.

Dargent, on the other hand, is intended to solve a different problem. The Dargent data layout descriptions grant programmers the ability to dictate to the Cogent compiler how it should lay out the algebraic data types used by the Cogent program itself. The compiler generates code that works directly with data laid out according to the programmer's specifications, as well as Isabelle/HOL proofs showing that it has done so correctly.

Depending on the application, Dargent therefore can either eliminate entirely or reduce the need for data (de)serialisation code, be it manually written or automatically derived, when interacting with the external world. Because algebraic data types can be represented directly in their binary data formats with Dargent, the programmer does not need to decode raw data first into some other in-memory representation in order to operate on it as a data type. Eliminating these (de)serialisation steps naturally results in more concise and readable code, better performance, and easier informal and formal reasoning.

This additional power in expressiveness can also be used to improve the performance of the compiled Cogent code, e.g. by having smaller memory footprints or a specialised memory layout optimal for the underlying architecture, independent of the interoperation between languages or systems. It also enables Cogent programmers to directly write code that is binary-compatible with native C programs.

Cogent is readily amenable to an extension for prescribing data layouts and fine-tuning the compilation of algebraic data types by virtue of its lack of a runtime system and direct compilation to C. Before Dargent, it simply adopted the layout conventions of the underlying C compiler. The introduction of Dargent into the framework enabled improvements to some outstanding inefficiencies in the prior design, such as a reduced reliance on (de)serialisation code within file system implementations [Amani, Hixon, et al. 2016] and directly representing device register formats as data types (Section 3.6).

Furthermore, we have extended the Cogent compiler so as to preserve the benefits of Cogent's high-level type system and semantics. The upshot is that our compiler automatically translates read and write operations on heap-allocated objects to take account of their particular data layout. The translation's correctness is guaranteed by the enhanced data refinement theorem (Section 3.3.1). To the best of our knowledge, this is the first framework that is able to leverage data layout specifications for generating bit-level accessors with formal correctness guarantees.

We make the following contributions in this chapter:

- We designed and implemented Dargent, a *data layout description* language for controlling the memory layout of algebraic data types, down to the bit level. We present the formalisation of the core calculus and its static semantics (Section 3.2).
- We discuss the compilation process of Cogent code down to C, and the extended verification framework to *automatically verify* the translation of high-level read/write accesses (known as *getters* and *setters*) to explicit offsets within a well-defined memory region (Section 3.3).
- We explore the design space of the Dargent language, and compare with alternative designs (Section 3.4). We also compare and contrast Dargent with data (de)serialisation languages, which are arguably more thoroughly studied (Section 3.5).
- We demonstrate the utility of Dargent in the context of device drivers by an extended suite of examples (Section 3.6).
- We discuss how to use Cogent abstract types and functions to compose an API for manipulating variable-sized data structures, which would require other extensions to Cogent and Dargent to be handled natively (Section 3.7 and Section 3.8).

All the results described in this chapter, including the case studies we present, are associated with formal proofs in Isabelle/HOL. These materials are packaged as a virtual machine image, which is publicly available [Z. Chen, Lafont, O'Connor, et al. 2022]. It is derived from a development branch of the Cogent project repository [The Cogent team 2023], and it was originally submitted as the supplementary material for the POPL'23 paper [Z. Chen, Lafont, O'Connor, et al. 2023], from which this chapter is derived.

```
type Example = {
    struct : #{a : U32, b : Bool},  -- nested embedded record
    ptr : {c : U8},                 -- pointer to another record
    sum : ⟨A U16 | B U8⟩            -- variant type
}
```

Figure 3.1: Example of a COGENT type

3.2 The DARGENT Language

We have designed and implemented a data layout description language called DARGENT, which describes how a COGENT algebraic data type may be laid out in (heap) memory, down to the bit level. Layout descriptions in DARGENT are transparent to the shallow embedding of COGENT's semantics, but they influence the definition of the refinement relation to C code generated by the compiler. In Section 3.3.1, we describe in more detail the extensions to our verification framework to accommodate the DARGENT layout descriptions. Here, we focus on the language definition. To begin with, we give an informal overview of DARGENT's language features.

3.2.1 An Informal Introduction to DARGENT

DARGENT offers the possibility to assign any boxed COGENT type a custom layout describing how the data should be stored in the heap. A boxed type assigned with a custom layout is compiled to a C struct with a single field: an array of 32-bit words.[1] This array *represents* the COGENT type: it can be deemed as untyped in C, but the DARGENT description contains enough information to access individual parts of the type correctly. How data is laid out in memory is purely a low-level concern, and it does not affect the functional semantics of a COGENT program in any way. In other words, COGENT functions are parametric over the layouts of types. Under the hood, the COGENT compiler generates custom getters and setters in C, retrieving and setting the relevant parts of the type from the representing array.

As an example, consider the COGENT type in Figure 3.1. This record consists of three fields: struct, ptr and sum. struct is an unboxed record, denoted by the leading # symbol. This field is embedded inside the parent `Example` record. The ptr field is a boxed record, and is stored somewhere else in the heap, referenced by a pointer in the `Example` type. The last field is a variant type, with two alternatives tagged A and B respectively. This variant is unboxed (recall that there is no boxed variant in COGENT) and is stored in the heap inside the parent record.

A layout for this record type must specify where each field is located in the word array.

[1]Throughout the chapter, unless we explicitly specify the size, the term "word" always refers to unsigned integer types of 1 byte, 2 bytes, 4 bytes or 8 bytes and the actual size is usually less relevant in the discussion. It does not necessarily imply pointer-sized words. We discuss the implications of the choice of the word size later in Section 3.3.1.

$$
\begin{array}{llcll}
\text{Sizes} & s & ::= & n\mathrm{B} \mid n\mathrm{b} \mid s + s & \\
\text{Layout expressions} & \ell & ::= & s & \text{(memory blocks)} \\
 & & \mid & l & \text{(layout variables)} \\
 & & \mid & \mathsf{pointer} & \text{(pointer layout)} \\
 & & \mid & L\ \overline{\ell_i} & \text{(layout names)} \\
 & & \mid & \ell\ \mathsf{at}\ s & \text{(offsets)} \\
 & & \mid & \ell\ \mathsf{after}\ \mathrm{f} & \text{(relative locations)} \\
 & & \mid & \ell\ \mathsf{using}\ \omega & \text{(endianness)} \\
 & & \mid & \mathsf{record}\ \{\overline{\mathrm{f}_i : \ell_i}\} & \\
 & & \mid & \mathsf{variant}\ (\ell)\ \{\overline{\mathrm{A}_i\ (n_i) : \ell_i}\} & \\
 & & \mid & \mathsf{array}\ \{\ell\}\ [n] & \\
\text{Declarations} & d & ::= & \mathbf{layout}\ L\ \overline{l_i} = \ell & \\
\text{Layout names} & & \ni & L & \\
\text{Endianness} & \omega & ::= & \mathsf{BE} \mid \mathsf{LE} & \\
\text{Field names} & & \ni & \mathrm{f} & \\
\text{Constructors} & & \ni & \mathrm{A} & \\
\text{Natural numbers} & & \ni & n, m & \\
\text{Layout contexts} & C & ::= & \overline{l_i \sim \tau_i} &
\end{array}
$$

Figure 3.2: The syntax of the DARGENT surface language

Overlapping is not allowed, except for the payloads of the two constructors of the variant type, since only one of them is relevant at each time, depending on the tag value. The layout must also specify what the tag values for the variant constructors A and B are. We will give a layout to this type after a short introduction to the DARGENT language constructs.

Layouts cannot currently be assigned to *unboxed* records, i.e. records that are not behind a pointer (for example, a stack-allocated local variable). We plan to overcome this limitation in the future.

In Figure 3.2 we present the surface syntax for DARGENT. A *layout expression* is a description of the usage of some (heap) memory. It only describes the low-level view of a memory region—it is not associated to any particular algebraic data type. From this perspective, DARGENT descriptions are independent of COGENT types. As we will shortly see, however, a given COGENT type can only be laid out in certain ways, which places restrictions on which layouts can be assigned to a given type.

When specifying a layout, two pieces of information are relevant: how much space a component occupies in memory and where it is placed in relation to the overall heap object in which it is contained. Primitive types, such as integer types and booleans, are laid out as a contiguous block of memory of a particular size. For example, a contiguous 4-byte block would be an appropriate layout for the 32-bit word type U32. A block of memory is specified as a size expression, which can be in bytes (B), bits (b), or additions of smaller sizes. Additionally, memory blocks of word size (e.g. 1 byte, 2 bytes, 4 bytes and 8 bytes) can be given an endianness (BE or LE), with the using keyword.

A boxed component of a composite type is represented as a pointer, and thus must be described with the special pointer layout, and not as a chunk of memory. This special layout improves readability of code, and also offers better portability: a pointer layout will have different sizes depending on the host machine's architecture.

Layouts for record types use the record construct, which contains sub-expressions for the memory layout of each field. As we can specify memory blocks down to the individual bits, we can naturally represent records of boolean values as a bitfield:

$$\textbf{layout}\ \mathsf{Bitfield} = \mathsf{record}\ \{\mathsf{x} : 1\mathsf{b}, \mathsf{y} : 1\mathsf{b}\ \mathsf{at}\ 1\mathsf{b}, \mathsf{z} : 1\mathsf{b}\ \mathsf{at}\ 2\mathsf{b}\}$$

Here the at operator is used to place each field at a different bit offset, so that they do not overlap. If two record fields have overlapping layouts, the description is rejected by the compiler.

The at operator can be applied to any layout expressions, which will shift the entire expression by the specified amount. Alternatively, the after operator can be used in a record layout to specify the location of a field relative to another one.

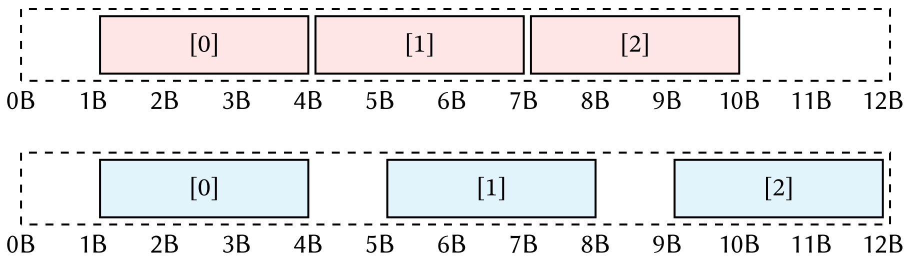
$$\textbf{layout}\ \mathsf{Bitfield} = \mathsf{record}\ \{\mathsf{x} : 1\mathsf{b}, \mathsf{y} : 1\mathsf{b}\ \mathsf{after}\ \mathsf{x}, \mathsf{z} : 1\mathsf{b}\ \mathsf{after}\ \mathsf{y}\}$$

In the layout above, a later field is placed right after the previous one. This saves the programmer from calculating the concrete offset values. When no offset (at or after) is given, it will by default place the field after the previous one. The layout above can be simplied to

$$\textbf{layout}\ \mathsf{Bitfield} = \mathsf{record}\ \{\mathsf{x} : 1\mathsf{b}, \mathsf{y} : 1\mathsf{b}, \mathsf{z} : 1\mathsf{b}\}$$

Layouts for variant types use the variant construct. It firstly requires a layout expression for the *tag*. Then, for each constructor in the variant, a specific tag value needs to be assigned, followed by a layout expression for the *payload* of that constructor. When one alternative is taken, the memory used by other alternatives becomes irrelevant, which is why the memory for the payloads can overlap. Additionally, Dargent allows for zero-sized payloads. For instance, the Maybe *a* type, defined as ⟨Just *a* | Nothing ()⟩, may be given a layout in which the payload for constructor Nothing does not occupy any memory.

Array layouts are used for describing built-in array types in Cogent. Currently we only allow a uniform layout for all elements of an array. Therefore, in the array layout expression, the programmer only needs to supply a layout for the *first* element, along with the length of the array, which is restricted to a constant integer. Unlike layouts for other types, because of the iterative nature of arrays, working out the occupied bits for each array element could be tricky: even though we require all elements to share a uniform layout, each element clearly has a different offset to the beginning of the array. Therefore if an offset is given to ℓ in an array layout array $\{\ell\}\,[n]$, it effectively applies the offset to the entire array, rather than the element's offset to each array cell. For instance, a layout array {3B at 1B} [3] will render an array in the first shape rather than the second one:


[0]
[1]
[2]
0B 1B 2B 3B 4B 5B 6B 7B 8B 9B 10B 11B 12B
[0]
[1]
[2]
0B 1B 2B 3B 4B 5B 6B 7B 8B 9B 10B 11B 12B

To achieve the second layout, some extra work is required. Thankfully, such a layout is arguably less useful in practice, and if it is genuinely desired, one would need to define the element type to be four-byte long, including an explicit padding field for the unused byte in each element. We will give a formal semantics to layout expressions in Section 3.2.5.

Similar to Cogent types, Dargent expressions are also structural. Layout synonyms can be defined using the `layout` keyword, just as Cogent type synonyms are defined using the `type` keyword. For example,

layout `FourBytes` = 4B

defines a layout synonym `FourBytes`, which is definitionally equal to 4B on the right hand side. Layout and type synonyms can take parameters.

We can now give a Dargent description to the `Example` type in Figure 3.1 (assuming a 64-bit architecture):

```
layout ExampleLayout = record {
  struct : record {a : 4B, b : 1b},
  ptr : pointer at 8B,
  sum : variant (1b)
    {A(0) : 2B at 1B, B(1) : 1B at 1B} at 5B
}
```

Figure 3.3 gives a pictorial illustration of this memory layout. In the layout above, it is worth noting that the at 1B offsets for the two payloads of A and B are in relation to the beginning of the sum field, which is 5 bytes (5B) from the beginning of the top-level structure. We can equivalently write variant (1b at 5B) {A(0) : 2B at 6B, B(1) : 1B at 6B} at 0B for the sum field without changing its layout.

At this point, this layout is still independent of the Cogent type, and the compiler will only check that this layout definition is well-formed: that it does not have overlapping fields, that the

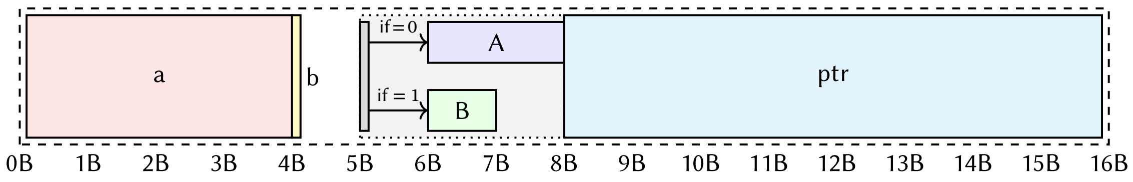


Figure 3.3: The `ExampleLayout`, visualised

tag values are distinct, and so on. To associate a layout to a type, we add a **layout** keyword to the type language of Cogent. For this example, the type `Example` **layout** `ExampleLayout` describes the type `Example` laid out according to the description in `ExampleLayout`. The compiler will check that `ExampleLayout` is an appropriate layout for the Cogent type `Example`. We will talk more about the well-formedness and matching rules later in Section 3.2.5. To reduce verbosity, a type synonym can be given to the layout-annotated type above.

3.2.2 Layout Polymorphism

We extend Cogent's existing parametric polymorphism mechanism to support *layout polymorphism*. This feature allows users to abstract over the layout that they would like to assign to a certain type. In a Cogent function signature, *layout variables*, just as type variables, can be universally quantified. These layout variables may be constrained, similarly to type constraints in Haskell, which require that a layout variable matches a type.[2] For example, in the code snippet below,

$$\begin{aligned}&\textbf{type}\ \mathsf{Pair}\ \alpha = \{\mathsf{fst} : \alpha, \mathsf{snd} : \alpha\}\\ &\textbf{layout}\ \mathsf{LPair}\ l = \mathsf{record}\ \{\mathsf{fst} : l, \mathsf{snd} : l\ \mathsf{at}\ \mathsf{4B}\}\\ &\mathit{freePair} : \forall(\alpha, l :\sim \alpha).\ \mathsf{Pair}\ \alpha\ \textbf{layout}\ \mathsf{LPair}\ l \to ()\end{aligned}$$

we define a parametric type synonym Pair α and layout synonym `LPair` l, and an abstract polymorphic function that operates on such a pair. In the function's type, we require layout l to be compatible with type α, so that the `LPair` l is always a valid layout expression for type Pair α. Layout-polymorphic functions may be explicitly applied to layouts, akin to explicit type applications. For example, *freePair*[`U32`]{`FourBytes`} instantiates α to `U32` and l to `FourBytes`. If the type–layout application is incompatible, as for instance in *freePair*[`U8`]{`1b`}, where `1b` is not big enough to store the `U8`, the typechecker will reject such a program. The typechecker also ensures that any instantiation of l produces layouts that are well-formed. For example, if l is instantiated with `8B`, it will render `LPair` l ill-formed, as the snd field will overlap with the fst. This can be rectified by using the after relative location (or leaving the location implicit) instead, which will automatically place the snd field right after fst.

[2]A formal definition of layout matching is discussed in Section 3.2.5.

Layout polymorphism is necessary to retain the full generality of type polymorphism in the presence of Dargent, as demonstrated by the example above. Layout polymorphism also facilitates code reuse in several scenarios:

(1) It can be used in programs that are architecture-agnostic. For the same program running on different architectures, the same algebraic datatype may be laid out differently according to the hardware it runs on. These datatypes are not necessarily part of the API, as developers tend to design interfaces in an architecture-independent manner and these types thus have uniform layouts across different architectures. However, datatypes that are internal to the program can be represented with appropriate layouts to best suit the architecture and the respective C compiler for better performance.

(2) Besides architectural differences, layout polymorphism allows types to be reused more broadly across different applications. For instance, colours are a general concept independent of the application. One common way to logically represent a colour is the ARGB model. However, the actual low-level layouts for the logical ARGB value vary. Commonly used layouts include ARGB32 and RGBA32, and other layouts also exist. In this case, developers can simply define an ARGB colour type with a polymorphic layout to be determined at use-sites.

(3) Layout polymorphism also facilitates software engineering practices. For example, developers can start with the default layout without a Dargent annotation, and *incrementally* optimise the program to use more compact and efficient Dargent layouts. In this case, they simply instantiate the layouts differently, without needing to re-define the types. Benchmarking is another use case: developers can easily have the same type with different instantiations *side-by-side* to study their performance differences.

3.2.3 Dargent Typechecking

In this section, we give an informal account of the typechecking phase of the surface language. The Dargent surface language will be later desugared into a core calculus (cf. Section 3.2.4), and typechecking will be repeated on the core language. The two typechecking phases are largely similar. Since we only formalise the core language, we will defer the discussion of formal typing rules to the core language in Section 3.2.4. Here we only try to provide some intuition.

The first check on the Dargent layout is its well-formedness. It not only checks for the obvious malformed Dargent definitions such as duplicate field names in a record, but also for the allocated memory for the layout. For example, the fields of a record cannot overlap with each other. This is done by recursively examining each component in the layout definition and observe which bits are used by each substructure. A notion of *allocation* is coined to capture this information. A well-formed layout can be represented as a valid allocation, which is a non-overlapping set of bits that the layout claims to use.

Once the Dargent layouts and Cogent types are all checked to be well-formed, the type-

$$\begin{array}{llcll}
\text{Bit ranges} & r & & & \\
\text{Offsets} & o & \in & \mathbb{N} & \\
\text{Layouts} & \ell & ::= & () & \text{(unit layout)} \\
 & & \mid & r_\omega & \text{(primitive layouts)} \\
 & & \mid & l\{o\} & \text{(layout variables)} \\
 & & \mid & \mathsf{record}\ \{\overline{\mathrm{f}_i : \ell_i}\} & \\
 & & \mid & \mathsf{variant}\ (\ell)\ \{\overline{\mathrm{A}_i\ (n_i) : \ell_i}\} & \\
 & & \mid & \mathsf{array}\ \{\ell\}\ [n] & \\
\text{Field names} & & \ni & \mathrm{f} & \\
\text{Constructors} & & \ni & \mathrm{A} & \\
\text{Endianness} & \omega & ::= & \mathsf{BE} \mid \mathsf{LE} \mid \mathsf{ME} & \\
\text{Natural numbers} & & \ni & n & \\
\text{Layout contexts} & C & ::= & \overline{l_i \sim \tau_i} &
\end{array}$$

Figure 3.4: The syntax of the Dargent core language

checker also ensures that the given layout matches the type to which it is attached. Intuitively, it means that the layout is suitable to store the data of a certain type. For example, the space to store a `U16` needs to be big enough, and a record layout should be used to store the data of a record type. The surface typechecker, as detailed in [O'Connor 2019b], is constraint-based. To resolve the constraints about matching layout-type pairs, the expected pairs of layouts and types will be added to a pool of constraints. Most of the constraints will be decomposed into smaller and simpler constraints during the structural decomposition phase—the simplifier, and eventually bottom out as axioms (e.g. memory blocks and primitive types) or errors. Most of the constraints involving Dargent layouts can be solved independently of other type constraints. Of course, they will need to wait until the relevant type unifiers are instantiated. Apart from the layout-type matching, the surface typechecker is also responsible for the inference of implicit Dargent layout applications, akin to implicit type applications in polymorphic functions.

3.2.4 The Dargent Core Calculus

After the surface typechecking phase, with the inferred types and layouts explicitly annotated in the syntax tree, the Dargent surface language is desugared into a smaller core calculus, whose syntax is outlined in Figure 3.4. The core calculus is the language on which the verification is based. As can be seen in the figure, layouts consist fundamentally of *bit ranges*, which describe which bits in memory are used to store each piece of data. The definition of bit ranges is not given in the figure, because our core calculus is parametric over the exact bit range representation—at different points in the compilation pipeline, bit ranges are represented differently, to suit the purpose of that particular phase of the compilation.

Bit ranges are annotated with an endianness to form a primitive layout in our core language. When the endianness is not specified in the surface language, a *machine endianness* will be given by default, written ME in core. When C code is generated, a subroutine will be invoked to determine the machine endianness, so that the C code works as intended. In this chapter, when the endianness is unimportant, we will leave out the endianness subscript from the syntax r_ω.

When the surface layout expressions are first desugared, the bit ranges are represented as BitRange (o, s), which is a pair of natural numbers, indicating the offset (o bits) from *the beginning of the top-level heap object* in which it is contained, as well as the range occupied (s bits). In BitRange (o, s), we require $s > 0$ for convenience. For zero-sized layouts, the empty layout () can be used instead.

Layout variables are given the form $l\{o\}$, as we additionally need to remember the offset to be applied to the layout variable after its instantiation. When l is instantiated, it will be shifted to the right by o bits. The other core layouts are very similar to their counterparts in the surface language.

In summary, the desugarer converting the surface layout expressions to the core calculus must perform the following tasks:

- expanding layout names to layout definitions;
- computing size expressions into bit ranges;
- computing offsets relative to the beginning of the top-level heap object;
- inserting explicit layout applications to any layout polymorphic function calls.

The desugaring algorithm is a simple inductive definition, which can be found in Figure 3.5. Note that, because the after offset operator needs to know about the fields in the records, they have been expanded to the equivalent at operators in an earlier phase of compilation. Therefore the desugaring algorithm is not concerned with the relative offsets.

Dargent is used for describing the layout of heap memory. To cover all the heap memory that is addressable from an object, it suffices to attach a layout description to all the pointers the object contains. Recall that in Cogent, if an object is referenced by-pointer, it has a boxed type. This is to say that we only need to annotate boxed types with layout information. In the `Example` type that we showed earlier in Figure 3.1, the layout annotation `ExampleLayout` is attached to the `Example` type, which is a boxed type. That layout description contains all the information about how the fields should be laid out, including the unboxed fields struct and sum, which also reside in the heap. For the boxed field ptr, however, the description in `ExampleLayout` does not extend beyond the indirection—it only knows that ptr is a pointer but it is oblivious to the memory layout behind it. To prescribe the layout for {c : `U8`}, a separate layout annotation should be attached to this type, *e.g.* {c : `U8`} **layout** record {c : `1B`}.

As we have discussed informally in Section 2.3, the boxedness of a type is denoted by a sigil in Cogent. Since we only need layouts to be specified at boxed type locations, we can extend

$$\boxed{\ell \overset{dsgr}{\hookrightarrow} \ell'}$$

$$\frac{\mathsf{sizeBits}\ (s) \neq 0 \qquad \ell = \mathsf{BitRange}\ (0, \mathsf{sizeBits}\ (s))}{s \overset{dsgr}{\hookrightarrow} \ell_{\mathsf{ME}}}\textsc{SizeDs} \qquad \frac{\mathsf{sizeBits}\ (s) = 0}{s \overset{dsgr}{\hookrightarrow} ()}\textsc{Size0Ds}$$

$$\frac{}{l \overset{dsgr}{\hookrightarrow} l}\textsc{VarDs} \qquad \frac{\ell = \mathsf{BitRange}\ (0, M)}{\mathsf{pointer} \overset{dsgr}{\hookrightarrow} \ell_{\mathsf{ME}}}\textsc{PtrDs}$$

$$\frac{\ell \overset{dsgr}{\hookrightarrow} \ell'}{\ell\ \mathsf{using}\ \omega \overset{dsgr}{\hookrightarrow} \ell'_{\omega}}\textsc{EndianDs} \qquad \frac{L\ \overline{l_i} = \ell \in \mathsf{TypeDefs} \qquad \ell \overset{dsgr}{\hookrightarrow} \ell'}{L\ \overline{l_i} \overset{dsgr}{\hookrightarrow} \ell'}\textsc{NameDs}$$

$$\frac{\ell \overset{dsgr}{\hookrightarrow} \ell' \qquad \mathsf{sizeBits}\ (s) = b}{\ell\ \mathsf{at}\ s \overset{dsgr}{\hookrightarrow} \mathsf{offset}\ (b, \ell')}\textsc{OffsetDs}$$

$$\frac{\text{for each } i\text{: } \ell_i \overset{dsgr}{\hookrightarrow} \ell'_i}{\mathsf{record}\ \{\overline{\mathsf{f}_i : \ell_i}\} \overset{dsgr}{\hookrightarrow} \mathsf{record}\ \{\overline{\mathsf{f}_i : \ell'_i}\}}\textsc{RecordDs}$$

$$\frac{\ell_t \overset{dsgr}{\hookrightarrow} \ell'_t \qquad \text{for each } i\text{: } \ell_i \overset{dsgr}{\hookrightarrow} \ell'_i}{\mathsf{variant}\ (\ell_t)\ \{\overline{\mathsf{A}\ (v_i) : \ell_i}\} \overset{dsgr}{\hookrightarrow} \mathsf{variant}\ (\ell'_t)\ \{\overline{\mathsf{A}\ (v_i) : \ell'_i}\}}\textsc{VariantDs}$$

$$\frac{\ell \overset{dsgr}{\hookrightarrow} \ell'}{\mathsf{array}\ \{\ell\}\ [n] \overset{dsgr}{\hookrightarrow} \mathsf{array}\ \{\ell'\}\ [n]}\textsc{ArrayDs}$$

$$\begin{aligned}
&\mathsf{offset}\ (o, ()) &&= () \\
&\mathsf{offset}\ (o, \mathsf{BitRange}\ (o', s)) &&= \mathsf{BitRange}\ (o + o', s) \\
&\mathsf{offset}\ (o, l\{o'\}) &&= l\{o + o'\} \\
&\mathsf{offset}\ (o, \mathsf{record}\ \{\overline{\mathsf{f}_i : \ell_i}\}) &&= \mathsf{record}\ \{\overline{\mathsf{f}_i : \mathsf{offset}\ (o, \ell_i)}\} \\
&\mathsf{offset}\ (o, \mathsf{variant}\ (\ell_t)\ \{\overline{\mathsf{A}_i(v_i) : \ell_i}\}) &&= \mathsf{variant}\ (\mathsf{offset}\ (o, \ell_t))\ \{\overline{\mathsf{A}_i(v_i) : \mathsf{offset}\ (o, \ell_i)}\} \\
&\mathsf{offset}\ (o, \mathsf{array}\ \{\ell\}\ [n]) &&= \mathsf{array}\ \{\mathsf{offset}\ (o, \ell)\}\ [n]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sizeBits}\ (n\mathrm{B}) &&= 8n \\
&\mathsf{sizeBits}\ (n\mathrm{b}) &&= n \\
&\mathsf{sizeBits}\ (b_1 + b_2) &&= \mathsf{sizeBits}\ (b_1) + \mathsf{sizeBits}\ (b_2)
\end{aligned}$$

Figure 3.5: Desugaring DARGENT. M is the pointer size in bits; TypeDefs stores the layout synonyms.

the definition of sigils in the core language to accommodate Dargent layouts:

$$\begin{array}{llll} \text{Sigils} & s & ::= & ⓦ_\ell \quad \text{(writable)} \\ & & \mid & ⓡ_\ell \quad \text{(readonly)} \\ & & \mid & ⓤ \quad \text{(unboxed)} \end{array}$$

If the sigil is boxed (either writable or read-only), it can carry a Dargent layout. We use $ⓑ_\ell$ as a notational convenience when we do not wish to distinguish its mutability. When the layout annotations are left out, the type will be compiled with the default layout chosen by the compiler, as introduced in Section 2.3. This ensures that the language is backward compatible and allows for a smooth transition for the programmers.

3.2.5 The Static Semantics with Dargent

In the rest of this section, we give a formal account of the static semantics of the core calculus. Firstly, we present the well-formedness rules for Dargent layouts. The well-formedness of layouts are specified in terms of *allocations*, which denote the set of bits that each layout occupies in memory. In this formalisation, we choose to use *a set of bit ranges* $\overline{\mathsf{BitRange}\ (o_i, s_i)}$ to represent the set of allocated bits. This is not the only way to represent the memory allocation, but it is easy to work with both for the well-formedness checks and for the remainder of the compilation process. As a notational convenience, we use the single element to mean the singleton set that the element comprises. When the unions of bit ranges are taken, we do not need to collapse them into a single bit range. In many cases, they are disjoint anyway and cannot be combined into one.

A layout ℓ is well-formed, if there exists an allocation a such that $\ell \asymp a$ holds. The judgement $\ell \asymp a$ can be read as: the allocation a is reserved for layout ℓ. Formally, $\ell\ \mathbf{wf} \overset{def}{=} \exists a.\, \ell \asymp a$. The allocation rules are specified in Figure 3.6. It only highlights the allocation aspects of the well-formedness, but leaves implicit the scope checking of layout variables, which would require a context but is otherwise straightforward.

One important requirement in the allocation rules is that certain parts of a composite type do not overlap in memory. For example, the fields in a record type cannot overlap, and any payload in a variant type cannot overlap with the tag. For variant layouts, there are a few more things that need to be checked: (1) The tag values have to be distinct, so that the payload can be disambiguated. (2) The tag size is smaller than an **`int`** in C. (3) The tag size is no bigger than necessary to store the tag values; this ensures that there is a unique way to sensibly store the tag value and the layout description has no ambiguity. The second point above deserves some more elaboration. Technically, the restriction on the size of the tag is not necessary and we require it as such mainly for simplicity in code generation. The real technical restriction is, in a Cogent program, regardless of the use of Dargent, if there are more than 2^{31} distinct

$$\boxed{\ell \asymp a}$$

$$\frac{}{() \asymp \text{BitRange}\ (0,0)}\textsc{UnitAlloc} \qquad \frac{}{\text{BitRange}\ (o,s) \asymp \text{BitRange}\ (o,s)}\textsc{PrimAlloc}$$

$$\frac{\ell \asymp a \qquad a = \text{BitRange}\ (o,s) \qquad s \in \{8,16,32,64\}}{\ell\ \text{using}\ \omega \asymp a}\textsc{EndianAlloc}$$

(layout variables are not checked)

$$\frac{\overline{\mathrm{f_i}}\ \text{distinct} \qquad \text{for each } i\text{: } \ell_i \asymp a_i \qquad \overline{a_i}\ \text{disjoint}}{\text{record}\ \{\overline{\mathrm{f_i} : \ell_i}\} \asymp \bigcup_i a_i}\textsc{RecordAlloc}$$

$$\frac{\begin{array}{c}\ell_t \asymp \text{BitRange}\ (o_t,s_t) \qquad \overline{\mathrm{A}_i}\ \text{distinct} \qquad \overline{v_i}\ \text{distinct}\\ s_t \le N \qquad 2^{s_t-1} \le \max \overline{v_i}\\ \text{for each } i\text{: } v_i < 2^{s_t} \qquad \ell_i \asymp a_i \qquad \text{BitRange}\ (o_t,s_t),\ a_i\ \text{disjoint}\end{array}}{\text{variant}\ (\ell_t)\ \{\overline{\mathrm{A}_i\ (v_i) : \ell_i}\} \asymp \bigcup_i a_i \cup \text{BitRange}\ (o_t,s_t)}\textsc{VariantAlloc}$$

$$\frac{\ell \asymp \text{BitRange}\ (o,s)}{\text{array}\ \{\ell\}\ [n] \asymp \bigcup_{i=0}^{n-1} \text{BitRange}\ (i \cdot 8 \cdot \lceil s/8 \rceil + o, s)}\textsc{ArrayAlloc}$$

($\lceil \cdot / \cdot \rceil$ computes the integer division rounded up)

Figure 3.6: Space allocation rules for Dargent. N is the **`int`** size in bits.

constructors, the C verification may fail.[3] This is because tags are compiled to enumeration constants. The C standard [ISO 1999, §6.7.2.2] requires that the enumeration constants to be representable as **`int`**, and accordingly in our C verification tool AutoCorres [Greenaway 2015; Greenaway et al. 2014], enumeration constants are defined to be signed 32-bit words. As the Cogent compiler only uses the non-negative values for enumeration members, we can have at most 2^{31} constructors in any Cogent program. This restriction does not preclude us from having a tag value that is greater than $2^{31} - 1$. When Dargent is used, the value of a tag does not need to agree with the enumeration constant for that tag in the default layout. Dargent allows a Cogent type to be refined differently between the default layout and a custom one, or among custom layouts. What matters is the correspondence between the algebraic constructor and the underlying representation specified by a layout. It is perfectly fine for a constructor A that, the enumeration constant defined for it has value 1, whereas in the Dargent layout we specify it to be $2^{64} - 1$.

In the ArrayAlloc rule, we derive the respective allocation of each element in the array,

[3]Similarly, functions symbols are compiled to enumeration members, therefore there is a similar restriction on the amount of functions.

based on the layout of the first element. From this rule we can see that all elements can be accessed in the same way: the bitwise operations are the same for all elements, but on different bytes in the representing array.

We are unable to fully check the well-formedness of layouts which involve layout variables. For instance, a record layout record $\{\mathrm{f} : l, \mathrm{g} : 1\mathrm{B}\ \mathrm{at}\ 1\mathrm{B}\}$ can either be well-formed or ill-formed depending on what l is instantiated to. This requires that the well-formedness checks on (layout) polymorphic functions to be conducted at call sites, when the layout variables are instantiated to concrete layouts. This is enabled by the fact that in Cogent, polymorphism is restricted to predicative rank-1 universal quantifiers.

The second check after the well-formedness of layouts is the match relation between a layout and a type, which we mentioned earlier in Section 3.2.3. When we attach a Dargent layout to a Cogent type, we need to check that they are compatible. We notice that, as far as the layout is concerned, the structural and representational information of types are relevant, whereas the linearity properties are not. To reflect this, instead of checking directly on types, we apply an erasure function to Cogent types that removes the linearity information. These types are called *type representations*, which are denoted as $\hat{\tau}$. In fact, the notion of linearity erasure and type representation is used not only in layout checks; they are also key ingredients in the C refinement proofs [Rizkallah et al. 2016]. The definition of type representations are given in Figure 3.7.[4] In particular, a function type is unified to a single type fun, which represents a function symbol. This is because in the target language, functions are compiled to function symbols, which are elements of an enumeration in C. We keep the sigils in the type representations, as the sigils store the layout information of the data structure behind the pointer. That allows us to define the $\mathcal{Q} \vdash \hat{\tau}\ \textbf{ok}$ and $\mathcal{Q} \vdash \ell \sim \hat{\tau}$ rules mutually inductively later. Once we collect all the matching pairs that need to be checked, we can coalesce all the boxed type representations to a singleton pointer type representation, similar to how we treated functions.

In Figure 3.8, we show the type well-formedness rules and the companion match rules. Here, we only focus on the Dargent-related aspects of the well-formedness checks; other syntactic or semantic criteria such as duplicate field names or linearity properties are omitted from these rules. The match rules are rooted in the well-formedness checks for boxed types of the form $\mathcal{Q} \vdash \hat{\tau}\ \textbf{ok}$. This is because Dargent layouts are given as part of boxed sigils in types. Such checks can be decomposed into match rules of individual components of boxed composite types. The match rules are of the form $\mathcal{Q} \vdash \ell \sim \hat{\tau}$, meaning that under the assumption $\mathcal{Q}$, layout ℓ can accommodate a type $\hat{\tau}$, where $\mathcal{Q}$ contains the known constraints, which are either axioms or come from the constraint C in the type scheme $\forall \overline{\alpha_i}\ \overline{l_j}.\ C \Rightarrow \tau_1 \rightarrow \tau_2$ of a function's type.

For primitive types, we require the size of the layout to perfectly match the size of the type.

[4]We use a different meta-variable $\hat{\alpha}$ to range over type representation variables, as it subsumes type variables and the !-ed variants of type variables. However, for () and primitive types, we do not make such syntactic differentiation, because they correspond one-to-one with their original counterparts. Note that, for example, `U32!` $\equiv$ `U32`.

$$
\begin{array}{llll}
\text{Type representations} \quad \hat{\tau} & ::= & () & \text{(unit)} \\
 & \mid & \hat{\alpha} & \text{(variables)} \\
 & \mid & \mathsf{fun} & \text{(function)} \\
 & \mid & \{\overline{\mathsf{f}_i : \hat{\tau}_i}\}\ s & \text{(records)} \\
 & \mid & \langle\overline{\mathsf{A}_i\ \hat{\tau}_i}\rangle & \text{(variants)} \\
 & \mid & \hat{\tau}[n]\ s & \text{(arrays)} \\
 & \mid & \texttt{Bool} \mid \texttt{U1} \mid \ldots \mid \texttt{U64} & \text{(prim. types)} \\
 & \mid & \mathsf{C}\ \overline{\hat{\tau}}\ s & \text{(abs. types)}
\end{array}
$$

(lists are represented by $\overline{\text{overlines}}$)

Figure 3.7: Syntax of type representations.

This design may seem overly restrictive, but allowing for a larger layout is ambiguous. Questions like which bits are used to store the data and where the padding is placed can arise. We therefore require an exact match to avoid under-specification. It also simplifies other rules about layouts, as we will see shortly. When dealing with layout variables (rule VarMatch), we need to deduce $l \sim \hat{\tau}$ from the knowledge we have in the context. The judgement $\hat{\tau} \lesssim_{\ell} \hat{\tau}'$ (Figure 3.9) is essentially an implication that $l \sim \hat{\tau}' \implies l \sim \hat{\tau}$. This relation, although defined as an implication, is in fact an equivalence relation in the current setting, as we don't allow under-specifying the layout of a type. We keep this asymmetric notion for future extensions.

As Dargent layouts now become part of the sigils in boxed types, they also affect the subtyping relations among types. This is the third check that we need to perform due to Dargent layouts. As remarked in Section 2.3.1, one key criterion for subtyping is that if two types form a subtyping relation, these two types must have the same underlying representation. With the addition of Dargent, we simply retain this criterion, requiring that the subtype and the supertype have the same Dargent layout, if they are both boxed. From this extension, it again clearly demonstrates the advantages of our design principle of subtyping, without which the subtyping relation would be much more convoluted with Dargent.

Finally, there is another static check that needs to be adapted for Dargent—typechecking polymorphic functions. We need to extend the Cogent typechecking rules to accommodate layout applications. The extended typing rule for simultaneous type/layout applications is as follows (other contexts and checks that are independent of layouts are omitted):

$$
\frac{\mathrm{typeOf}(f) = \forall\, \overline{\alpha_i}\ \overline{l_j}.\ C \Rightarrow \tau_1 \to \tau_2 \qquad \mathcal{Q} \vdash C[\overline{\tau_i/\alpha_i}][\overline{\ell_j/l_j}]}{\mathcal{Q} \vdash f[\overline{\tau_i}]\{\overline{\ell_j}\} : (\tau_1 \to \tau_2)[\overline{\tau_i/\alpha_i}][\overline{\ell_j/l_j}]}\ \textsc{TyLyApp}
$$

It instantiates the constraint C in the type scheme, and the instantiated layout-type pairs should be subject to the match rules displayed in Figure 3.8, under the set of known constraints $\mathcal{Q}$. Note

$$\boxed{\mathcal{Q} \vdash \hat{\tau}\ \mathbf{ok}}$$

$$\frac{\mathcal{Q} \vdash \ell \sim \{\overline{\mathsf{f}_i : \hat{\tau}_i}\}\ \textcircled{u}}{\mathcal{Q} \vdash \{\overline{\mathsf{f}_i : \hat{\tau}_i}\}\ \textcircled{b}_\ell\ \mathbf{ok}}\textsc{RecordOk} \qquad \frac{\mathcal{Q} \vdash \ell \sim \hat{\tau}[n]\ \textcircled{u}}{\mathcal{Q} \vdash \hat{\tau}[n]\ \textcircled{b}_\ell\ \mathbf{ok}}\textsc{ArrayOk} \qquad \frac{\mathcal{Q} \vdash \ell \sim \mathtt{C}\ \hat{\tau}\ \textcircled{u}}{\mathcal{Q} \vdash \mathtt{C}\ \hat{\tau}\ \textcircled{b}_\ell\ \mathbf{ok}}\textsc{AbsTyOk}$$

$$\boxed{\mathcal{Q} \vdash \ell \sim \hat{\tau}}$$

$$\frac{\text{bits}\ (T) = s \qquad T \text{ is a primitive type}}{\mathcal{Q} \vdash \mathsf{BitRange}\ (o, s) \sim T}\textsc{PrimTyMatch} \qquad \begin{array}{ll} \text{bits}\ (\mathsf{U}n) & = n \\ \text{bits}\ (\mathsf{Bool}) & = 1 \end{array}$$

$$\frac{l \sim \hat{\tau}' \in \mathcal{Q} \qquad \hat{\tau} \lesssim_\ell \hat{\tau}'}{\mathcal{Q} \vdash l\{o\} \sim \hat{\tau}}\textsc{VarMatch}$$

$$\frac{\hat{\tau} \text{ is boxed} \qquad \mathcal{Q} \vdash \hat{\tau}\ \mathbf{ok} \qquad M \text{ is the pointer size}}{\mathcal{Q} \vdash \mathsf{BitRange}\ (o, M) \sim \hat{\tau}}\textsc{BoxedTyMatch}$$

$$\frac{N \text{ is the size of } \mathtt{int}}{\mathcal{Q} \vdash \mathsf{BitRange}\ (o, N) \sim \mathsf{fun}}\textsc{FunMatch} \qquad \frac{}{\mathcal{Q} \vdash () \sim ()}\textsc{UnitMatch}$$

$$\frac{\text{for each } i\text{: } \mathcal{Q} \vdash \ell_i \sim \hat{\tau}_i}{\mathcal{Q} \vdash \mathsf{record}\ \{\overline{\mathsf{f}_i : \ell_i}\} \sim \{\overline{\mathsf{f}_i : \hat{\tau}_i}\}\ \textcircled{u}}\textsc{URecordMatch}$$

$$\frac{\text{for each } i\text{: } \mathcal{Q} \vdash \ell_i \sim \hat{\tau}_i}{\mathcal{Q} \vdash \mathsf{variant}\ (\mathsf{BitRange}\ (o_t, s_t))\ \{\overline{\mathsf{C}_i\ (v_i) : \ell_i}\} \sim \langle\overline{\mathsf{C}_i\ \hat{\tau}_i}\rangle}\textsc{VariantMatch}$$

$$\frac{\mathcal{Q} \vdash \ell \sim \hat{\tau} \qquad \ell \text{ is byte aligned}}{\mathcal{Q} \vdash \mathsf{array}\ \{\ell\}\ [n] \sim \hat{\tau}[n]\ \textcircled{u}}\textsc{UArrayMatch}$$

$$\frac{\ell \sim \mathtt{C}\ \hat{\tau}\ \textcircled{u} \in \mathcal{Q} \qquad \ell = \mathsf{BitRange}\ (o, s)}{\mathcal{Q} \vdash \ell \sim \mathtt{C}\ \hat{\tau}\ \textcircled{u}}\textsc{UAbsMatch}$$

Figure 3.8: DARGENT Typing Rules

that we also need to check the well-formedness of all the involved layouts that were previously inhibited by the presence of layout variables. This is left implicit in the rule above.

3.2.6 The Meta-Properties of the Type System

With the augmentation of DARGENT to COGENT's type system, we are able to re-establish the meta-properties about the type system. The changes needed are minimal, and are chiefly around type specialisation (see [O'Connor, Z. Chen, Rizkallah, et al. 2016] for more details). The specialisation of types is a key ingredient in proving the correctness of monomorphisation. It allows us to carry results of the polymorphic language over to the monomorphic instantiations, upon

$$\text{blockSize}\ (\hat\tau) = \begin{cases} \text{bits}\ (\tau) & \text{if } \hat\tau \text{ is a primitive type} \\ N & \text{if } \hat\tau \text{ is a function type} \\ M & \text{if } \hat\tau \text{ is a boxed type} \\ 0 & \text{if } \hat\tau = () \\ \bot & \text{otherwise} \end{cases}$$

(M is the size of a pointer, and N is the size of an **`int`**)

$$\boxed{\hat\tau \lesssim_\ell \hat\tau'}$$

$$\frac{}{\hat\alpha \lesssim_\ell \hat\alpha}\,\textsc{VarCompt} \qquad \frac{\text{blockSize}\ (\hat\tau) \leq \text{blockSize}\ (\hat\tau')}{\hat\tau \lesssim_\ell \hat\tau'}\,\textsc{BlockCompt}$$

$$\frac{\text{for each } i\text{: } \hat\tau_i \lesssim_\ell \hat\tau_i'}{\{\overline{\mathrm{f}_i : \hat\tau_i}\}\ \textcircled{u} \lesssim_\ell \{\overline{\mathrm{f}_i : \hat\tau_i'}\}\ \textcircled{u}}\,\textsc{URecordCompt}$$

$$\frac{\text{for each } i\text{: } \hat\tau_i \lesssim_\ell \hat\tau_i'}{\langle\overline{\mathrm{C}_i\ \hat\tau_i}\rangle \lesssim_\ell \langle\overline{\mathrm{C}_i\ \hat\tau_i'}\rangle}\,\textsc{VariantCompt} \qquad \frac{\hat\tau \lesssim_\ell \hat\tau'}{\hat\tau[n]\ \textcircled{u} \lesssim_\ell \hat\tau'[n]\ \textcircled{u}}\,\textsc{UArrayCompt}$$

Figure 3.9: Type-compatibility rules

which the dynamic semantics is defined. Contrary to the original language, in which only type polymorphism is present, the introduction of layout polymorphism with constraints restricts the domain of layouts and types that can apply to a function. As we have just seen, in the typing judgement, we need to keep an extra context $\mathcal{Q}$, which keeps track of the pairs of layouts and types that need to match. In particular, in the type specialisation lemmas [O'Connor, Z. Chen, Rizkallah, et al. 2016, Lemmas 3 and 4], we can see how the context $\mathcal{Q}$ gets extended with the new pairs induced by the type and layout instantiations:

Lemma 3.1 (Specialisation lemmas)**.** *Let $\overline{\rho_i}$ be a list of well-formed types and $\overline{\ell_j}$ be a list of well-formed layouts. Let $\overline{\alpha_i}$ denote the list of type variables declared in Δ, and $\overline{l_j}$ is a list of layout variables forming L.*

- *$\Delta; L; \mathcal{Q} \vdash \tau$ **wf** implies $\Delta; L; \mathcal{Q}' \vdash \tau[\overline{\rho_i/\alpha_i}, \overline{\ell_j/l_j}]$ **wf**; and*
- *$\Delta; L; \mathcal{Q} \vdash e : \tau$ implies $\Delta; L; \mathcal{Q}' \vdash e[\overline{\rho_i/\alpha_i}, \overline{\ell_j/l_j}] : \tau[\overline{\rho_i/\alpha_i}, \overline{\ell_j/l_j}]$*

when the following conditions hold:

- *for each i, $\Delta; L; \mathcal{Q}' \vdash \rho_i$ **wf**;*
- *for each k such that $\ell_k \sim \hat\tau_k \in \mathcal{Q}$, $L \vdash \ell_k[\overline{\ell_j/l_j}]$ **wf** and $\mathcal{Q} \vdash \ell_k[\overline{\ell_j/l_j}] \sim \hat\tau_k[\overline{\hat\rho_i/\hat\alpha_i}]$,*

where $\mathcal{Q}'$ is $\mathcal{Q}$ extended with $\ell_k[\overline{\ell_j/l_j}] \sim \hat\tau_k[\overline{\hat\rho_i/\hat\alpha_i}]$ for each $\ell_k \sim \hat\tau_k \in \mathcal{Q}$.

In this lemma, the $\Delta; L; \mathcal{Q} \vdash \tau$ **wf** judgement is the Cogent type well-formedness judgement augmented with the $\mathcal{Q} \vdash \hat\tau$ **ok** check outlined in Figure 3.8, and $L \vdash \ell$ **wf** is the well-formedness of layouts implied by the rules in Figure 3.6. $[\overline{\rho_i/\alpha_i}, \overline{\ell_j/l_j}]$ denotes the simultaneous substitution of type variables α_i and layout variables l_j. The lemma states that, if a poly-type τ is well-formed, then instantiating it with well-formed types and layouts preserves its well-formedness. It also states that the type and layout instantiation preserves the typing of expressions. The second condition in the lemma ensures that the instantiation does not render any currently well-formed layout ill-formed, nor does it invalidate any existing layout-type matching registered in $\mathcal{Q}$. Once they are checked, the instantiated pairs will be added to the context $\mathcal{Q}'$, and under the extended context $\mathcal{Q}'$ we check for well-formedness of ρ_i for all i. This reflects how we progressively check for layout well-formedness and layout-type matching as we gain more knowledge about how variables are instantiated.

We give a concrete example to show an application of the lemma.

Example 3.1. *Assuming the appropriate contexts Δ, Δ', L and L'. Let* $\ell_1 \overset{def}{=} \text{record } \{\text{f}_1 : l_1, \text{f}_2 : l_2\}$ *and* $\tau_1 \overset{def}{=} \{\text{f}_1 : \alpha_1, \text{f}_2 : \alpha_2\} \, ⓑ_{\ell_1}$. *We know that* $\Delta; L; \ell_1 \sim \hat{\tau}_1 \vdash \tau_1 \ \mathbf{wf}$. *Let* $\ell_2 \overset{def}{=} \ell_1[\text{1B at 1B}/l_2]$ *and* $\tau_2 \overset{def}{=} \tau_1[\mathtt{U8}/\alpha_2]$. *According to Lemma 3.1, we need to show that ℓ_2 is well-formed and $\ell_2 \sim \hat{\tau}_2$, which can be easily checked as per the rules defined in Section 3.2.5. In conjunction with the other conditions, we are able to conclude that* $\Delta'; L'; \ell_1 \sim \hat{\tau}_1, \ell_2 \sim \hat{\tau}_2 \vdash \tau_2 \ \mathbf{wf}$. *If we further instantiate l_1 and α_1 with* 4B *and* `U32`, *we will get stuck with the well-formedness of* $\ell_2[\text{4B}/l_1]$, *because the two fields overlap in memory. This is only made possible with the extended $\mathcal{Q}'$ context, which contains the partially instantiated ℓ_2.* ◇

3.3 Compiling Dargent

In this section, we shift our focus to the back-end of the compilation, briefly discussing the target C code generation process. Firstly, we look into the generated C code for boxed records when Dargent annotations are present (Section 3.3.1), and we briefly discuss how the C refinement proof is extended to account for layouts (Section 3.3.2). Later, we will discuss arrays and abstract types (Section 3.3.3). Finally we dive in to the core language transformations needed to facilitate code generation (Section 3.3.4).

3.3.1 C Code for Records

Records are the most widely used types in Cogent, and they are currently the only built-in types in Cogent that can be boxed. We therefore use records as an example to discuss how Dargent layouts influence the generated C code.

Without custom layouts, a Cogent record is directly compiled to a C struct with as many fields as the original record: if $T = \{a : A, b : B\}$, then $[\![T]\!]$ is a pointer type to `struct` $\{[\![A]\!]\ a; [\![B]\!]\ b;\}$, where $[\![\cdot]\!]$ denotes the compilation of Cogent types to C types.

The Dargent extension to the compiler relies on the observation that we are free to choose what a Cogent boxed record is compiled to as long as we provide getters and setters for each field, since they account for all the available Cogent operations on boxed records. Assigning a custom layout ℓ to a Cogent record T results in a C type that we denote by $[\![T]\!]_\ell$, consisting of a C struct with a fixed sized array of words as a single field. The implementation chooses the word size to be 32 bits, primarily because most Cogent programs we develop are targeting 32-bit embedded systems, but this is not fundamental to the design and is easily made configurable to any word size (by setting a compiler flag). It is worth mentioning that this does not mean that a layout has to be a neat multiple of 32 bits in size. It is absolutely valid to have a record layout like record {a : 1B, b : 3b}, 11 bits in total. When this layout is embedded in another layout, e.g. record {x : record {a : 1B, b : 3b}, y : 2B}, the remaining bits after the field b will be used by the following field y, without any implicit padding.

The getters and setters for each field are generated according to the layout. If a top-level boxed record T contains a field a : A, the C prototypes are

$$[\![A]\!]\ \texttt{get\_a}\ ([\![T]\!]_\ell\ t);$$
$$\texttt{void set\_a}\ ([\![T]\!]_\ell\ t, [\![A]\!]\ a);$$

Note that $[\![A]\!]$, the return type of the getter (and similarly for the second argument of the setter function) does not involve the layout ℓ. If A is a boxed type, then $[\![A]\!]$ is a pointer to the type A, whose layout is dictated by the layout information stored in A's sigil, independent of ℓ. If A is unboxed, the getter function is the point at which we convert the low-level custom representation governed by the layout ℓ into a standard representation of A, so that the value of the field a can be inspected by the rest of the program. As an example, consider the `struct` field in Figure 3.1. Roughly, the generated C getter has prototype `struct { U32 a; bool b; } get_struct (Example * t)`, where `Example` is a C structure with a single field consisting of a fixed sized array spanning 16 bytes.

Remarks—the choice of the target C types. We use a singleton struct rather than a bare C array for a Dargent-annotated Cogent record. The reason is that we would like to retain a one-to-one correspondence between Cogent types (up to their type representations) and C types. Otherwise, any two Cogent types with the same length will be compiled to the extract same array, which will obscure the correspondence between Cogent types and C types, rendering the C refinement proofs much harder.

As we have briefly mentioned earlier, the choice of the 32-bit word array is primarily to suit the applications that we developed using Dargent and is configurable. When the array is accessed, the size of the array's element type determines the granularity at which we process data. When accessing a field in a record, the memory used by the field is partitioned into a list of *parts*. Bit-wise operations are conducted on each individual part, and the results are combined to form the value of a field.

When we write programs that access registers and memory of I/O devices, the architecture mandates a certain access pattern, and the part size is thus determined by the application. In other cases, the choice of the part size is a matter of performance fine-tuning and can be set to any size that the C compiler and the underlying architecture is optimised for. The smaller the size, the more partitions needed, and vice versa. Currently, we have not yet attempted to systematically study the performance implications of the part size.

When choosing a different part size, the verification of course needs to be adapted accordingly. The C refinement verification works equally well for any chosen sizes: none is easier or more difficult than another, and the proof is fully automatic in all cases, so it is mostly transparent to the end users. What may matter is the amount of C code being generated due to the different part size, and it has an impact, of a varying degree, on the performance of verification (i.e. the resources Isabelle draws to process the proof scripts). **End of remarks.**

Getters and setters are generated recursively following the structure of the involved field types. The process typically involves generating auxiliary *nested* setters and getters that directly manipulate the data array based on the value of the nested field (such as a or b in the example of Figure 3.1), and similarly for getters.

For example, the getter and setter for the `struct` field of Figure 3.1 are roughly implemented as follows[5]:

```
// the data array
typedef struct Example { U32[4] data } Example;
struct field_struct { U32 a; bool b; };

// setter for the struct field
static inline void set_struct(Example * d, struct field_struct v) {
  // calls to nested setters
  set_struct_a(d, v.a);
  set_struct_b(d, v.b);
}

// getter for the struct field
static inline struct field_struct get_struct(Example * d) {
  // calls to nested getters
  return (struct field_struct){.a = get_struct_a(d), .b = get_struct_b(d)};
}
```

As can be seen, a getter function (or similarly a setter) incurs a data format conversion: it turns the low-level data format into a high-level typed value in C. Therefore getting and passing around a large unboxed type can be expensive at run-time. In practice though, if the program is carefully

[5]The Cogent compiler is highly customisable with compiler flags and pragmas. For instance, the C compiler does not have to support compound literals; the Cogent compiler is capable of generating them in alternative ways. The C refinement proof generation may need to be extended with a little extra engineering work, as it is currently tailored to the flags that we use in our experiments.

designed and implemented, and the programmer only passes the minimal structures needed to external functions, there is a good chance that an optimising C compiler is able to eliminate unnecessary data conversions. The Cogent language and its compiler allow users to configure the generated C functions. For instance, as shown in the code snippet above, getters and setters are by default generated as `static inline` functions, which is configurable via compiler flags. Cogent also allows programmers to annotate a Cogent function with a `CINLINE` pragma, so that it is compiled to an `inline` C function, exposing more optimisation opportunities.

Invalid bit patterns Calling a getter on an external word array can lead to unexpected behaviours when the data format is invalid. This can happen, for instance, when variant types are involved, if the value in the tag part does not match any tag values declared in the Dargent layout. Any undefined tag values will be treated as the last constructor's tag. For example, if the layout of a variant is variant (2b) $\{A(0) : 1B, B(1) : 2B, C(2) : 4B\}$ and the tag value seen in the data format is 3, which does not match any defined tag values but also fits in the 2-bit space reserved for the tag, it will be deemed as the last alternative, namely C.

Dargent is not a low-level data parsing language, nor an interface language, therefore the generated C getters do not check for validity, and assume that the input is valid. Users of the language are responsible for checking the validity of any incoming data. These checks are typically performed at the C-Cogent language boundaries. In fact, the integrity of data within a Cogent program should be maintained at all times, regardless of Dargent. Otherwise, the compiler may fail to generate a valid proof, or may result in untrue assumptions in the theorems that can never be discharged. In either case, the soundness of the Cogent verification framework is never compromised.

3.3.2 Formal Verification of Records

As mentioned earlier, the getter and setter for each field are what define the Dargent layout. On the value semantic level (recall Section 2.5 that value semantics is the functional semantics of Cogent), the getters and setters need to satisfy the following two *compositional properties*:

1. **[Roundtrip]** Getting a field should yield a value equivalent to what was previously set;
2. **[Frame]** Setting a field does not affect the result of getting another field.

It is worth mentioning that, somewhat unintuitively, extending the verification framework to account for Dargent does not require proving that the generated getters and setters are accessing data at the locations specified by the layout; it is sufficient to show that they compose correctly as listed above. We prove the compliance of the accessor functions with the Dargent specification as an additional layer.

The verification of any software system must assume the correctness of a *trusted computing base* (TCB) [Rushby 1981]. In Cogent systems, this TCB is largely comprised of externally provided C code. While the Cogent framework allows for manual verification of this C code [Cheung et al. 2022], with Dargent we eliminate large swathes of this C code entirely, specifically the so-called “glue code” which translates between data formats. This reduces the size of the TCB without imposing any additional verification burden.

In Cogent’s refinement story, the addition of Dargent only affects the refinement between Cogent’s stateful update semantics and the generated C code. Above this low-level layer, the functional embeddings of Cogent still use high-level algebraic data types, regardless of the specified layouts. To re-establish the refinement relation between the C code and the Cogent semantics, we developed some Isabelle/ML tactics to automatically prove the following:

(1) the correspondence between Cogent record values and C flat arrays;
(2) the correspondence between record operations in Cogent and in C;
(3) the compositional properties of generated getters and setters stated above;
(4) the correspondence between generated getters/setters and the specified layout.

For more details about the construction of the automatic verification mechanism, we refer interested readers to [Z. Chen, Lafont, O’Connor, et al. 2023].

3.3.3 Arrays and Abstract Types

Arrays and abstract types work slightly differently to record types. We first discuss array types. Due to the iterative nature of arrays, when we generate getter/setter functions for accessing array elements, we would ideally do it schematically: generate one getter and one setter function, which take the index as an argument, and access the element in a uniform manner. This is in contrast to record accessors, in which case we can generate one pair of accessors for each field individually. To simplify this task, we choose the part size to be a single byte, rather than four bytes as we did for records. This allows for an easy calculation of the position of each element in the array, as we require the array elements to be byte aligned.

When accessing unboxed abstract types residing on the heap by value, there are two possible reasons why the compiler-generated getters/setters are not suitable. Firstly, Cogent is oblivious to the internal structures of abstract types. Even though with the size information in the layout, Cogent knows *where* to access such an abstract type, it does not necessarily know *how*. Recall that in Cogent, only heap-objects can have custom layouts with the help of Dargent. It means that the abstract type on the heap may have a different layout to its stack counterpart. When moving an object of an abstract type from the heap to the stack, it may require converting from the custom layout to a default one (and similarly for the other direction). Secondly, the size of the abstract type can be large, making value copying part by part inefficient. Instead, memory area copy functions may be preferred for these tasks, such as the `memcpy` family. To grant such flexibility to the users, the Cogent compiler allows users to define their own getter/setter functions for record fields.[6] The user-defined accessors may have different function signatures than

[6]The record field’s type does not have to be an abstract type; it can be any Cogent type. Custom getters/setters are not yet implemented for array elements, and are not needed for variant types.

the compiler-generated ones. They will suppress the generation of the getters/setters for the relevant fields of a record type. As the custom getter/setter functions are directly defined in C, it means that the user needs to manually prove the correctness properties of these functions.

3.3.4 Bit Range Transformation

After discussing what C code is generated, we now look at how to generate C code, from the core Dargent language. As mentioned in Section 3.2.4, the Dargent core calculus initially represents a bit range as a pair of integers denoted by $\mathsf{BitRange}\ (o, s)$, where o indicates the offset (in bits) to the beginning of the top-level datatype in which it is contained, and s indicates how many bits it occupies. This representation is concise and easy to work with when typechecking the core language. Such a representation, however, does not necessarily lend itself to a simple code generation algorithm. Therefore we have another step to convert each bit range into a list of *aligned bit ranges* tailored for C code generation. Aligned bit ranges is in fact a more formal notion for parts, which we introduced earlier in Section 3.3.1.

An aligned bit range $\mathsf{AlignedBitRange}\ (p, o, s)$ is a triple of integers, where p is the part-offset to the top-level datatype, o is the bit-offset inside a part, and s is the number of bits occupied. Each aligned bit range essentially contains the information about which bits in each part are used. The part-offset p for a part may differ from its position in the list: if no bit in a part is used, we will leave out such an empty part from the list. The following property needs to hold for aligned bit ranges: $\forall \mathsf{AlignedBitRange}\ (p, o, s).\ p \geq 0 \wedge o \geq 0 \wedge s \geq 1 \wedge p + o \leq S$ where S is the part size.

Each generated C getter/setter function consists of a series of statements, each operating on one part by means of bit manipulation. Each such C statement is generated according to the information in one aligned bit range. This one-to-one mapping simplifies C code generation and the C refinement proof.

3.4 Alternative Designs

In this section, we explore some alternative designs that we have considered, ranging from the language design to the verification framework, and compare them with our current design.

In the language design space, we choose to define Dargent layouts separate from Cogent types, and to only relate them afterwards with the **layout** keyword. This design has received a lot of attention, because it seems to be a sub-optimal choice: As some common information is duplicated in both the types and the layouts, it requires us to have a set of dedicated typing rules to relate types and layouts (as we have seen in Figure 3.8). We make this design decision for several reasons.

Firstly, while the Cogent type structure is central to the functional semantics of any Cogent program and the reasoning about its function correctness, the exact layout of algebraic data types is just an implementation detail, totally irrelevant to the top-level Isabelle/HOL embedding of the Cogent program, and whose correctness is guaranteed by the automatic C refinement proof that the Cogent compiler produces. These two constructs are conceptually separate.

Secondly, this approach leaves more flexibility and is amenable to future extensions. Currently, as we have seen in Figure 3.8, the layout-type matching is fairly restricted: e.g. a `U16` type has to occupy 2 bytes (2B) in memory, and a record type has to be laid out in accordance with a record layout. In the future, several extensions might be made to relax this matching relation. For example, it is technically valid to store a smaller type in a larger memory area, say, a `U16` type in a memory region of 4 bytes, or a record type in a contiguous chunk of memory that is big enough. Some heuristics can be implemented in the compiler to decide how to arrange under-specified layouts. This feature can be useful in improving programmers' productivity, because the programmers do not always need to fully specify the layout when the layouts of parts of a type are unimportant. Also, as there is ongoing work towards adding refinement types to Cogent [Paradeza 2020] (also see Chapter 6), a refined type could possibly be laid out in a smaller memory area. For instance, $\{\nu : \texttt{U16} \mid \nu < 2\}$ can be laid out in a memory area as small as 1 bit. None of these extensions would be easy to implement if the layout information was baked into the types.

Thirdly, separating layouts and types encourages modularity. Developers using the Cogent language can write programs without needing to worry about the detailed layout of types, and are still able to prove functional correctness of their code against some high-level specification and ship their code to end users. The end users, with the knowledge of the particular target architecture and environment, can decide the layouts and plug them into the Cogent programs.

Finally, this design decision enables easier compiler engineering. Even though our design requires a dedicated set of rules for checking the layout-type matching, it actually significantly simplifies the compiler engineering in the long term. The Cogent compiler is very large, and the layout-type matching checker only constitutes a small part of the typechecker. The compiler not only compiles Cogent programs to C, but also generates information for various Isabelle proof tactics, numerous embeddings of the program in Isabelle/HOL and Haskell (see Chapter 5). A lot of these embeddings are only concerned with types, and not layouts. If the layouts and types were merged, any changes to the layout implementation would require changes to various irrelevant parts of the compiler.

Alternative designs of the Dargent language have been previously proposed (e.g. [Teege 2019]). Figure 3.10 shows an example of type definitions of a record and a variant in an alternative design, together with their counterparts in the current Dargent design. In this alternative design, the layout annotations are embedded in the type definitions and layouts are no longer first-class. In the record type `R`, the type `U8` implicitly prescribes its layout, which is one byte. On top of that, users can optionally provide more descriptors, such as the endianness. For those layout-agnostic types, such as the unboxed abstract type `#T`, a keyword sized is used to attach

```
-- In an alternative design
type T  -- an abstract type
type R = { f : U8 using BE, g : #T sized 4B at 1B }  -- a record
type V = < 2b | 0 → A (U8 at 1B) | 1 → B (#T sized 4B at 1B)>  -- a variant
```

```
-- In Dargent
type T
type R = { f : U8, g : #T }
type V = <A U8 | B #T>
layout LR = record { f1 : 1B using BE, g : 4B at 1B }
layout LV = variant (2b) { A(0) : 1B at 1B, B(1) : 4B at 1B}
```

Figure 3.10: A record and a variant in the alternative design vs. their DARGENT equivalents

a layout to a type. In the variant type V, the layout of the tag (2b) is included, and the tag value has to be given as part of the type in each alternative.

It is common practice to use types to carry the layout information. In fact, in all the language that we surveyed (see Section 3.9), none of them use a separate first-class layout language in parallel with a type system. In the case of DARGENT, however, for the reasons we stated above, we choose not to merge layouts with types. In particular, we value the modularity and separation of concerns between the algebraic view of objects (types) and the low-level implementation details (layouts). As can be seen in the DARGENT code snippet in Figure 3.10, lines 6–8 are purely about types, contrary to lines 3 and 4, in which the layout and type information is intermingled. The separation of concern is also manifested in the polymorphic function definitions. In DARGENT, we can define generic functions that operate on fixed types but parametric in the layouts used for the types. Once types and layouts are merged, it would require a more complicated subtype system to group the types that share the same algebraic structure, contrary to the use of layout-type matching constraints $l \sim \tau$ that form the layout context (cf. C in Figure 3.4).

* * *

Now we move our focus to the data refinement framework that DARGENT provides. DARGENT helps users solve slightly different problems in two different scenarios. The first scenario is when the users need to specify the layout of types in the memory in order to conform to a specification that is imposed by an industry standard or a protocol. Another scenario is when a COGENT program needs to interoperate with some existing C code. For the former, the layout compliance is very easy to achieve with DARGENT, as the layout information is directly encoded in terms of bits and bytes. For the latter, the users need to know the layout of the native C types, so that they can prescribe the layout of the COGENT types to match the interfacing C types in memory. In general, the existing C types cannot be directly used by Cogent, because they cannot always be abstracted to Cogent-level algebraic data types. It in turn requires the user to have certain level of knowledge about the implementation details of the C compiler being used—more precisely, how the C compiler represents C types in memory in terms of their layout.

Astute readers may have already noticed that, Dargent does not talk about C types at all—all it says is how Cogent types map to the bits and bytes in memory. In fact, as explained in Section 3.3.1, the certifying compilation is not concerned with whether getters and setters access the designated bits and bytes. What Dargent provides is a data refinement between the generated C types (together with the accessors) and the Cogent types (and the corresponding operations). The data layout serves only as a communication medium for relating the Cogent types and the C types.

It is possible to have a different refinement set up, in which the user instead specifies directly how a Cogent type is compiled to a C type. The obvious advantage of this approach is that the compilation and the refinement between the source Cogent code and the compiled C code are independent of the C compiler in use. This allows the user to freely choose any C compiler or switch to a different one at anytime without needing to change the Dargent code. When the exact memory layout becomes relevant, the user can then bring their knowledge about the C compiler on board, just as how one writes programs in C directly. The disadvantage of this approach is that directly defining the correspondence between a Cogent type and a C type (and the respective operations) is more involved. The simple, but arguably less desired, Cogent to C mapping, for example the default Cogent code generation algorithm (i.e. without Dargent), is relatively easy to define. As the correspondence gets more convoluted, say, when the user wants to specify a very compact C representation for an algebraic data type, it quickly becomes less obvious how the correspondence can be encoded. It also makes the synthesis of the accessor functions harder to implement. It may end up requiring the users to manually define the refinement relation, and the accessor functions on the C level, similar to what is required in the property-based testing framework, which we will cover in Chapter 5.

In summary, even though the dependency on the C compiler renders the system less generic, the data layouts do serve as a convenient intermediary for specifying the data refinement down to the C level, requiring the least amount of user input and allowing the compilation and refinement proof to be fully automatic.

3.5 Dargent Is Not for Data Marshalling

As we have argued in Section 3.1, Dargent is designed to specify the memory layout of algebraic datatypes and use this information to guide the compiler to generate code that can directly access individual parts of a data structure without the need to transport or transform data. This is a completely different problem than what data description languages (DDLs) try to solve. They are used to synthesise programs according to a specification to convert the data from one memory presentation to another.

Dargent and DDLs have some technologies in common, though. For example, both of them use a declarative language to specify the memory layout of high-level datatypes, and the compiler is responsible for the heavy-lifting. In fact, the language features present in both types of languages are almost identical. Since the solutions to these two problems are technically similar, is it possible to repurpose Dargent and use it for data (de)serialisation? In summary, there are two major reasons why Dargent cannot be used as a DDL.

Cogent's type system limitations Assuming a record type R, whose in-memory layout is ℓ_m and the on-disk layout is ℓ_d, we can declare the serialisation and deserialisation functions as follows:[7]

$$\begin{array}{lcll} \textbf{type}\ R_m & = & R\ \textbf{layout}\ \ell_m & \\ \textbf{type}\ R_d & = & R\ \textbf{layout}\ \ell_d & \\ \mathit{serial}_R & : & (R_d\ \textbf{take}\ (..), R_m!) & \rightarrow\ R_d \\ \mathit{deserial}_R & : & (R_d!, R_m\ \textbf{take}\ (..)) & \rightarrow\ R_m \end{array}$$

Intuitively, we simply need to copy all the fields from the input record and put them back into the output record. Suppose that $R \overset{def}{=} \{\mathrm{x} : X, \mathrm{y} : Y\}$, then the ostensible definitions are:

$$\begin{array}{lcl} \mathit{serial}_T\ (\mathit{buf}, t) & = & \mathit{buf}\ \{\mathrm{x} = t.\mathrm{x}, \mathrm{y} = t.\mathrm{y}\} \\ \mathit{deserial}_T\ (\mathit{buf}, t) & = & t\ \{\mathrm{x} = \mathit{buf}.\mathrm{x}, \mathrm{y} = \mathit{buf}.\mathrm{y}\} \end{array}$$

The problem with these definitions is that they cannot be defined recursively. Intuitively, if a field of the top-level record is also a record, say X, then the (de)serialisation function for R should subsequently invoke a (de)serialisation for X, *ad infinitum*. This is however not the case with Cogent's type system. Cogent does not support referencing unboxed heap-objects with pointers. The recursion scheme we outlined above bottoms out at unboxed types, including unboxed records and variants. When such a type is taken out from the parent boxed record, it will be moved as a whole to the stack and loses its original layout (recall that stack-objects cannot have layouts in Cogent). It entails that we have to use the stack as an intermediate storage when we transform heap-objects from one format into another.

In practice, having unboxed data embedded in a larger structure is common in the in-memory representation, for fewer indirections and better locality. In the on-disk representation, as everything is stored linearly, all substructures are necessarily unboxed. If we used Dargent to (de)serialise data, we would have to use the stack to store unboxed types, and to pass them around among functions. This is clearly undesirable, not only because of its suboptimal performance, but also because the available stack may be very limited (e.g. on embedded devices). This can be particularly devastating in systems programming, which is usually performance-critical and tight in resources. Large unboxed structures are not uncommon in systems programming,

[7]The T **take**(..) syntax means that all fields in a record T are taken, and the $T!$ syntax indicates that a type T is readonly.

such as the directory entry bitmap and the file name in F2FS's directory entry block data type[8]. In other cases, it might be preferable to situate large data structures unboxed inside other structures in Cogent due to the uniqueness type system, such as data blocks in a file system or packet payloads in a network protocol, even though they are boxed in the typical C implementation.

To solve this problem, we need to augment Cogent's type system with a pointer-reference operator which is capable of referencing unboxed heap-objects by pointers. This extension is not only going to be useful for data (de)serialisation, but it also brings more flexibility and performance benefits to Cogent programming in general. Region types and locations [Fluet et al. 2006; Gay and Aiken 2001; Morrisett et al. 2005; Walker and Watkins 2001] are natural candidate type system extensions to consider. In Section 3.7, we propose a less powerful but simpler extension to Cogent for referencing heap-allocated unboxed types.

Cogent cannot currently handle dynamic-sized structures, which are very common in real-world applications. A typical pattern is that in a record, a numeric field is used to define the length of a later field or the size of the entire record. Cogent's type system needs to know the size of each heap-object in order to statically guarantee uniqueness properties. For a dynamically sized data structure, this is simply impossible. For this reason, Dargent cannot be used to deserialise NULL-terminated arrays, as the size of the type (and thus the memory region) is only known dynamically. In Section 3.7, we present a C library API to handle variable-sized data structures.

Program synthesis The foundation of a DDL is its capability of synthesising programs to perform (de)serialisation according to an abstract, declarative specification. However, Dargent does not synthesise programs. Dargent is designed to *access* datatypes regardless of their layout, rather than *transform* between different data layouts. As we have seen above, to convert the type R from layout ℓ_m to ℓ_d, the definition of the conversion function is not a simple identity. Instead, in the function definition, the programmer needs to manually recursively decompose the input object into units of unboxed types, and then reconstruct the output with these parts. Theoretically, such function definitions can be synthesised. But as Dargent currently stands, it does not do it, which disqualifies Dargent to be a useful DDL.

Summary From the discussion above, although repurposing Dargent for data (de)serialisation is currently infeasible, none of the restriction is intrinsic to Dargent. Once the restrictions from the Cogent side are lifted, with a thin layer on top of Cogent to synthesise the (de)serialisation function code, we anticipate the combination to be suitable as a DDL.

3.6 Programming with Dargent

In the previous section, we showcased a data store implementation that uses Dargent layout mainly on variant types, along with other minor extensions to the language (mainly the zero-

[8] https://elixir.bootlin.com/linux/v6.1.12/source/include/linux/f2fs_fs.h#L579

sized array syntax), to implement variable-sized data types that can subsequently be stored in a fixed-size buffer. The goal was to develop a systems component that is binary compatible with its native C counterpart, and to eliminate obvious performance overhead that would have otherwise been inevitable had we implemented it in Cogent naïvely without such facilities. In this section, we turn our focus back to Dargent itself, and briefly summarise some more applications of Dargent in systems programming, and show how Dargent helps simplify the formal verification of these programs. The full implementations discussed in this section can be found in [Z. Chen, Lafont, O'Connor, et al. 2022].

3.6.1 A Power Control System

To demonstrate the improved readability of programs that Dargent offers, we reimplemented a simple power control system for the STM32G4 series of ARM Cortex microcontrollers by ST Microelectronics. Our Cogent implementation is based on a C implementation from LibOpenCM3 [*LibOpenCM3* n.d.], an open-source low-level hardware library for ARM Cortex-M3 microcontrollers. The original C code[9] and the corresponding Cogent reimplementation are displayed in Figure 3.11a and Figure 3.11b for comparison.

In this type of low-level program, compactness of the data types is key. Keeping the memory footprint small makes the code fit on embedded devices. For that reason, developers want every bit of the memory to be well-utilised. Hence the C implementations heavily rely on bit-twiddling, or some of them use the bit-fields feature in the C language. Bit-fields offers an easier interface to field-operations, but according to the C standard [ISO 1999, §6.7.2.1], the allocation of the bit-fields are implementation-dependent. This is why many drivers are implemented using manual bitwise operations to ensure a certain bit allocation, which is crucial for compliance with the hardware. When the program is implemented in this manner, it often also involves a lot of macro definitions for the bit masking and shifting values, without which the readability and maintainability of the code would be even worse.

In the power control code, a 32-bit power control register holds several pieces of information, each occupying a certain number of bits in the register. To implement it in Cogent, we define the registers as record types (`Cr1`, `Cr2` and `Cr5` in Figure 3.11c) and derive Dargent layouts from the hardware specifications of the device to prescribe the placement of each field.

As we can see from Figure 3.11c, the Dargent layouts are actually the most involved parts of the Cogent implementation. If we bring the line count of the Dargent layout definitions into the equation, the Cogent implementation is not necessarily shorter than its C counterpart, but the added value is in the abstraction that Cogent and Dargent provide, that the semantics of the Cogent functions are easily discernible. Each function in the C version performs a sequence of bitwise operations on the device register. In contrast, in the Cogent implementation, each

[9] `https://github.com/libopencm3/libopencm3/blob/504dc95d9ba1c2505a30d575371accfe49a69fb9/lib/stm32/g4/pwr.c`

```c
void pwr_set_vos_scale(enum pwr_vos_scale scale)
{
  uint32_t reg32;

  reg32 = PWR_CR1 & ~(PWR_CR1_VOS_MASK << PWR_CR1_VOS_SHIFT);
  reg32 |= (scale & PWR_CR1_VOS_MASK) << PWR_CR1_VOS_SHIFT;
  PWR_CR1 = reg32;
}

void pwr_disable_backup_domain_write_protect(void)
{ PWR_CR1 |= PWR_CR1_DBP; }

void pwr_enable_backup_domain_write_protect(void)
{ PWR_CR1 &= ~PWR_CR1_DBP; }

void pwr_set_low_power_mode_selection(uint32_t lpms)
{
  uint32_t reg32;

  reg32 = PWR_CR1;
  reg32 &= ~(PWR_CR1_LPMS_MASK << PWR_CR1_LPMS_SHIFT);
  PWR_CR1 = (reg32 | (lpms << PWR_CR1_LPMS_SHIFT));
}

void pwr_enable_power_voltage_detect(uint32_t pvd_level)
{
  uint32_t reg32;

  reg32 = PWR_CR2;
  reg32 &= ~(PWR_CR2_PLS_MASK << PWR_CR2_PLS_SHIFT);
  PWR_CR2 = (reg32 | (pvd_level << PWR_CR2_PLS_SHIFT) | PWR_CR2_PVDE);
}

void pwr_disable_power_voltage_detect(void)
{ PWR_CR2 &= ~PWR_CR2_PVDE; }

void pwr_enable_boost(void)
{ PWR_CR5 &= ~PWR_CR5_R1MODE; }

void pwr_disable_boost(void)
{ PWR_CR5 |= PWR_CR5_R1MODE; }
```

(a) The C implementation extracted from [*LibOpenCM3* n.d.]

Figure 3.11: The ARM Cortex-M3 power control system

```
pwr_set_vos_scale : (Cr1, Vos_scale) → Cr1
pwr_set_vos_scale (reg, scale) = reg { vos = scale }

pwr_disable_backup_domain_write_protect : Cr1 → Cr1
pwr_disable_backup_domain_write_protect reg = reg { dbp = True }

pwr_enable_backup_domain_write_protect : Cr1 → Cr1
pwr_enable_backup_domain_write_protect reg = reg { dbp = False }

pwr_set_low_power_mode_selection : (Cr1, Lpms) → Cr1
pwr_set_low_power_mode_selection (reg, lpms) = reg { lpms }

pwr_enable_power_voltage_detect : (Cr2, Pvd_level) → Cr2
pwr_enable_power_voltage_detect (reg, pls) = reg { pls, pvde = True }

pwr_disable_power_voltage_detect : Cr2 → Cr2
pwr_disable_power_voltage_detect reg = reg { pvde = False }

pwr_enable_boost : Cr5 → Cr5
pwr_enable_boost reg = reg { r1mode = False }

pwr_disable_boost : Cr5 → Cr5
pwr_disable_boost reg = reg { r1mode = True }
```

(b) The Cogent implementation

Figure 3.11: The ARM Cortex-M3 power control system

function amounts to a mere record field update. The code base is in turn more palatable for programmers to work with and it greatly lowers the (formal and informal) verification cost.

To summarise, this example shows that Cogent allows systems programmers to write code on an abstract level, while Dargent allows them to retain low-level control of the implementation details.

3.6.2 Bit-Fields

In systems code, it is common to have integer values that are stored in fields of non-standard bit widths, as in the following example taken from a CAN driver[10] where the first field is an identifier of 29 bits. This is different from the power control register that we showed above, which also involves non-word-size fields. Such fields in the power control register are modelled as variant types. For example, `Vos_scale` is a two-bit field defined as a variant, with bit patterns `0b01` and

[10] https://github.com/seL4/camkes-vm-examples/blob/89f5d7b7ac373c8e9f000e80b91611e561358ef6/apps/Arm/odroid_vm/include/can_inf.h#L34

```
type Vos_scale = < PWR_SCALE1 | PWR_SCALE2 >
layout LPwr_vosscale = variant (2b) { PWR_SCALE1(1) : 0b, PWR_SCALE2(2) : 0b }

type Lpms = < PWR_CR1_LPMS_STOP_0
            | PWR_CR1_LPMS_STOP_1
            | PWR_CR1_LPMS_STANDBY
            | PWR_CR1_LPMS_SHUTDOWN >
layout LLpms = variant (3b) { PWR_CR1_LPMS_STOP_0(0)   : 0b
                            , PWR_CR1_LPMS_STOP_1(1)   : 0b
                            , PWR_CR1_LPMS_STANDBY(3)  : 0b
                            , PWR_CR1_LPMS_SHUTDOWN(4) : 0b }

type Cr1 = { vos : Vos_scale, dbp : Bool, lpms : Lpms }
  layout record { vos  : LPwr_vosscale at 9b
                , dbp  : 1b            at 8b
                , lpms : LLpms         at 0b}

type Pvd_level = < PWR_CR2_PLS_2V0
                 | PWR_CR2_PLS_2V2
                 | PWR_CR2_PLS_2V4
                 | PWR_CR2_PLS_2V5
                 | PWR_CR2_PLS_2V6
                 | PWR_CR2_PLS_2V8
                 | PWR_CR2_PLS_2V9
                 | PWR_CR2_PLS_PVD_IN >
layout Lpvd_level = variant (3b)
  { PWR_CR2_PLS_2V0(0)    : 0b
  , PWR_CR2_PLS_2V2(1)    : 0b
  , PWR_CR2_PLS_2V4(2)    : 0b
  , PWR_CR2_PLS_2V5(3)    : 0b
  , PWR_CR2_PLS_2V6(4)    : 0b
  , PWR_CR2_PLS_2V8(5)    : 0b
  , PWR_CR2_PLS_2V9(6)    : 0b
  , PWR_CR2_PLS_PVD_IN(7) : 0b }

type Cr2 = { pls : Pvd_level, pvde : Bool }
  layout record { pls  : Lpvd_level at 1b
                , pvde : 1b         at 0b }

type Cr5 = { r1mode : Bool } layout record { r1mode : 1b at 8b }
```

(c) The Cogent types with Dargent layouts

Figure 3.11: The ARM Cortex-M3 power control system

`0b10` representing two predefined scale factors. In the CAN driver, the 29-bit field is modelled as an integer.

```
struct can_id {
  uint32_t id:29;
  uint32_t exide:1;
  uint32_t rtr:1;
  uint32_t err:1;
};
```

In order to represent the 29-bit field, we can use a primitive 29-bit integer `U29`. These non-word-size integers are a new extension we added to Cogent for this purpose, and they required very few changes in the compiler code. We compile such integers to the smallest standard integer type that can contain the type, since the C language does not natively support such types. For instance, `U7` is compiled to `U8`, while `U20` is compiled to `U32`. Thanks to the extension of the value relation (see Section 3.3.2) to those new types, compiled Cogent programs maintain the invariant that C values always fit in the narrower bit-width.

With a Dargent layout, we can define a Cogent version of the above C structure as follows:

```
type CanId = { id : U29, exide : Bool, rtr : Bool, err : Bool }
  layout record { id : 29b, exide : 1b, rtr: 1b, err:1b}
```

The non-word-size integer extension to Cogent, coupled with Dargent, renders the Cogent language expressive enough to represent C bit-fields, and is as abstract as its C counterpart, if not more. It guarantees that the bit-fields are operated in a type-safe manner.

3.6.3 Custom Getters and Setters

Because the semantics of Cogent record is characterised by the getter and setter functions of its fields, it gives us the opportunity to give a *partial view* in Cogent to a C record. This is useful when we write a Cogent system that works on a complicated, externally defined C structure that involves many fields, but we are only concerned with a few specific fields.

Concretely, we can represent this large C structure transparently as a simple Cogent record with only the relevant fields, and define *custom* getters and setters that compose well as per the meta-properties introduced in Section 3.3.2:

```
type Entry = { id : U32, value : U32 }
```

We have successfully applied this approach on a small example, where the C structure has the same fields as the original Cogent type, extended with an additional field. Whilst this example is contrived, this feature does have real application: it makes it possible to *inherit* pre-defined C data structures. For example, in a previous Cogent implementation of a Linux `ext2fs` driver, Amani, Hixon, et al. [2016] had to define their own Cogent inode type `Ext2Inode` which corresponds to the standard C `struct ext2_inode` type, but did not include fields that were irrelevant to the Cogent implementation, such as certain unsupported file modes and spinlocks. This required tedious glue code to marshal back-and-forth between the representations. Using this technique, the standard C `struct ext2_inode` can be used directly as the representation of the `Ext2Inode` type, thus eliminating this glue code entirely.

3.6.4 Verification of a Timer Device Driver

As a case study, we also formally verified, in Isabelle/HOL, a timer driver implemented in Cogent; the driver was successfully run on an ODROID hardware, based on the seL4 operating system [*The seL4 Microkernel* 2016]. Our formalisation took advantage of the fact that the shallow embedding of the Cogent program in Isabelle/HOL remains simple, as the layouts are fully transparent to the functional semantics of the Cogent program.

Our Cogent implementation is based on a C implementation[11]. Both implementations are about the same size (≈60 LoC, excluding type and layout declarations). The timer driver consists of an interface for two timers, called A and E, provided by the device. The E timer can be used to measure elapsed time since its initialisation, and the A timer generates an interrupt at the end of a (possibly periodic) countdown. The driver state is passed around as a C struct, which stores the memory location of the device registers and a boolean flag remembering whether the countdown is disabled.

In the original C implementation, operations on the timer registers are largely based on bitwise operations, which are typeless, unintuitive to read, error-prone, and more difficult to prove correct. As we have shown with other examples, modelling the timer registers as algebraic data types, like records and sum types, makes the program easier to read and to reason about, while Dargent still allows us control over low-level representation details.

To formally verify the driver, we first wrote a purely functional specification of the driver. Then, we proved that the shallow embedding of the Cogent driver refines it. Both the specification and the manual functional correctness proof are approximately 150 lines each. The manual proof is established by straightforward equational reasoning—much easier than reasoning about the bitwise operations implemented in C. Layouts are transparent on the shallow embedding level: Cogent records are encoded as Isabelle records, just as if no layouts were specified. This manual functional correctness proof composed with the automatically generated compiler certificate establishes the correctness of the compiler generated C code. All the tedium in the layout details is successfully hidden by our automatically generated compiler proofs.

The formal verification managed to uncover several bugs or implicit assumptions that had been made in the original C driver. These defects are typically corner cases that can be easily missed by programmers, and are hard to detect with conventional software engineering practices,

[11]`https://github.com/seL4/util_libs/blob/c446df1f1a3e6aa1418a64a8f4db1ec615eae3c4/libplatsupport/src/plat/odroidc2/meson_timer.c`

such as testing.

Firstly, the original C implementation of the initialisation function enabled the countdown timer A without setting a starting value for it. The behaviour of the timer device in this case is unspecified. Another related issue is that the initialisation function does not ensure that the `disable` flag of the driver state is synchronised with an enable flag of the device register, but rather assumes such. We introduced a specific invariant to the functional correctness specification of this function, to make this assumption explicit.

Additionally, when verifying a function that fetches the time from the device, we had to explicitly assume that the timer value in the device state is not too large. While the device provides the timer value in micro-seconds, the function is specified to return the time in nano-seconds. The conversion requires a multiplication by one thousand, possibly triggering an overflow if the timer value is larger than approximately 500 years. Thus, we had to add a precondition to rule out such cases.

3.7 Variable-Sized Data Structures and the Buffer API

Variable-sized structs are a common type of data structure that can be seen in systems code. Typically, such a struct consists of a *header* and a *body* (or *payload*). In the header, it has a field of an integer type, indicating the size of the entire struct, or the length of the payload data it is carrying. The payload of such a struct is placed at the end of the struct; in C, it is usually implemented either as a *flexible array member* (e.g. declared as `char data[];`) at the last field, or simply excluded from the type's definition and is assumed to be place right after the header, accessed via pointer arithmetic.

Typically, such variable-sized data structures are stored consecutively in a pre-allocated fixed-size (e.g. the size of a page) buffer. In order to access an element in the buffer, the programmer needs to start from the beginning of the buffer, and use the length information stored in the header of each element to find the beginning of the next element. This is similar to following the pointers in a linked list in C (even though one is via pointers, and the other via pointer arithmetic), and random access to buffer elements is normally not available.

This pattern poses some challenges if we want to implement it in Cogent. Firstly, the size of each variable-sized element is dependent on the information in its header. From a type systems perspective, they can be best described as dependent records. Supporting dependent records requires a dependent type system and it complicates the Cogent's type system and its certifying compiler dramatically. Secondly, the uniqueness requirement by Cogent's type system cannot be retained if we want to access the elements in the buffer by reference: The unique reference to the buffer owns the entire space allocated to the buffer, and it stops us from having any other references to any parts of the buffer, including buffer elements. This is why in the Cogent implementation of the BilbyFs, Amani [2016] had to deserialise the elements to circumvent the linearity restrictions. This leads to some significant performance overhead, especially when the data (the header and/or the payload) of each element has to be scanned sequentially in order to, say, search for a particular element.

We define a C library API using Cogent's abstract types and abstract functions. The API hides the non-uniqueness from Cogent's type system. The design of this API needs to make the following guarantees:

(1) The interface functions satisfy the language contract imposed by Cogent's type system, so that invoking the API does not compromise the type soundness of other parts of the program.
(2) The C definitions of these API functions are written in a way that can be manually proved to be memory safe relatively easily.

The buffer API, although designed independently, bear some resemblance to the API presented in [Yanovski et al. 2021]. The central concept in the buffer API design is that the buffer has two *views*. One view sees a monolithic buffer object[12] without any knowledge about (nor any access to) the internals, and the other view exposes the buffer's internal structures, allowing for access to disjoint portions of the buffer. By switching the view, at any time, the set of available references are always non-aliasing, which preserves the invariants imposed by the Cogent language. The core function that enables this view-switching idea is the following *focus* function:

```
type FocusFArg a b = #{hd : BHeader, bu : BUsed, bf : BFree, acc : a, obsv : b}
type FocusFRet r   = #{hd : BHeader, bu : BUsed, bf : BFree, res : r}

focus : ∀ (a, b :< DS, r).
        #{buf : Buffer, acc : a, obsv : b, f : FocusFArg a b → FocusFRet r}
      → (r, Buffer)
```

It takes a buffer (buf) as input, along with some accumulators (acc) and observables (obsv). It allows the programmer to run a function f operating on the decomposed view of the buffer (we will come back to the technical details shortly). Once done, the worker function f will recycle the references to the parts of the buffer and exchange them for the reference to the monolithic buffer, switching the view back. When the *focus* function returns, the programmer no longer has access to internal structures of the buffer. This circumvents the aliasing problem we have when accessing the contents in the buffer. The *focus* function itself cannot be implemented in Cogent, as it requires us to reinterpret a chunk of typed memory as an object of a different type (similar to a typecast), which breaks type safety principles. The *focus* function can neither be defined as a primitive operator in Cogent. Since the `Buffer` type is not definable as a native Cogent type in the first place, there is no way to typecheck *focus* in a sensible way. It therefore has to be defined as an abstract function in C and to be verified manually.

With the buffer-content view, we have access to the buffer's header (typed as `BHeader`), its

[12] *Buffer objects* should not be confused with *buffer elements*. We use the former to refer to the whole buffer, not the objects residing in the buffer, which is referred to using the latter term.

occupied portion (`BUsed`), and its unused portion (`BFree`). These three objects are disjoint, but they also amount to the entirety of the buffer. Therefore there is no aliasing, and no memory leaks. Our API provides a set of functions as building blocks, which can be utilised in different combinations to perform a wide range of tasks on the buffer elements. Once the job is done, the *focus* function will exchange the references to the buffer elements back for the buffer object, and normal operations resume from here.

The buffer's header `BHeader` should not be confused with the header of a variable-sized element. The buffer header contains meta-information about the buffer. For example, the buffer header commonly keeps track of the number of elements in the buffer, and holds a pointer to the beginning of the unoccupied portion of the buffer. It is important to note that the buffer header may not physically reside next to the buffer contents in memory. For instance, there may be some global states storing the meta-information about the buffer, which serve as the buffer header. Logically, however, we can consider them to be part of the buffer object. Care shall be taken to ensure that the uniqueness invariants are maintained when accessing (especially updating) the buffer header. Alternatively, we can treat the buffer header as an independent object and thread it through the *focus* function as any ordinary writable linear object. With either design, within the worker function called by *focus*, the buffer header should be accessible and writeable. Very often, the `BHeader` type can be defined in COGENT as a regular COGENT type.

On the other hand, the `BUsed` and `BFree` types are typically only definable in C as raw pointers to buffer memory regions. The assumption is that the buffer is filled up sequentially from the buffer head, and the elements will not be deleted. This may seem very restrictive, and in some cases it indeed is, but there are also a group of applications that can be implemented with these restrictions. Element deletion typically does not require the element in question to be physically erased from the buffer. It is more common to modify the element length information of the previous element. This is similar to rerouting the "next" pointer when removing a node from a linked list. But even for this form of shallow deletion, it requires finer-grained reference tracking that our current buffer API does not yet support. There are some strategies that can be used to signal that an element has been removed without deleting it. For example, a "junk" flag can be kept in each element, and a removed element will have that flag set. Another strategy is employed in the BilbyFs's implementation [Amani 2016]: When one element is to be deleted, a deletion object will be appended to the buffer, indicating that a certain element in the buffer is no longer current. The real deletion happens at a later stage, managed by a garbage collection mechanism.

Now we move on to talk about how buffer elements can be accessed in the buffer-content view. The worker function discussed above exposes two buffer portions: the occupied portion and the unoccupied portion. It does not yet give us more details about individual elements. We introduce the element type `Entry`, which is another important building block in the buffer API design. `Entry` is the type that has some internal size-dependency—the size of the object is remembered by a *length field* in the type. To abstract away from the concrete implementation details of `Entry`, two functions are implemented on `Entry`. One is an *entry_size* : `Entry!` → `U32` function, which returns the size of the object.[13] We also define a *has_next* : `Entry!` → `Bool` function, which is used to query whether this entry is the last one in the buffer. In many applications, the last element has its length field set to `0` as the sentinel. The actual definition of *has_next* depends on how the elements are tracked in the particular application.

Next, we describe a set of API functions that allow the users to work on the elements within the buffer. To ease the reasoning about memory safety, we only allow a single pointer to each portion of the buffer: For the free portion, the pointer will always point to the beginning of it. After all, if the insertion of new elements is always sequential, there is no need to access any places in the free buffer other than the head of it. For the used portion, on the other hand, the single pointer can move along the vector of elements. It starts from the head of the used portion, and can only move forward by one element at a time. It allows the user to scan through all the elements, or to search for and operate on any element. Since there is only one pointer available, it is impossible to implement deletion of elements, which would require pointers to the current element and to the previous one.

The `BUsed` object gives the users access to *an* element, but no more information about the element is available. In particular, it does not say where the element is in the buffer. It thus has an existential trait to it. To inspect the contents of the element in focus, a *read* : `BUsed!` → `Entry!` function is provided by the buffer API. Under the hood, the function *read* casts a raw pointer to the occupied portion of the buffer to an element type (`Entry`). This function may be used to inspect an element, retrieve information from it, but not to update the element in-place.

The API also provides a *next* function, which moves the `BUsed` pointer from its current place to the next element. It has a signature of *next* : (`BUsed`, `BFree!`) → ⟨`This BUsed` | `Next BUsed`⟩. The function returns a variant type, depending on whether a next element exists or not. Due to the linearity requirement of the Cogent language, an object of `BUsed` will be returned in either case. The readonly `BFree` is passed in mainly for sanity checks: that the returned `BUsed` pointer is not running into the region controlled by `BFree`.

There may be a generic (C) implementation of the *next* function: by calling the *has_next* and the *entry_size* function, we know how to find the address of the next element. Then we compare the address and see if it is beyond the `BFree` pointer (the second argument to the *next* function). Knowing the definition of the `Entry` type and some application-specific information, the generic *next* definition can be overwritten with a more appropriate implementation, or a more efficient one. We can see that the definition of *next* has a very strong object-oriented programming flavour: for an interface function, there may be a default implementation, which can be overwritten in an ad hoc manner. The Cogent language currently does not yet have such infrastructure for type classes; we leave the language feature for future work.

[13]`U32` is the integer type of our choice; the width is not essential.

No function in the API discussed so far is able to modify anything within the buffer. Typically, two functions are needed. One is an $\textit{update} : (\texttt{BUsed}, \texttt{Entry} \rightarrow \texttt{Entry}) \rightarrow \texttt{BUsed}$ function, which can be used to perform in-place update on the current element in view. The other is an $\textit{append} : (\texttt{BHeader!}, \texttt{BUsed}, \texttt{BFree}, \texttt{Entry!}) \rightarrow (\texttt{Bool}, \texttt{BUsed}, \texttt{BFree})$ function, which allows one element to be appended to the end of the sequence of existing elements. The returned `Bool` indicates whether the append operation has succeeded or not, and the function also returns the extended occupied portion of the buffer, along with the contracted free portion at the end of the buffer if successful. The invariant is that, before and after the append, no matter whether the operation succeeded or failed, the occupied buffer and the free buffer will always add up to the entire buffer.

* * *

In summary, the buffer API actually solves three separate problems:

(1) it allows for navigating through the variable-sized element types, which can be deemed as a special way of laying out an object, which Dargent does not yet support natively;
(2) it allows users to reinterpret memory (e.g. in the Buffer API it reinterprets the `Buffer` type as a vector of `Entry`s via the `BUsed` and `BFree` as an intermediate step; and
(3) it allows for pointer references to unboxed structures in the heap.

The solutions to (2) and (3) together offer us a way to efficiently access the buffer elements without having to copy (or worse, deserialise) the unboxed structures to the stack, in a way that is safe yet disallowed by Cogent's type system. We need (2) here because there is no native type in Cogent that can be used to represent the internal structure of such a buffer. In fact, the problem (3) stands more broadly, independent of the reinterpretation of the memory. Now we expand on this point a bit more.

The inability to take the pointer-reference to an unboxed data structure in the heap is due to the simplistic uniqueness type system in Cogent. Imagine a naïve $\textbf{take}_{\&}$ operation governed by the following typing rule:

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta; \Gamma_1 \vdash e_1 : \{\cdots, \mathrm{f}^{\circ} : \rho\ \textcircled{u}, \cdots\}\ \textcircled{w} \\ \Delta; x : \{\cdots, \mathrm{f}^{\bullet} : \rho\ \textcircled{u}, \cdots\}\ \textcircled{w}, y : \rho\ \textcircled{w}, \Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta; \Gamma \vdash \textbf{take}_{\&}\ x\ \{\mathrm{f} = y\} = e_1\ \textbf{in}\ e_2 : \tau}\ \textsc{Take-\&}$$

It says that, with the split contexts Γ_1 and Γ_2, if we take out a field f of some unboxed type ρ[14] from a boxed record e_1, then, in the context of the continuation e_2, y is a pointer to the unboxed field f in the heap, denoted by the typing judgement $y : \rho\ \textcircled{w}$. This idea is not so problematic yet, but as soon as we want to define a dual Put-& rule, it immediately falls apart. The type system is not powerful enough to figure out if the pointer put back is the same as the one that was taken

[14]Note that the $\textcircled{u}$ sigil with the unboxed type ρ is somewhat informal, as sigils are currently only defined for records and abstract types.

in the first place. This would normally require a notion of locations (e.g. [Morrisett et al. 2005]) to track the pointers.

COGENT's type system does not offer such fine-grained control over pointer locations. Nevertheless, we can extend the current COGENT language with a read-only MEMBER-& rule, so that it will not have the in-place update problem. The new MEMBER-& rule is sketched out below:

$$\frac{\Delta; \Gamma \vdash e : \{\overline{\cdots, \mathsf{f}^{\circ} : \tau\ ⓤ, \cdots}\}\ ⓡ}{\Delta; \Gamma \vdash e {\rightarrow} \mathsf{f} : \&^{\mathsf{r}}(\tau\ ⓤ)}\textsc{Member-\&} \qquad \begin{array}{lcl} \&^{\mathsf{r}}(\{\overline{\mathsf{f}_i : \tau_i}\}\ ⓤ) & = & \{\overline{\mathsf{f}_i : \tau_i}\}\ ⓡ \\ \&^{\mathsf{r}}(\mathsf{T}\ \tau\ ⓤ) & = & \mathsf{T}\ \tau\ ⓡ \end{array}$$

If the record is read-only, we can access the unboxed field f via a pointer using the new _→_ operator, contrary to the existing _._ member access, which incurs a copy. We also define a meta-level partial function $\&^{\mathsf{r}}$(_) to change the sigil of a type from ⓤ to ⓡ. It is a partial function, which subsequently makes implicit assumptions in the typing rule about what types the _→_ operation can be applied to.

The need to take pointer-references to unboxed structures it not limited to record types. If we had a variant type whose payloads were unboxed structures, a similar operation would turn out to be useful as well. When a heap-allocated variant is pattern matched, it is copied to the stack, as variants are all unboxed. This has several unfortunate consequences: If the unboxed payload is very large, it incurs a significant performance overhead and may cause the stack to overflow. Thus what we need are boxed variant types and a **case-of** alternative that takes the pointer-reference to the payload. The extension is however more involved than that for records.

While adding support for boxed variants, we still want their sigils to be restricted: we only allow ⓤ and ⓡ sigils. Disallowing a writable variant is sensible, because in-place update to a variant is not intuitive in a functional language, especially when the payload is unboxed and each potentially has a different size. Thus we exclude in-place updates to boxed variants from our discussion. Existing typing rules concerning (unboxed) variant types still hold. We only need to focus on the read-only variants. The construction of read-only variants involves memory allocation on the heap, which will be realised by defining abstract functions in COGENT just like boxed records, as no built-in allocation mechanism is provided. In fact, the only additional typing rules needed are the pattern-matching rules, namely a CASE-& and an ESAC-& rule:

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e_1 : \langle \mathsf{A}^{\circ}\ \rho, \overline{\mathsf{A}_i^{\mathfrak{u}}\ \tau_i} \rangle\ ⓡ \\ \Delta; x : \&^{\mathsf{r}}(\rho), \Gamma_2 \vdash e_2 : \tau \qquad \Delta; y : \langle \mathsf{A}^{\bullet}\ \rho, \overline{\mathsf{A}_i^{\mathfrak{u}}\ \tau_i} \rangle\ ⓡ, \Gamma_2 \vdash e_3 : \tau \qquad \Delta \vdash \tau\ \textbf{Escape}\end{array}}{\Delta; \Gamma \vdash \textbf{case}_{\&}\ e_1\ \textbf{of}\ \mathsf{A}\ x.\, e_2\ \textbf{else}\ y.\, e_3 : \tau}\textsc{Case-\&}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \leadsto \Gamma_1 \boxplus \Gamma_2 \\ \Delta; \Gamma_1 \vdash e_1 : \langle \mathsf{A}^{\circ}\ \rho, \overline{\mathsf{A}_i^{\bullet}\ \tau_i} \rangle\ ⓡ \qquad \Delta; x : \rho, \Gamma_2 \vdash e_2 : \tau \qquad \Delta \vdash \tau\ \textbf{Escape}\end{array}}{\Delta; \Gamma \vdash \textbf{esac}_{\&}\ e_1\ \textbf{of}\ \mathsf{A}\ x.\, e_2 : \&^{\mathsf{r}}(\tau)}\textsc{Esac-\&}$$

$$\begin{array}{lcl} \&^{\mathsf{r}}(\cdots) & = & \cdots \\ \&^{\mathsf{r}}(\langle \overline{\mathsf{A}_i\ \tau_i} \rangle\ ⓤ) & = & \langle \overline{\mathsf{A}_i\ \tau_i} \rangle\ ⓡ \end{array}$$

It is crucial to ensure that the resuting type of the pattern match is escapable, so that it does not leak the readonly pointer to the payload of the heap-allocated variant. We also extend the domain of the $\&^{\mathsf{r}}(\_)$ function to variants accordingly. Existing CASE and ESAC rules that operate on unboxed variants can be extended to read-only variants, preserving the behaviour of copying the payload to the stack when it is unboxed:

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e_1 : \langle \mathsf{A}^{\circ}\ \rho, \overline{\mathsf{A}_i^{\mathfrak{u}}\ \tau_i} \rangle\ s \qquad s \neq \textcircled{w} \\ \Delta; x : \rho, \Gamma_2 \vdash e_2 : \tau \qquad \Delta; y : \langle \mathsf{A}^{\bullet}\ \rho, \overline{\mathsf{A}_i^{\mathfrak{u}}\ \tau_i} \rangle\ s, \Gamma_2 \vdash e_3 : \tau\end{array}}{\Delta; \Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathsf{A}\ x.e_2\ \mathbf{else}\ y.e_3 : \tau}\textsc{Case}^*$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Delta; \Gamma_1 \vdash e_1 : \langle \mathsf{A}^{\circ}\ \rho, \overline{\mathsf{A}_i^{\bullet}\ \tau_i} \rangle\ s \qquad s \neq \textcircled{w} \qquad \Delta; x : \rho, \Gamma_2 \vdash e_2 : \tau\end{array}}{\Delta; \Gamma \vdash \mathbf{esac}\ e_1\ \mathbf{of}\ \mathsf{A}\ x.\, e_2 : \tau}\textsc{Esac}^*$$

The newly introduced $\_\rightarrow\_$, $\mathbf{case}_{\&}$**-of** and $\mathbf{esac}_{\&}$**-of** operators all reference unboxed objects in the heap. These new pointer-reference operators are not yet implemented in the compiler. They can however be emulated using abstract functions, as we did earlier with the buffer API's *focus* function. Emulating a single call site of such a pointer-reference operator is relatively straightforward, and each such function can be manually verified. Alas, it does not scale if such operations are used across a large number of data types. To emulate the $\mathbf{case}_{\&}$**-of** operation, it requires one such abstract function for each alternative in every scrutinee type, if we restrict ourselves to exhaustive pattern matches. What we need instead are *schemes* of operators that are parametric in the types that they operate on. Thus defining these operators as primitive operators will improve productivity greatly, on both the implementation and the proof automation fronts.

* * *

Now we return to the discussion about the buffer API and envision its implementation and formal verification. Conceptually, the buffer API works not only on a single `Buffer` type, but rather, it should work uniformly for a class of buffer-like types. This generic nature calls for a language feature which can support some forms of ad hoc polymorphism, for abstraction and function overloading. Such language features are widely available from the literature and real-world languages, such as existential types [Cardelli and Wegner 1985; J. C. Mitchell and Plotkin 1985], type classes in HASKELL [Hall et al. 1996], signatures and functors in standard ML [MacQueen 1984], and classes in object-oriented languages, just to name a few. With such a language feature in place, we can define instances of the buffer API class. The technical challenge is that we need to ensure that the interface functions are general enough so that at least the majority of such buffer-like types can be defined as instances of the class.

On the verification end, we take an axiomatic approach, in which we characterise each interface function by a set of axioms. The axioms specify what properties each interface function needs to satisfy. For each instance of an interface function, we can then manually prove that the function's implementation indeed satisfies (or, refines) the specification prescribed by the axioms. On the update semantics level, the axioms of the interface functions need be sufficiently strong to prove the memory safety properties about the language, playing a similar role to the frame conditions when verifying ordinary abstract functions [Cheung et al. 2022; O'Connor, Z. Chen, Rizkallah, et al. 2016].

In a nutshell, if the axioms for each interface are properly defined and the implementations are in line with the axioms, everything outside of the *focus* function should stay exactly the same as before, and the operations within the invocation of *focus* should be memory safe. Importantly, the API functions may not satisfy Cogent's language contract delineated by the frame conditions, but Cogent's type system ensures that any violation of the frame conditions is restricted to an encapsulated scope—within a call to the *focus* function and the memory region controlled by a `Buffer` object. We need to separately verify that the axioms are strong enough, manually, once-and-for-all, that the use of the interface functions does not undermine the overall memory safety guarantees provided by Cogent.

To give some intuition, we take the *next* function as an example. Informally, the axioms will need to say, if $next\ (bu, bf) = \mathsf{This}\ bu'$, it must be the case that $bu = bu'$ and it is pointing to the last element in the buffer; if $next\ (bu, bf) = \mathsf{Next}\ bu'$, then bu' must be pointing to the next element and it does not escape the region controlled by the `BUsed` type. Unsurprisingly, the formal definition will be more involved and care needs to be taken to ensure that all the details are considered and captured.

3.8 Example: A Data Store

With all the ingredients introduced, we put everything together and in this section we show an example which uses Dargent, the buffer API, and the pointer-reference operations emulated with abstract functions. The example is a miniature data store. Albeit contrived, the core data structures and operations used here resemble those used in BilbyFs.

The data store keeps a heterogeneous list of data entries, which can be a `Person`, an `Address`, or a `Date` (of birth). Each entry has an id number, and entries that share the same id are related. The program reads in the entries from a text file. The entries are in random order in the input text file, and will be stored in the same order as they were read in. For any id that appears in the data store, a `Person` entry of that id must be present, but the `Addr` and `Date` entries are both optional. For simplicity, we do not handle any malformed or corrupted data entries. The only function of this data store is for the user to query the profile of a person by his/her name. It will then display all the available information about the person in question, or prompt that the relevant information is not available in the store.

The data store example is a proof-of-concept exercise trying to capture the core data types and operations in the BilbyFs file system, so that we are confident that the same techniques can be applied to the much larger-scale BilbyFs. We first briefly introduce the relevant aspects of

BilbyFs (and refer the interested readers to Amani's PhD dissertation [Amani 2016] for a fuller picture and a more detailed account), and then justify that the data store is a simplistic yet faithful representation of a real world application.

3.8.1 BilbyFs's Key Data Structures

BilbyFs [Amani 2016] is a log-structured [Rosenblum and Ousterhout 1992] flash file system for Linux. It interfaces with Linux via the virtual file system (VFS) layer and works on raw flash devices that are commonly found in embedded systems via the Unsorted Block Images (UBI) abstraction layer [MTD n.d.].

In BilbyFs, the file system state is recorded on the device as a sequential (or more precisely, circular) log, consisting of a list of log entries. Each entry can either be a file system object (such as inodes, directory entries, data blocks) or a metadata object. The log contains assorted log entries. To locate an entry, an *index* kept in the memory can be used, which maps the object identifiers to their physical addresses on the device. Contrary to typical flash file systems, which store the index on the disk, BilbyFs only stores the index in memory for simplicity. The trade-off is that every time the file system is mounted, the flash needs to be scanned in order to reconstruct the index.[15] When a state-changing operation is performed, more log entries will be appended to the end of the log. When a log entry is updated, it is not removed from the log immediately. Instead, a new entry will be appended to the end of the log, and the superseded entry will be marked obsolete. At a later stage, some bookkeeping will be done by the garbage collector to remove the obsolete entries from the log, restoring space in the device.

For our purpose, it suffices to only look at the abstraction of the flash that BilbyFs provides, and leave all the underlying mechanism behind the scenes. BilbyFs abstracts the flash as an *object store*, which is a sequential collection of objects for log entries. Two preallocated buffers are utilised: a *read buffer* mirroring the certain flash fragment that we are interested in, and a *write buffer* which caches objects to be committed to the flash.

Objects are stored contiguously in the buffers in sequence. All objects share a common header type, and each carries a different type of payload. If we model the buffer as an array of objects, the elements need to have the same type, in a strongly-typed setting (in our case, Cogent). We organise them as a record type `Obj` which contains the fields of the common header, and a variant type at the end which carries different types of payloads.

Some object types are variable-sized, such as directory entry objects containing the file name strings or data objects storing the contents of files. Prior to Dargent, a variant type is compiled to a struct in C, consisting of a tag field and one field for the payload of each alternative. It means that if the payload types are large unboxed types, the overall variant can be gigantic, rendering the memory utilisation very poor (after all, only one alternative is active at any time).

As a result, the data structures used in the Cogent implementation had to deviate from

[15]Summaries are used in BilbyFs to improve the performance, but that is orthogonal to our discussion.

BilbyFs's reference C implementation, by storing a pointer to the variable-sized array, which itself is stored elsewhere, rather than directly embedded in the object type. Consequently, the Cogent implementation of BilbyFs is not binary compatible with its C counterpart, which means that, the Cogent BilbyFs cannot be used to mount a flash device that was previously created with the C version of BilbyFs. Another adaptation made is that every time an object is read from the buffer, it needs to be deserialised. This is because the buffer is already referenced by a linear `Buffer` type, which includes the object in question, and the uniqueness type system forbids aliasing. Similarly, we cannot update buffer elements in-place. When a new object is written to the buffer, instead of filling in the data on the fly, we have to fully prepare all the data, create an object in memory, and serialise it to the buffer. These steps are a major source of performance overhead and implementation complexity. In particular, during (de)serialisation, some data structures are too large to be kept on the stack, and requires heap memory allocation, which is an expensive operation in terms of performance penalties. In the rest of this section, we demonstrate how Dargent, the buffer API, and the pointer-reference operations can be used to fix all these problems with the miniature data store as a prototype.

3.8.2 The Data Store

The data store keeps all the data entries in a pre-allocated fixed-sized buffer. The data entries and the buffer mimic the `Objs` and the read buffer in BilbyFs respectively. The entry type is defined as follows:

```
type Entry =
  { id   : U32
  , size : U32  -- size of the entire Entry
  , data : Data
  } layout record {id : 4B, size : 4B at 4B, data : LData at 8B}

type Data = < Person #TPerson | Addr #TAddr | Date #TDate >
layout LData = variant (2b) { Person(0) : LPerson at 1B
                            , Addr(1)   : LAddr   at 1B
                            , Date(2)   : LDate   at 1B }

type TPerson = {len : U8, gender : Gender, nationality : U8#[2], name : U8#[0]}
type TAddr   = {len : U8, addr : U8#[0]}
type TDate   = {yr  : U16, mon : U8, day : U8}

layout LPerson = record { len         : 1B
                        , gender      : LGender at 1B
                        , nationality : array[1B] at 2B
                        , name        : array[1B] at 4B }
layout LAddr   = record { len : 1B, addr : array[1B] at 1B }
layout LDate   = record { yr  : 2B, mon : 1B at 2B, day : 1B at 3B }

type Gender = < Male | Female >
layout LGender = variant (1b) { Male(0) : 0b, Female(1) : 0b}
```

As can be seen from the definitions above, the `Entry` type has all the key components of the `Obj` type in BilbyFs. It has a size field which is the size of the entire entry, or, equivalently, the offset of the next entry in the buffer to the beginning of the current entry. The data field is a variant which can be one of `Person`, `Addr`, or `Date`, the first two among which have flexible array members, indicated by the `U8#[0]` type. When not used with DARGENT, COGENT compiles it to a `char[0]` type in (GNU) C, which has a very similar semantics to the flexible array member feature as per specified by the C standard [GCC n.d.]. This is because the C code generator that COGENT uses does not support the flexible array member syntax. When used with DARGENT, where all the COGENT types are compiled to word arrays, the flexible array member will not be counted towards the size of the word array. This is in line with the C compiler in which the **`sizeof`**`()` function will consider the size of the flexible array member to be zero. A variant layout is given to the `Data` type, so that the variant type is compiled to a type with the same layout as a C union type with a tag field for disambiguating the union, instead of the space-inefficient default C struct with all the alternatives as fields. A similar treatment is applied to the `Gender` type, which only occupies one bit, rather than as a 12-byte struct (four bytes for the tag, and four bytes each for the two payload types, which are both the unit type (), chosen to be the size of an **`int`** by default).

The algorithm itself is straightforward. The top-level function *find_PersonInfo* (Figure 3.12) is composed of two *focus*es, each one iterating the elements in the buffer. The first loop tries to find the `Person` entry with the queried name and returns the person's id, and the second one uses the retrieved id to find any other entries with the same id. The person's information is aggregated into a `PersonInfo` structure, which is a stack-allocated unboxed record which contains pointers to the name and address strings residing in the buffer. The idea is clear: for small data types, we copy them to the stack; for large ones (i.e. the strings), we instead follow pointers to their original locations.

There is an important caveat, though. In order to return (read-only) pointers to any fragment of the linear buffer type, the buffer object itself also has to be read-only. Recall that in COGENT, we do not allow read-only references to coexist with writeable references to the same or overlapping regions of memory. This restriction therefore implies that, during the entire lifespan of the returned `PersonInfo` object, the buffer object has to be read-only. It necessitate a read-only variant of the *focus* function, which is the *focus_ro* used here. In practice, the input buffer object is usually writable by construction and we need to **let**! the buffer object before we call the *focus_ro* function to turn the object into read-only.

```
type PersonInfo =
  #{ name        : CString!
   , gender      : Gender
   , nationality : U8#[2]
   , dob         : Option (#TDate)
   , addr        : Option (CString!)
   }

find_PersonInfo : (Buffer!, CString!) → Option PersonInfo
find_PersonInfo (buf, name) =
  focus_ro #{buf, acc = (), obsv = name, f = find_person}
  | None → None
  | Some (id, info) → Some (focus_ro #{buf, acc = info, obsv = id, f =
    collect_info})
```

Figure 3.12: The top-level function of the data store example

Each loop is implemented in terms of the *iterate_do* function provided by the Cogent standard library.[16] It resembles a do-while style loop in C. The loop body is comprised of two sub-procedures: a generator and a consumer. The generator takes inputs to the iteration and produces some intermediate result, which can be one of Stop, Yield or Return. They respectively mean that the loop is exhausted and should end naturally, an intermediate result has been generated and further work remains to be done, or the loop decided to exit prematurely. In the case of Yield, the consumer will consume the intermediate result and return one of Stop, Next or Return, where Next means that the loop will continue to the next iteration and the other two with the same meanings as above.

In our example, for both loops, the generators are the same function, which proceeds to the next element in the buffer. For the reason explained above, we also need a read-only *next_ro* function. The *next_ro* function behaves exactly the way we want for a generator: if there are more elements, it proceeds to Yield the next element, otherwise it Stops as it has exhausted all the elements.

Figure 3.13 shows the two consumer functions used in the two loops respectively. They both first *read* the `BUsed` object and extract the element type `Entry`. Once the element object is acquired, a custom getter function *get_Entry_data* for the data field is invoked. This function is an emulation of the _→_ operator. We cannot use the primitive member operation provided by Cogent, since we want to get a pointer to the (unboxed) field rather than make a copy. After obtaining a reference to the `Data` object, the two loops both pattern match the `BoxedData`

[16]https://github.com/au-ts/cogent/blob/master/cogent/lib/gum/common/iterator.cogent

```
cons_entry_Person : #{obj : (), acc : (BUsed!,()), obsv : (BFree!,CString!)}
                  → ConsumerResult (U32, PersonInfo) () (BUsed!, ())
cons_entry_Person (r {acc = (bu, _), obsv = (_, name)}) =
  let entry    = read bu
  and data     = get_Entry_data entry
  and (ret, _) = match_BoxedData #{scr=data, acc=(), obsv=(name, entry),
                   person=get_Person_id_info, addr=skip_Addr, date=skip_Date}
   in ((bu, ()), ret)

cons_entry_by_id : #{obj : (), acc : (BUsed!,PersonInfo), obsv : (BFree!,U32)}
                 → ConsumerResult () () (BUsed!, PersonInfo)
cons_entry_by_id (r {acc = (bu, info), obsv = (_, id)}) =
  let entry = read bu
  and data  = get_Entry_data entry
  and (ret, info) = match_BoxedData #{scr=data, acc=info, obsv=(id, entry),
                      person=is_Person, addr=get_Addr, date=get_Date}
  in ((bu, info), ret)
```

Figure 3.13: The two consumer functions (explicit type applications omitted for brevity)

object (which emulates the boxed version of `Data`) by calling the *match_BoxedData* function. This function emulates a nested variant pattern matching construct involving **case**$_{\&}$-**of** and **esac**:

```
match_BoxedData : ∀ (acc, obsv :< DS, r).
  #{ scr  : BoxedData!
   , acc  : acc
   , obsv : obsv
   , person : (TPersonL!, acc, obsv) → (r, acc)
   , addr   : (TAddrL!  , acc, obsv) → (r, acc)
   , date   : (#TDate   , acc, obsv) → (r, acc)
   } → (r, acc)
```

This function takes three continuations, each of which is executed should the corresponding tag be matched. Note that for the Person and Addr cases, we use the pointer-reference version of the pattern match, whereas we use the traditional payload-copying pattern match in the case of Date, as it is a relatively small data structure. The functions passed to the continuations will extract data from different types of the entries, utilising the relevant _→_ operations emulated with abstract functions.

* * *

To give C definitions to these abstract types such as `Person`, `Data` and `Entry`, we use bit-fields and the `__packed__` attribute to fine-tune the layouts. In fact, in a real-world application, these C types are more likely to pre-exist. They may come from the system that the Cogent program interfaces with, such as the Linux kernel. The job of the Dargent annotations on the Cogent side is thus to serve as an FFI, so that the Cogent data types and their C counterparts are layout-compatible, and can be converted back and forth with a single type cast. This is indeed the case in this example. For instance, the C type for `Entry` is defined as:

```
struct __attribute__ ((__packed__)) DataC {
  unsigned char     :6;
  unsigned char tag :2;
  union {
    PersonC Person;
    AddrC   Addr;
    DateC   Date;
  } payload;
};
```

We can see that the Dargent annotation of the Cogent type `Data` shown at the beginning of Section 3.8.2 correctly prescribes the layout of it in accordance with this C struct definition, assuming the layout of the bit-fields in the C compiler that we use.

3.9 Related Work

The idea of describing low-level data layout with high-level languages is not new. The rich area of research makes it challenging to fully contextualise our work within the space. We can roughly bifurcate the literature into research on *program synthesis* and *program abstraction*.

For instance, Prolac [Kohler et al. 1999], PacketTypes [McCann and Chandra 2000], DataScript [G. Back 2002], Melange [Madhavapeddy et al. 2007], the PADS family of languages [Fisher and Gruber 2005; Fisher and Walker 2011; Mandelbaum et al. 2007], Protege [Wang and Gaspes 2011], Nail [Bangert and Zeldovich 2014], the generic packet description by Geest and Swierstra [2017], the verified Protocol Buffer [Ye and Delaware 2019] built upon the Narcissus framework [Delaware, Suriyakarn, et al. 2019], EverParse [Ramananandro et al. 2019], and contiguity types [Slind 2021] are all concerned with synthesising a parser program (and also a pretty-printer for some of them) from a high-level specification of the data format.

Dargent's primary focus is on the data refinement of algebraic data types rather than securely operating on wire formats. Even though our technology shares a lot in common, the problem we try to solve is very different. In particular, Dargent is not a language for parsing or converting between data formats. It is an extension to Cogent for its compiler to fine-tune the target code generation so that compiled code is already in the desired format that is suitable for systems software. In many cases, Dargent can eliminate the need for such a data marshalling tool entirely.

Along with Dargent, LoCal [Vollmer et al. 2019], SHAPES [Franco, Hagelin, et al. 2017; Franco, Tasos, et al. 2019] and `hobbit` [Diatchki and Jones 2006; Diatchki, Jones, and Leslie 2005] fall in the program abstraction camp and are concerned with compiling data structures in a program into specific layouts dictated by the user. Programmers can therefore still work with high-level source code, while the compiler does the heavy-lifting to generate the low-level mechanisms, retaining the separation of program logic from low-level concerns.

LoCal [Vollmer et al. 2019] is a compiler for a first-order pure functional language that can operate on recursive serialised data by translation into an intermediate *location* calculus, LoCal, mapping pointer indirections of the high-level language to pointer arithmetic calculations on a base address. The final compiler output is C code which, interestingly, preserves the asymptotic complexity of the original recursive functions, although this property is implementation-defined and not assured by any formal theorem. Nonetheless, LoCal's type safety theorem does ensure a form of memory safety: each location is initialised and written to exactly once. The latter property is a key difference to our work, since Dargent can operate on mutable data by virtue of Cogent's linear types. On the other hand, Cogent is a total language and purposely lacks full support for recursion; we therefore do not yet support recursive layout descriptions. Primitive recursive types for Cogent are under development and use records (see [E. Murray 2019]). Since we already support layout descriptions on records, we believe, once recursive types are supported, adding support for recursive layouts would be a straightforward engineering task. As a systems language, Cogent code often uses abstract types such as arrays and iteration constructs over such types. Arrays and iteration constructs over arrays were recently verified through Cogent's FFI [Cheung et al. 2022]. We have ensured these proofs work with our Dargent extensions. The most significant difference with LoCal is that Dargent is a *certifying* data layout language, with generated theorems that the translation is correct, whereas LoCal offers no verified guarantees about its final compiler output.

SHAPES [Franco, Hagelin, et al. 2017; Franco, Tasos, et al. 2019] is an extension to an object-oriented language for fine-tuning the layout of class objects to improve cache performance. It allows users to define layout-unaware classes and specify what layout to use at object instantiation time. This class parameterisation mechanism shares some similarity with Dargent's layout polymorphism. The layouts that SHAPES is concerned with are primarily arrays of values, which are key to better cache locality but are not how compilers of managed languages natively represent data in memory. Their layouts are not down to the bit-level, but rather on the level of record fields. In contrast, Dargent's layouts are lower-level and more flexible, and are less tailored for a specific optimisation. SHAPES's type system maintains memory safety properties of the program when it splits and lays out boxed data types. This bears some resemblance to Cogent's uniqueness type system. In our work, these two aspects are independently managed: Dargent does not directly interfere with memory safety properties guaranteed by COGENT's type system.

The hobbit interpreter [Diatchki, Jones, and Leslie 2005] extends a HASKELL-like functional language with first-class support for bit-level types and operations (e.g. bit concatenation and splitting), supporting external representations of bit-level structures. Their work initially focused on bit-data that can be stored within a single register and later gets extended to memory areas realised as arrays [Diatchki and Jones 2006]. Instead of assigning a high-level type and a low-level layout to an object in memory, their types already prescribe the layouts, by virtue of the first-class bit-data support in the language. In that sense, it is more comparable to the bit-fields feature in the C language, or to COGENT if the DARGENT layout descriptors were subsumed by the COGENT types. Their research novelty also lies in using advanced type system features that are readily available in HASKELL to encode the new language constructs and to perform sophisticated typechecking, which is arguably an orthogonal matter to data layouts.

Floorplan [Cronburg and Guyer 2019] is also somewhat relevant to DARGENT, but does not fit in either category. It is a memory layout specification language for declaratively describing the structure of a heap as laid out by a memory manager. It therefore chiefly serves the implementors of memory managers rather than systems developers and users in general, and the abstraction it provides does not necessarily extends to algebraic types of heap objects. The compiler follows the specification to generate memory safe Rust code to perform common tasks that are needed in the implementation of a memory manager. The semantics of a heap layout specification is denoted by the set of values that the heap can take. In contrast, the semantics of a DARGENT layout is characterised by the getter and the setter functions.

3.10 Conclusion

Systems code must adhere to stringent requirements on data representation to achieve efficient, predictable performance and avoid costly mediation at abstraction boundaries. In many cases, these requirements result in code that is error prone and tedious to write, ugly to read, and very difficult to verify.

By using DARGENT, we can avoid the need for having the *glue* code (be it manually written or synthesised) that marshals data from one format into another, and eliminate error-prone bit-twiddling operations for manipulating specific bits in device registers. Instead, we enable programmers to provide declarative specifications of how their algebraic datatypes are laid out. Given these specifications, our *certifying* compiler generates corresponding C code that operates on these data types directly, along with proofs that the generated code is functionally correct. We have shown the applicability of DARGENT on a number of examples, showcasing its support for low-level systems features including the formal verification of a timer device driver.

Chapter 4

The COGENT-C Foreign Function Interface

This chapter is derived from the following publication:

- Zilin Chen and Christine Rizkallah. 2022. *Why It's Nice to be Quoted: A COGENT FFI.* Unpublished

All the technical development, except for the optimisation of the FFI compiler described in Section 4.6.1 is done by the author of this thesis.

A foreign function interface (FFI) of a language typically exports function and type names, so that programmers can refer to them in a guest language. These exported names are oftentimes mangled to avoid name clashes. This name-based approach to FFI has serious drawbacks and it is sometimes inadequate to use, for instance, if the host language has a structural type system and the types are nameless. In this chapter, we present an alternative design of an FFI in the context of COGENT, a structurally typed polymorphic language implemented in Haskell. The FFI leverages ideas from Haskell's quasiquoting mechanism: Instead of inventing names for the functions and types of the host language, it allows programmers to include host language snippets in the guest language directly. A lightweight FFI compiler will then process the guest language and replace the host language snippets with appropriate guest language code. We provide a recipe for constructing an FFI compiler using existing tools that only requires very little engineering effort. We envision how this approach can be applied to a wide range of languages.

4.1 Introduction

Language interoperability is key in modern, large-scale software development. Employing multiple languages in a single project allows programmers to combine the advantages of each language, and to use language-specific features to solve different problems. In some cases, language interoperation may be necessary because a program implemented in one language needs to communicate with existing components or systems that are implemented in a different language. A *foreign function interface* (FFI) is a mechanism that allows for calling functions implemented in a guest language, and it is widely available in most serious programming languages (both mainstream languages and also many research languages).

The most common and straightforward way of exporting host language objects, such as functions and types, and importing guest language objects, is to reference them by name. For instance, in HASKELL's C FFI [Marlow 2010b], function names declared in HASKELL will be mirrored to the C side. In the code snippet below, the functions `strlen` and `addInt` will be imported from and exported to C respectively.

```
foreign import ccall "string.h strlen"
  cstrlen :: Ptr CChar → IO CSize
foreign export ccall addInt :: Int → Int → Int
```

In other cases, such a simple naming scheme is inadequate to deal with name clashes and name mangling is needed to further disambiguate language objects. In some languages, the *name resolution* mechanism can be quite complicated. For example, Java Native Interface (JNI) can produce very convoluted names for its methods. The documentation excerpt below specifies how the method names are resolved [JNI 2022]:

A native method name is a concatenation of the following components:

- the prefix `Java_`
- a mangled fully-qualified class name
- an underscore ("_") separator
- a mangled method name
- for overloaded native methods, two underscores ("__") followed by the mangled argument signature

Name-based FFIs, in general, come with some (serious) drawbacks, and in some scenarios are infeasible to build. In this chapter, we present an alternative solution to FFI design that avoids the reliance on names. Specifically, we present our problems and show an elegant solution for interfacing with the C language in the context of COGENT [O'Connor 2019b; O'Connor, Z. Chen, Rizkallah, et al. 2016; O'Connor et al. 2021], a purely functional language for systems programming implemented in HASKELL. We dissect the limitations of name mangling in FFI and demonstrate why it is highly unsatisfactory in our scenario. Our FFI between COGENT and C, which we call *antiquoted-C*, connects the two languages via a totally different route.

Antiquoted-C is a dialect of C (see Figure 4.1 for a first impression), in which programmers can quote COGENT types and terms directly without needing to worry about name mangling at all.

The interface language is then lightly processed by a compiler. This FFI compiler is constructed with Cogent compiler modules and tools that are readily available in the Haskell ecosystem with no or little modification. Specifically, the tools it uses are a C parser and pretty-printer from the `language-c-quote` library [Mainland 2021], and a generic programming library [Jeuring et al. 2008] for syntax tree traversal. The resulting interface system is not only easy to use and avoids the drawbacks of name mangling, but also only requires minimal engineering effort to build.
To summarise, the main contributions presented in this chapter are:

- Based on our analysis and experience with name-based FFIs, we argue that name resolution schemes must meet five criteria to be considered practically usable and user-friendly. We illustrate why the criteria cannot be met in the presence of some language features, exemplified by Cogent (Section 4.2).
- We present antiquoted-C, our answer to the Cogent-C FFI problem, and introduce its key features to demonstrate how antiquoted-C effectively satisfies the criteria that we set out without actually taking the name mangling route (Section 4.3).
- We outline the compilation pipeline of antiquoted-C code, and show how the compiler is constructed with minimal engineering effort by reusing existing tools (Section 4.5).
- We evaluate the Cogent-C FFI and the antiquoted-C compiler by performance, development effort, user experience, and discuss alternative designs and future improvements (Section 4.6).
- We share the recipe for building an antiquoted-C style FFI. We believe our approach is transferable to other programming languages and can benefit more language designers (Section 4.6.5).

4.2 Why Antiquoted-C: A Challenge in Assigning Names to Types

Developing an FFI between Cogent and C is conceptually simple. Because Cogent compiles to C, there is no need to be concerned about the calling conventions or the application binary interface (ABI). Furthermore, because Cogent does not have a module system and it is compiled down to a single C file, the FFI only needs to stitch the Cogent-generated C code and the manually written C library together at the C program text level. After that, the combined code can be compiled by an off-the-shelf C compiler like a regular C program.

Recall that a C function can be invoked from Cogent by defining an abstract Cogent function (i.e. a function signature without a definition). On the Cogent side, the abstract function is treated similarly as any ordinary Cogent function. The Cogent compiler compiles the abstract function down to a C function prototype, which is paired with its function body provided by the user in C. Calling a COGENT function from C is also conceptually straightforward because a COGENT function is compiled to a regular C function. No matter which direction the function call goes, the programmer only needs to know the name of the function, and the names of the function's input and output types.

COGENT employs a *structural type system*. In contrast to a *nominal type system*, which is used in most mainstream programming languages [Pierce 2002], two types in a structural type system are, by definition, equal if they have the same structure. Type synonyms (i.e. type names) can usually be defined as shorthand for a structural type, but they are merely syntactic convenience for the programmers and are semantically identical to spelling out the full structure of the type. The user can opt not to define such type synonyms at all and stick with the structural definitions throughout the program and bear with the verbosity.

The structural type system and polymorphism void our plan of exporting COGENT names to C to build the FFI: COGENT types are structural, and can be parametric. It means that COGENT types generally do not have names at all. Moreover, COGENT functions can be polymorphic, and once they are monomorphised by the COGENT compiler, the functions' names will necessarily have to change. The former is in fact the more fundamental issue among the two. If we had a way of naming types, we could subsequently devise a naming convention for monomorphised functions by name mangling, such as appending the names of the type arguments to the function name.

Unfortunately, it is generally a non-solution to demand users to prescribe names for all types occurring at the language boundary. The types that the users may need to reference in their C code are not limited to those of a function's argument and resultant. When defining a COGENT function in C, it is likely that many intermediate types will need to be mentioned. Consider the following deeply nested record type:

$$\textbf{type}\ \texttt{Nested} = \{\mathrm{f}_1 : \{\mathrm{g}_1 : \{\cdots\}, \mathrm{g}_2 : \{\cdots\}, \cdots\}, \mathrm{f}_2 : \{\cdots\}\}$$

To initialise an object of this type, several intermediate structural types are involved, such as the type of the field f_1. Giving the entire record a type name `Nested` is therefore inadequate, as the user may want to declare variables and initialise a field such as g_1 or g_2 and subsequently also f_1 or f_2, which are themselves also structural but remain nameless. In fact, deeply nested types are very common and abundant in real applications of COGENT. For this reason, although mandating programmers to assign names to types only at the language boundary is a plausible idea, it is in fact not practical to do in COGENT in light of its structural type system.

It is also not viable for users to define type synonyms for recurring algebraic types and rely on the compiler to leverage the name abbreviations for all subsequent occurrences of the same structure. In COGENT, type synonyms are top-level and global. For a common structural type, say, a pair of `U32`, it may mean many different things in different parts of a program. It could be a range with a lower and an upper bound, it could as well be an initial index and a step number for a **`for`**-style loop. When a `(U32, U32)` is encountered, the compiler is unable to faithfully determine the actual meaning of the type, and using the wrong type synonym would be more confusing than illuminating.

Additionally, type synonyms in Cogent are only present in the surface language, and types in the core language are nameless, which makes the compilation and the verification much more tractable. Choosing a structural type system in a research programming language is common practice, and is justifiable [Pierce 2002]. The problem we are facing is how to create names for nameless types in a principled manner. More specifically, we define the following criteria for a valid, practically usable, and user-friendly name resolution scheme:

1. **[injective]** the generation of type names is injective—namely, if two types are different, the names of the types are also different;
2. **[concise]** the length of the names are reasonably short and are practical to be used by programmers;
3. **[stable]** the naming of types is stable—for any structural type, it is always compiled to the same name regardless of the context and the ambient program;
4. **[a priori]** the name of a type can be inferred from the type's definition, without needing to run the compiler and inspect the result;
5. **[indicative]** the name of a type should be somewhat meaningful, which is critical to make the program readable.

It is not obvious how to achieve these criteria using a name generation algorithm in the context of Cogent. Simple strategies fail immediately. For instance, the current Cogent implementation generates short names such as `t1` or `t2` for types. These names are not stable, as the numeric component in the type name depends on the order in which types are processed. If the order of type definitions in a program changes, the name mapping will change accordingly. Therefore the generated names are not only uninformative, but also unstable, and a posteriori (namely, one can only work out the name assignment by running the compiler and observing the output).

Another naïve attempt is to devise a scheme for name mangling that reflects the structure of a type. For instance, a record type #{ f_1 : `U8`, f_2 : `(Bool,U32)` } can be named *urec_f1_U8_f2_tp_Bool_U32____*, where *urec* indicates that it is an unboxed record, and *tp* indicates that f_2 is a tuple. If the algorithm is very carefully designed, it is possible to make it injective. Obviously, this naming scheme is stable, a priori, and indicative of the meaning of the type. However, the generated name can be frustratingly long. In our experience, the names of many types in a file system implemented in Cogent [Amani, Hixon, et al. 2016] can be several thousand characters long.

In order to find a solution that fulfils the desired criteria, we have to deviate from the original plan of referencing Cogent types by name. Instead, programmers can reference Cogent types by directly writing Cogent concrete syntax in their C code. Using Cogent concrete syntax trivially satisfies the five properties listed above. However, it requires compiling the C program with the Cogent syntax snippets to ordinary C code. What we need, therefore, is a tool that can

```
-- Cogent
type RepParam acc obsv =
  #{ n    : U64
   , stop : (acc!, obsv!) → Bool
   , step : (acc,  obsv!) → acc
   , acc  : acc
   , obsv : obsv! }

repeat: ∀ (acc, obsv). RepParam acc obsv → acc
```

```
// antiquoted-C
$ty:acc $id:repeat ($ty:(RepParam acc obsv) arg) {
  $ty:(U64) i = 0;
  $ty:(StepParam acc obsv) a;
  a.acc = arg.acc;
  a.obsv = arg.obsv;
  for (i = 0; i < arg.n; i++) {
    $ty:(Bool) b = (($spec:((acc!, obsv!) -> Bool)) arg.stop)(a);
    if (b.boolean) { break; }
    a.acc = (($spec:((acc,  obsv!) -> acc)) arg.step)(a);
  }
  return a.acc;
}
```

Figure 4.1: An example of antiquote-C

(1) parse the C dialect that can embed COGENT syntax; (2) locate the COGENT code snippets and compile them into valid C code fragments; and (3) coalesce the compiled C fragments and the native portion of C code into a single valid C program.

The C language is quite complex and we would like to avoid the onerous work of developing a parser and a pretty-printer for it. On the other hand, it is a widely used, well-established, and relatively well-defined language. Both factors lead us to search for an existing library that can parse and pretty-print C code, with the flexibility of handling the injected COGENT snippets. Luckily, such a tool already exists: the C quasiquoting library `language-c-quote` [Mainland 2021] in HASKELL. Unsurprisingly, some extra work is needed for it to serve our purpose; after all, our use case is not what the library is designed for. Although it does not work out of the box, it does appear very promising to us, as the effort needed for repurposing the library is fairly minimal.

We call this C dialect with COGENT snippets antiquoted-C. An example of the *repeat* function

Antiquotes	Description
`$id`	For function identifiers or type identifiers when defining them
`$ty`	Refer to a Cogent type
`$exp`	Call a Cogent function; any Cogent expressions
`$spec`	Specify the type of a function callback (using typecast syntax)
`$esc`, `$escstm`	Any code that should not be preprocessed before antiquoted-C is compiled

Table 4.1: Antiquoted-C syntax: a summary

from Cogent's standard library is shown in Figure 4.1. In the antiquoted-C code, we can see that the function's identifier follows a `$id:` symbol and Cogent types are enclosed in a `$ty:()` or a `$spec:()` construct, and the rest is just vanilla C code. The quotation syntax is simply the *antiquotation* syntax in Haskell [Mainland 2007]. The fact that we use the antiquotation mechanism to include Cogent code is where the name antiquoted-C came from, as some readers may well have wondered.

The high-level idea is relatively simple. The `language-c-quote` package comes with a C parser for C quasiquoting. Additionally, it also supports a wide range of antiquotes, allowing Haskell code to be spliced into the C syntax. The key enabler of this solution is that, in the library's implementation, the content that can be enclosed by an antiquote is untyped—it is a bare string—which means that we can embed Cogent code instead of Haskell code in the C program. We will then use an FFI compiler to find all the Cogent snippets in the C code, compile them as normal Cogent program texts, and substitute them in the C syntax tree with the corresponding target C code.

As we have alluded to, programming in Cogent heavily relies on the capability of interfacing with C, and this language interface is what we will primarily discuss throughout this chapter. Aside from FFIs, the limitations of name mangling also manifest themselves in language embeddings. When the users work with the Isabelle/HOL and Haskell embeddings of Cogent programs, they are also exposed to Cogent function and type names. We will briefly discuss these embeddings in Section 4.6.4.

4.3 Antiquoted-C at a Glance

The interface between Cogent and C allows for access in both directions. Cogent supports abstract types and abstract functions, whose declarations are given in Cogent and definitions are given in C. It is essentially a means for Cogent to access C code. The other direction is also supported: Cogent exports its functions and types so that they can be accessed from the C code. In this section, we walk through the key features of the Cogent FFI with examples. Table 4.1 summarises the antiquoters, which will be explained in turn.

4.3.1 Defining COGENT Types

Defining a non-parametric abstract COGENT type is easy, because the name of the defined type is statically known and the users can refer to them directly. The trickier case is when defining parametric types. Abstract parametric types can be defined parametrically or ad hoc. The latter is similar to how associated types in a type class are instantiated in GHC/HASKELL [Chakravarty et al. 2005]. If the type is parametrically defined, then the COGENT compiler will scan for all occurrences of the type instances in the program and will only generate those being used. In antiquoted-C, we use the **`$id`** antiquoter to define COGENT abstract types. In the example below, we give definitions to two contrived parametric abstract types `R a b` and `T a b c`:

```
-- Cogent
type R a b
type T a b c

// Antiquoted-C
struct $id:(R a b) { /* ... */ };
typedef struct $id:(R a b) $id:(R a b);
typedef struct $id:(T x y z) { /* ... */ } $id:(T x y z);
```

`typedef`s can also include the **`$id`** antiquotes. One restriction on **`typedef`** is that the type synonym has to be identical to the type being defined. COGENT does not natively support recursive data types, while they are key to many applications. This has to be done with antiquoted-C. For instance, to define the linked list type:

```
-- Cogent
type List a

// Antiquoted-C
struct $id:(List a) {
   struct $id:(List a)* next;
   $ty:a val;
};
typedef struct $id:(List a) $id:(List a);
```

Inside the type definition, we can use antiquoter **`$ty`** to refer to the type parameter a.[1] **`$ty`** is arguably the most commonly used antiquoter. It can quote any COGENT types in C function definitions and type definitions. A caveat is that the quoted type must also be used in the COGENT code. This restriction makes perfect sense: after all, the role of FFI is to relate guest code with the code in the host language; an orphan antiquoted COGENT type obviously does not relate to anything meaningful in the COGENT code.

[1] In HASKELL's antiquotation syntax, when the quoted string is a single identifier starting with a lower-case letter, the parentheses around the quoted code can be omitted.

4.3.2 Defining COGENT Functions

When implementing a monomorphic abstract function in antiquoted-C, the name of the function does not concern us. The name is known, so we can simply use the function name directly. To refer to the argument and return types, we already have `$ty` from our toolbox. The more interesting case is when defining polymorphic functions. Note that, in COGENT, it is not necessary to define polymorphic functions parametrically; namely, programmers can implement instances of a polymorphic function in an ad hoc manner, similar to function overloading. During verification, they would (manually) prove that each monomorphic C instance refines the semantics of the polymorphic function. This flexibility allows programmers to choose implementation strategies that suit each particular type instantiation, and potentially achieve better performance by implementing type-specific optimisations.

To define a function parametrically, we use the `$id` antiquoter to reference the name of the function and the compiler will generate all the instances of the function that are used in a program. For example, if we have a function signature in COGENT:

```
foo : all (a, b). a → b
```

Then in antiquoted-C, we can define the function `foo` by the following:

```
$ty:b $id:foo ($ty:a arg) { /* ... */ }
```

Note that inside the `$id` antiquote, we do not include the type parameters. The compiler is able to figure out which type variables (`a` and `b` in the example above) are in scope.

To give ad hoc instantiations to the polymorphic function, type arguments should be included inside the `$id` antiquote for the function name, where the square brackets is the syntax for type application in COGENT. For example:

```
$ty:(U8) $id:(foo[(U8, U32), U8]) ($ty:((U8, U32)) arg)
{ /* ... */ }
```

The two types inside the square brackets are the type arguments supplied to the function. Accordingly, the argument type and the return type are necessarily `(U8, U32)` and `U8` respectively.[2] Once applied, the type parameters `a` and `b` are no longer in scope, and should not be referenced inside the definition of the function, otherwise an error for unknown type variables will be raised during typechecking.

4.3.3 Calling COGENT Functions

As we have seen earlier, COGENT types can be quoted with `$ty`. Another point of interaction is to invoke a COGENT function from antiquoted-C. It can be done by using the `$exp` antiquoter, which can enclose a COGENT function symbol. For polymorphic functions, the type application must be

[2]When the type itself has parentheses around them, such as tuples, the pair of parentheses for the antiquote should not be forgotten.

made explicit. In Cogent, type applications in many cases can be inferred by the typechecker. However, inside an antiquote, there is not enough context for the inference to be fully functional. Therefore explicit type applications are needed to allow the compiler to typecheck the quoted code. For example:

```
int call_foo() {
  $ty:(U32) a = 5;
  $ty:(U8) b = $exp:(foo[U32, U8])(a);
  // ...
}
```

Higher-order functions are handled differently, as Cogent compiles a higher-order function call into an invocation of a dispatch function. Imagine a Cogent function which takes two arguments f : `A` → `B` and x : `A`, and we woud like apply f to x in the function's definition. In Cogent, with no surprise, we just write a function application $f\ x$. The compiled code uses a dispatch function to perform the function call:

```
B dispatch_t3 (func_enum f, A arg) {
  switch (f):
    case FUN_ENUM_bar:
      return bar (arg);
    case FUN_ENUM_foo_2:
      return foo_2 (arg);
    ...
}
```

Depending on the input function symbol (of an enumeration type `func_enum` in the example above), the dispatch function invokes the respective function with the given argument (`arg`). The compiler will search through the program text and find all functions with the type `A` → `B` (e.g. `bar` and `foo_2`) and include them in the dispatch table. The name of the dispatch function is determined by the type of the functions that it dispatches.

To write a higher-order function call in antiquoted-C, we employ a different antiquoter than the `$exp` for first-order function calls, while mimicking the syntax of an ordinary C function application for better readability. We simply piggyback on the C typecast syntax to **spec**ify the Cogent type of the high-order function with a `$spec` antiquoter. The syntax is otherwise just a normal function call. In the following example, `arg.p1` and `arg.p2` are the first and the second projections of the argument pair `arg`.

```
B hof_ex ($ty:((A -> B, A)) arg) {
  // ...
  (($spec:(A -> B)) arg.p1) (arg.p2);
  // ...
}
```

The `$spec` type annotation ("typecast") is used to inform the FFI compiler which dispatch function to use. We choose not to let the users call the dispatch functions directly for two reasons. Firstly, the names of the dispatch functions are not known a priori, and the users would need to do some name mangling with the help of antiquoters. The more important reason is that we want to make the compilation strategy of high-order function calls fully transparent to the users, so that changes to the Cogent compiler do not affect existing antiquoted-C code.

It is necessary to have a different syntax specifically for higher-order function calls. Using the same syntax as first-order function applications is not viable. This is because, unlike first-order functions, a higher-order function object can be an arbitrary C expression, while `$exp` can only quote Cogent expressions (e.g. a function name).

4.3.4 Expressions

Besides function calls, we extend the `$exp` antiquoter to support any valid Cogent expressions. However, its expressiveness is limited, because antiquotes cannot mention any C variables or expressions. Albeit being very restrictive, it has been proved to be very convenient in constructing constant Cogent expressions, especially when the expression's type is also given by a `$ty` antiquote. This gives an extra layer of abstraction over the low-level implementation details of the Cogent compiler. In fact, in our experience, when the user strives for additional expressiveness from an `$exp` antiquote, it is usually an indication that the FFI code should be rearranged by moving the antiquoted-C function to Cogent.

4.3.5 Escape Sequences

The antiquoted-C compiler works by first preprocessing (à la `cpp`) the input antiquoted-C source files. This gives programmers more flexibility in their programming strategies and practices. For example, #-directives (e.g. `#define` and `#if`) can be included for conditional compilation and generating code from templates. After preprocessing, the antiquoted-C code is further compiled to ordinary C. There are situations where we want certain code to be hidden from the preprocessor and bypass the antiquoted-C compilation altogether. There are several typical reasons why it is desirable.

(1) When there is a solid chunk of plain C code not tinted by any antiquotes, we do not need to pass them through the FFI compiler. This can improve the overall performance of the compilation process.

(2) Even though the `language-c-quote` library, which we use to parse the antiquoted-C code, supports a wide range of `gcc` extensions [Mainland 2021], it still falls short of full syntax coverage, especially when we deal with exotic C code like the Linux kernel. We can simply hide the unsupported syntax from the antiquoted-C compiler, rather than extend the

`language-c-quote` library.

(3) Sometimes we need to preserve the preprocessor directives in the output C code, thereby the C compiler can subsequently observe them and use them accordingly. It is particularly useful to preserve `#if`-conditions in the generated C code if the code is to be shipped to downstream developers, so that they can choose their preferred compiler (e.g. `gcc` vs `clang`) and target different versions of the Linux kernel.

To meet these user requirements, our antiquotation mechanism provides two more antiquoters `$esc` and `$escstm` to **esc**ape the problematic code from being preprocessed and parsed by `language-c-quote`. The difference between them is solely syntactic—the former works on the declaration level, and the latter works on the statement level. They both preserve whatever text that is enclosed in the antiquote.

They are in one way very different from all the other antiquoters that we have introduced thus far: they quote C program texts rather than Cogent ones. So strictly speaking, they are on a tangent to the rest of the FFI story that we present in this chapter. However, remarkably, they hint towards a great potential of our antiquoted-C approach to language interoperability. It can easily scale to a multilingual setup: the quoted contents do not necessarily need to be in the same language. Imagine that we developed an experimental language Refulgent in preparation for retiring Cogent. The language might be vastly different from Cogent but it also compiled to C. We could then extend the antiquoted-C with a set of new antiquoters (e.g. `$ty_ref` and `$exp_ref` for types and expressions) for Refulgent. This would allow us to phase out the old Cogent code by incrementally replacing Cogent functions with Refulgent ones. The antiquoted-C code would house both sets of antiquotes from the two languages. The FFI compilation process would be nearly identical to how it is now, as we shall see in Section 4.5; only step (4) in Figure 4.3 would have to be done differently: two compilers would be used in parallel instead of one. Each compiler would only compile the antiquotes of its own language, and the two compilation pipelines would operate independently of each other.

4.4 The Ingredients for Antiquoted-C

In this section, we provide a brief overview on the programming techniques and tools that will be needed in compiling the Cogent-C FFI. Readers who are already familiar with Haskell's quasiquotation mechanism and the `language-c-quote` library (Section 4.4.1) and generic programming à la SYB in Haskell (Section 4.4.2) can skip the relevant subsections and fastforward directly to Section 4.5.

4.4.1 Quasiquotation

The concept of quasiquotation is rooted from mathematical logic research and later saw fruitful development in the Lisp family of languages [Bawden 1999]. It is a powerful tool primarily used in constructing language IR. It allows programmers to write literally in a language's concrete syntax to construct its abstract syntax tree. The resulting program is therefore a lot more readable, and also crucially, more relevant to the programmers. It hides the implementation details of the abstract syntax tree, which typically contains a lot of irrelevant meta-information such as line numbers, from the reasoning of the program. Antiquotation further augments the expressiveness of quasiquotation. It allows programmers to write a program template, so to speak, in the concrete syntax, leaving some holes to be filled in later, much like what a `printf` function in C would do with the `%`-modifiers.

When quasiquotation was introduced to Haskell [Mainland 2007], it was designed for easing the development of embedded languages in Haskell. For example, to write a generic function which can be used for constructing a C code snippet of a variable declaration with initialisation, one has to know about the definitions of the abstract syntax, and the code is very verbose, barely comprehensible and error-prone (e.g. the `mkFnCallAssn` function shown in Figure 4.2). In contrast, the equivalent function which uses the quasiquotation technology (`mkFnCallAssn'`) is much more readable and easy to reason about. The `[citem| ... |]` is one of the quasiquoters that the `language-c-quote` package provides for building the `BlockItem` construct in C. Within the quoted snippet, it uses several antiquotes (e.g. `$ty:(...)`, `$id:(...)`) which allow the programmer to splice the Haskell values into the abstract syntax tree at runtime. Apart from the benefit that the developers can write directly in the concrete syntax of the guest language without having to know the implementation details of the abstract syntax, antiquotation brings the parsing of the guest language to compile time, rather than being a runtime check. It guarantees that the constructed syntax tree is well-typed.

In the literature, a different variant of language quotation mechanism, such as that of Scala [Shabalin et al. 2013] and F# [F# Language Reference 2021; Syme 2006], is sometimes also referred to as quasiquotes (e.g. in [Parreaux et al. 2017], [Omar and Aldrich 2018]). These quasiquotes, however, are more comparable to Haskell's Template Haskell feature [Sheard and Peyton Jones 2002] than Haskell's quasiquotes [Mainland 2007], which extends on Template Haskell. The major difference between these notions of quasiquotes is that, Template Haskell, Scala's quasiquotes as described by Shabalin et al. [2013] and F#'s code quotation all manipulate the host language, turning a program text of the host language into its abstract syntax tree representation. Haskell's quasiquotes, on the other hand, is designed for handling languages deeply embedded in the host language. It means that the quoted language does not have to be, and in fact is rarely Haskell itself, but some embedded language in Haskell. In the context of this thesis, when we refer to quasiquotes, it is always in the sense of Haskell's quasiquotes, which operate on an embedded language rather than the host language.

More relevant to our development is the `language-c-quote` package [Mainland 2021] in Haskell. It is a library for quasiquoting C code. The library supports the C language with several groups of extensions, which include GNU C extensions. The library provides a score of

```
mkFnCallAssn :: (ToIdent a) ⇒ Type → a → Exp → [Exp] → BlockItem
mkFnCallAssn t lhs f es =
  let Type declSpec decl _ = t
  in BlockDecl
       (InitGroup declSpec []
         [Init (toIdent lhs noLoc) decl Nothing
           (Just (ExpInitializer (FnCall f es noLoc) noLoc)) [] noLoc]
        noLoc)

mkFnCallAssn' :: (ToIdent a) ⇒ Type → a → Exp → [Exp] → BlockItem
mkFnCallAssn' t lhs f es =
  [citem| $ty:t $id:lhs = $exp:f ($args:es); |]
```

Figure 4.2: HASKELL functions to construct a C initialisation statement without and with quasiquotation

quasiquoters for different C language constructs (e.g. expressions, types, and statements). More importantly, it also exports more than forty antiquoters. Although we use the quasiquotes extensively for C code generation in the COGENT compiler, for the FFI development, we do not need to quasiquote C code. What we need from the library are the C parser, the C pretty-printer, and the antiquotation support. The antiquotes are what allow us to embed COGENT code in the C template.

Table 4.2 is an excerpt of the package's documentation [Mainland 2021] for the antiquoters. The antiquotes are typed. The first part of each description says what type of C construct the antiquoter produces. For example, **$ty** produces a C type, therefore this antiquoter can only be used where a C type is expected. The second part further specifies what type of argument the antiquoter expects: for example, **$ty** expects a HASKELL term of type `Type`, which is the HASKELL type for C types.

4.4.2 Generic Programming

Generic programming à la *Scrap Your Boilerplate* (SYB) [Lämmel and Peyton Jones 2005, 2003] is a technology to allow programmers to write code to easily traverse rich mutually recursive data structures. The typical scenario where SYB should be considered is when a plethora of boilerplate code is needed solely for traversing the data structures, whereas only small individual bits in the structures are relevant to the actual computation. The SYB technology is often used in constructing and compiling embedded languages. The language's AST is typically comprised of multiple large mutually recursive data types for different language constructs (such as expressions, patterns and types), but at a time the compiler only needs to inspect and manipulate a

Antiquotes	Description
`$id`	A C identifier. The argument must be an instance of `ToIdent`.
`$ty`	A C type. The argument must have type `Type`.
`$exp`	A C expression. The argument must be an instance of `ToExp`.
`$spec`	A declaration specifier. The argument must have type `DeclSpec`.
`$esc`	An arbitrary top-level C "definition", such as an **`#include`** or a **`#define`**. The argument must have type `String`. Also: an uninterpreted, expression-level C escape hatch, which is useful for passing through macro calls. The argument must have type `String`.
`$escstm`	An uninterpreted, statement-level C escape hatch, which is useful for passing through macro calls. The argument must have type `String`.

Table 4.2: The documentation for some of the antiquoters

certain type of nodes. It is no surprise that we can also leverage the convenience that the SYB approach brings to process the antiquoted-C syntax tree.

Using SYB in HASKELL is straightforward. As an example, we sketch out how to traverse a C AST to append two underscores (`"__"`) to all C identifiers. We only need to define the computation on the `Id` type:

```
data Id = Id String Loc


append :: Id → Id
append (Id s l) = Id (s ++ "__") l
```

Apparently, the `append` function does not work on any other AST node types, but this is what SYB can automate. For instance, `Field` and `Exp` are data types which contain `Id`s, either as immediate child nodes or as deeply nested indirect sub-structures.

```
data Field = Field (Maybe Id) (Maybe Decl) (Maybe Exp) Loc
data Exp   = Var Id Loc
           | BinOp BinOp Exp Exp Loc
           | Member Exp Id Loc
           | ...

mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
```

The `mkT` (reads "make transformation") function provided by SYB automatically lifts a transformation on any type `b` to that on any allowable type `a`, so that `mkT append` can be applied to other node types in the syntax tree. When the type is `Id`, it applies the `append` function, otherwise it applies the identity function `id` to the data type. So far, we have not yet traversed any data types. We however do not need to handwrite any traversal functions ourselves. Note that the `mkT append` function is not recursive. In order to traverse, the SYB library provides a large set of built-in traversal schemes, such as the `everywhere` combinator which performs a bottom-up traversal.

```
everywhere :: (∀ a. Data a ⇒ a → a) → (∀ a. Data a ⇒ a → a)

appendDef :: Definition → Definition
appendDef = everywhere (mkT append)
```

We only need to call the traversal combinator `everywhere` on the top-level data type and it will recursively apply the appropriate transformation depending on the type of the argument. All the onerous work is hidden behind the scenes.

SYB is only one implementation (arguably the most widely known among Haskellers) of generic programming libraries and we use it as an example to illustrate how generic programming works in Haskell. These libraries can be vastly different in their structures, performance, flexibility and usability (see [Hinze et al. 2007; Rodriguez et al. 2008] for comparisons). But as far as the FFI compiler is concerned, it is not tied to any specific feature of a generic programming library. This allows us to freely choose the most appropriate one for lower engineering effort and better performance. In the actual FFI compiler's implementation, we swapped out the SYB library for the generic deriving infrastructure for Haskell [Magalhães et al. 2010] to exploit its superior performance, at the cost of a moderate increase in code size (see Section 4.6.1).

4.5 Compilation of Antiquoted-C

As introduced in Section 2.4, the Cogent compiler is comprised of several stages, from parsing, surface typechecking, desugaring, a series of core language transformations each followed by a core language typechecking phase, and finally to the C code generation and Isabelle/HOL generation along the way. For the Cogent snippets in the antiquoted-C code, we need to follow the same compilation process, so that the following *coherence* property is maintained: *The compilation result of a piece of Cogent code should be identical, regardless of whether the source code is antiquoted or is in a regular Cogent program.* Reusing the same compilation process is not only necessary, but is in fact desirable. It eliminates the need for duplicate compiler code, and guarantees coherence for free (with some caveat).

The compilation pipeline for antiquoted-C code is depicted in Figure 4.3. It consists of the following steps:

(1) Firstly, the raw input code is preprocessed by an off-the-shelf C preprocessor (e.g. `gcc`'s preprocessor), which will process the #-directives in the source code unless they are protected by the escape antiquotes (recall Section 4.3.5).
(2) The output of the last step is parsed by the C parser from `language-c-quote` (Section 4.5.1). The C parser will preserve the antiquotes in the syntax tree.
(3) The syntax tree produced from the last step is traversed to find all antiquotes (Section 4.5.2).

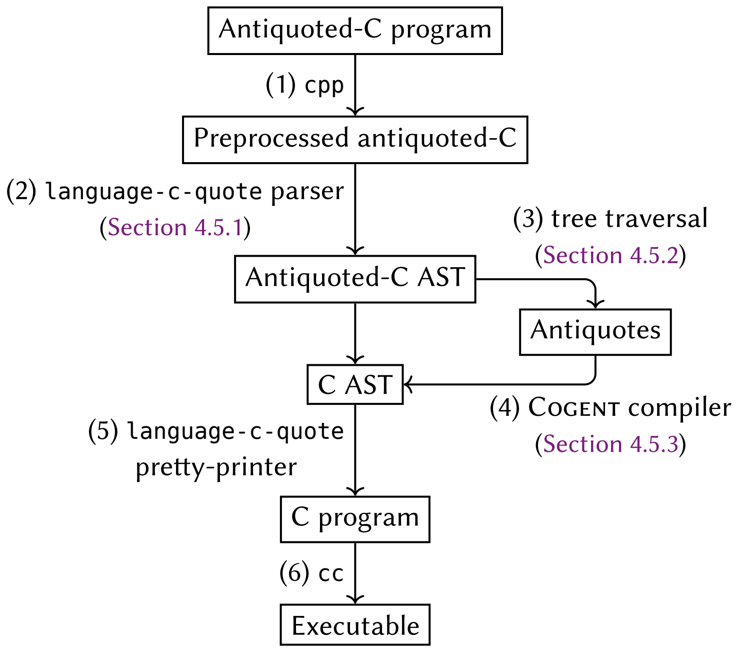

Antiquoted-C program
(1) cpp
Preprocessed antiquoted-C
(2) language-c-quote parser (Section 4.5.1)
Antiquoted-C AST
(3) tree traversal (Section 4.5.2)
Antiquotes
C AST
(4) COGENT compiler (Section 4.5.3)
(5) language-c-quote pretty-printer
C program
(6) cc
Executable

Figure 4.3: Antiquoted-C compilation pipeline

(4) For each piece of antiquoted COGENT code, it is compiled into C code using the main line of the COGENT compiler (Section 4.5.3). Each piece of the output C code will be put back into the AST to pointwise substitute for the corresponding antiquoted COGENT code.
(5) The plain C AST is then pretty-printed into ordinary C program text, using the printer from the `language-c-quote` library.
(6) Finally, the C program is to be compiled with any compatible C compiler as normal. C macros and language extensions may remain in the C code for the C compiler to process, if they were protected in escape antiquotes.

The engineering effort in the FFI compiler chiefly goes into steps (2), (3) and (4), which we will elaborate in turn.

4.5.1 Parsing Antiquoted-C

The C parser from the `language-c-quote` library can nearly be used out of the box. The antiquoters that we choose are all supported in the library, and in the case of unsupported C language extensions, we also have the tool of the escape antiquotes that will help us circumvent the limitations. The tricky part in parsing the antiquoted-C code is that the C language grammar is not *context-free* [Aho et al. 2007]. As a result, the C parser needs to keep track of the types that have been declared. The canonical example is the statement `something * odd;` in C, which is ambiguous if the compiler does not know whether `something` is a type or a variable. If it is a type, the code is declaring a pointer called `odd` to type `something`; if it is a variable, it is an expression calculating the multiplication of two variables `something` and `odd`. In light of the escape sequences introduced by **`$esc`** and **`$escstm`**, some type declarations may be absent or invisible to the C parser. To rectify this problem, the COGENT compiler requires the user to provide a list of type identifiers that the C parser should assume to exist.

4.5.2 Finding the Antiquotes

The FFI compiler not only needs to find all the antiquotes, but more crucially, needs to be able to cherry-pick the relevant antiquotes and compile them accordingly. This process is made harder because the AST traversal is not fully type-based: unlike the example that we showed earlier in Section 4.4, we cannot sieve the antiquotes by their types. For example, the **`$id`** names and for type names, but their respective compilation processes are totally different. Therefore, in the FFI compiler, we further need to determine the context in which each antiquote is used, via (sometimes deeply nested) pattern matching.

4.5.3 Reusing the Compiler

When we extract a string inside an antiquote, it can be passed to the COGENT parser. The COGENT parser is built using the monadic parser combinator library `parsec` [Leijen and Meijer 2001]. We can directly apply the relevant parser to different antiquoted COGENT constructs without having to touch the parser code, contrary to a parser constructed by a parser generator (e.g. happy [Marlow 2010a]), where additional parsers have to be specified.

Things start to be a bit hairy in the surface typechecking phase. Broadly speaking, we need to check for type well-formedness and the typing of terms. Both of them require building up some contexts. More precisely, the rules for the checks [O'Connor, Z. Chen, Rizkallah, et al. 2016] are $\Delta \vdash \tau\ \mathbf{wf}$ and $\Delta; \Gamma \vdash e : \tau$ respectively, where Δ is the kind context and Γ is the type context. When checking them in a normal COGENT program, the kind context is built up by inspecting the type signature of the function $f : \forall \overline{a_i}.\ \tau_1 \rightarrow \tau_2$, and the type context is constructed as the typing tree grows. Here in antiquoted-C code, we may not have direct access to this information. Typically, it requires us to find the type variables in the **`$id`** antiquotes and also in the corresponding COGENT declarations. For abstract type definitions, if the COGENT type declaration is $\mathsf{T}\ \overline{a_i}$ and the **`$id`** antiquote has $\mathsf{T}\ \overline{\tau_i}$ in it, then the type variables brought into the kind context will be $\mathsf{FV}(\overline{\tau_i / a_i})$, where $\mathsf{FV}(\cdot)$ returns a list of free type variables, and $\cdot/\cdot$ denotes a substitution. Similarly for abstract functions, if the type signature in COGENT is $f : \forall \overline{a_i}.\ \tau_1 \rightarrow \tau_2$ and the **`$id`** antiquoter carries $f[\overline{\tau_j}]$ ($j \leq i$, as COGENT allows for partial type applications), then the kind context will include $\mathsf{FV}(\overline{\tau_j / a_i})$ (for a_k where $j < k \leq i$, a_k will remain intact). The type context is always empty in an antiquoted-C function definition, and consequently only constant COGENT expressions can be antiquoted.

Besides the two contexts in the surface typechecking phase, each compilation stage uses several states global to the whole program. These states are typically constructed as the compilation progresses. To ensure coherence when compiling antiquotes, we want to perform the compilation under the same states as in the normal Cogent compilation. For instance, the C code generation phase keeps a mapping between Cogent types and their C type names. Obviously this information will be needed when compiling the **`$ty`** antiquotes. What we do in the compiler is that, after each stage of the compilation, the final state is cached, and the cached states will then be fed to the respective stage of the antiquoted-C compilation.

4.6 Discussion

The Cogent FFI has been used extensively over several years for developing the Cogent standard library and systems code of various complexity, from simple device drivers to real-world file systems, including the file systems reported in [Amani, Hixon, et al. 2016], which initially used some ad hoc Python scripts for name mangling or reverse engineering the name generation. The application of antiquoted-C has been exemplified in Figure 4.1 by the Cogent library function `repeat`, which performs general-purpose **`for`**-loops. Figure 4.4 presents two word array functions[3] from the library. No fundamental differences in the usage of antiquoted-C can be observed between library code and systems implementation code.

In this section, we evaluate the FFI from performance, engineering effort and user experience aspects. In particular, we discuss improvements that have been made and can be made for better performance, especially compile-time performance. Then we outline how our approach to FFI design can also be applied to other parts of Cogent, and more broadly, to other programming languages.

4.6.1 Performance

The performance that concerns us primarily is the compiler's runtime performance, namely how fast antiquoted-C code can be processed by the FFI compiler. The memory footprint is not a major bottleneck when the development happens on modern computers.

Having seen the compilation process, astute readers may have realised that the antiquoted-C compilation is rather cumbersome, despite the lightweight engineering effort. It encompasses loading multiple massive internal states from the Cogent compilation pipeline into the antiquotes compilation, parsing antiquoted-C code, traversing the antiquoted-C AST and processing each antiquote. From our experiments, it turned out, and to some extent surprisingly, that the SYB tree traversal was in fact the biggest performance bottleneck among the tasks, by a long way. We tackled this problem by switching out SYB for another traversal library.

In fact, various performance studies in generic programming libraries have noted that the SYB library is on the slow end of the spectrum [Adams and DuBuisson 2012; Brown and Sampson 2009; N. Mitchell and Runciman 2007; Rodriguez et al. 2008] despite its popularity among

[3]We used the `wordarray_length` function in Section 2.2; we will discuss the `wordarray_set` function more in Section 5.5.

```
-- Cogent interface
type WordArray a

{-# cinline wordarray_length #-}
wordarray_length: ∀(a :< DSE). (WordArray a)! → U32

type WordArraySetP a = (WordArray a, U32, U32, a)
wordarray_set: ∀(a :< DSE). WordArraySetP a → WordArray a
```

```
// Antiquoted-C
struct $id:(WordArray a) {
  int len;
  $ty:a* values;
};

u32 $id:wordarray_length($ty:((WordArray a)!) array)
{ return array->len; }

$ty:(WordArray a) $id:wordarray_set($ty:(WordArraySetP a) args)
{
  /*
   * args.p1 : array
   * args.p2 : start position
   * args.p3 : length from start position
   * args.p4 : element to set
   */
  $ty:(U32) start = args.p2;
  $ty:(U32) len = args.p3;

  if (start > args.p1->len)
    return args.p1;

  if (start + len > args.p1->len)
    len = args.p1->len - start;

  memset(args.p1->values + start, args.p4, len);
  return args.p1;
}
```

Figure 4.4: An excerpt of word array code from the standard library

Haskell programmers. We refer the interested readers to the excellent lecture notes on generic programming libraries by Jeuring et al. [2008] for more details.

In the antiquoted-C compiler, we choose to use the generic deriving approach [Magalhães et al. 2010] in lieu of the SYB library as an optimisation. Even though we have to manually declare the C syntax constructs to be instances of the `Generic` class, which converts C ASTs to and from the generic representation, and also the instances of the generic function that processes the C syntax, the performance gain has been tremendous. This optimisation brings the compilation time of the entire BilbyFs file system [Amani 2016], which comprises of around 4k lines of antiquoted-C code, from 142 seconds (s) down to under 33s. A breakdown shows that, around 30s is used in the main compilation pipeline, meaning that the FFI compilation time is reduced from 112s to 3s. The test is performed on a laptop with AMD Ryzen 7 4700U CPU and 16 GiB of RAM. Similar speed-up can be obversed on other platforms.

Other aspects of the FFI compiler can also be optimised to further boost the performance. We list several potential improvements that we have considered, and analyse their effectiveness and engineering costs.

Escape sequences The fact that we can bypass the C parsing via the `$esc` and its sister hints to us that, we can use similar tricks to reduce the amount of code to be parsed, and to potentially improve runtime performance. This however puts the burden on the users to include more escape sequences.

Custom parser One possibility is that, instead of using an off-the-shell parser, we build the antiquoted-C parser in-house. Writing a parser for a C dialect is by no means easy, but it is a trade-off that can be made if more flexibility is in demand. For example, we can define our own antiquoters so that they are all distinct. It then will eliminate the need for deeply-nested pattern matches to disambiguate the antiquoters. Similarly, we can define the C syntax in a way that each antiquote has a different type. This will allow for a type-based traversal. Alas, these methods turn out to be not very effective. The reason is that, the compilation of the antiquoted Cogent code snippets depends on the program context: as we have seen in Section 4.5.3, antiquotes are not independent of each other. We still need to traverse the AST as we do now in order to relate the antiquotes to ensure that the contexts are reconstructed appropriately. Defining a streamlined C AST in the style of CIL [Necula et al. 2002] may also help simplify tree traversal and analysis.

Partial parser We believe it is possible for the antiquoted-C parser to analyse the program texts at a coarser granularity. One observation is that, we are in fact not very interested in the tree structure of the syntax. It is almost adequate to *tokenise* the program text, with one exception of the `$id` antiquoter. As discussed above, if we write our own antiquoted-C parser and introduce distinct antiquoters for different language constructs, it is sufficient to sequentially process each element in the list of tokens, and update the kind context on the fly as we encounter the antiquotes that introduce type variables (the equivalent of the current `$id`). This approach should save the effort in analysing the syntax, and also eliminate the need for the user-provided list of declared C type names, as we do not construct the syntax tree. On the down side, fewer sanity checks can be performed due to the absence of the tree structure. Moreover, pretty-printing the code will become trickier for the same reason.

4.6.2 Engineering Effort

The Haskell code for the entire FFI compiler is around 550 LoC, which also includes around 220 LoC of auxiliary code for importing modules, defining datatypes for storing the states, handling the FFI compiler's command-line options, parsing input files for the assumed type declarations, etc. With the Haskell generic optimisation described in Section 4.6.1, the code size grows by around 130 LoC. Putting the numbers into perspective, the Cogent compiler consists of more than 22k LoC, half of which is directly reused by the FFI compiler.

From the Haskell package dependency point of view, the use of `language-c-quote` and `syb` does not add much overhead to the dependency footprint. Although `language-c-quote` itself has a relatively large dependency footprint, it is already used by the Cogent compiler for C code generation. The SYB library is an additional dependency, but it only depends on the Haskell compiler's `base` library [GHC base 2022], which is shipped with the compiler. The generic deriving mechanism that replaces SYB is even better in this regard: it is integrated with the `base` library.

4.6.3 User Experience

As a research language, Cogent is primarily only used within the research group and among research partners, and the user base is admittedly very small (a couple of dozens). The users, however, range widely in knowledge level and research background, from undergraduate students to professors, from systems engineers to formal methods experts. This allows us to grasp a better picture when we study the user experience of the FFI.

The feedback from the users has been broadly positive. They report no issues in understanding the antiquotation syntax, and the usage of each of them. The abstraction that the FFI provides over the compiler internals also alleviates the steep learning curve of Cogent. The main complaint about the FFI is that when the antiquoted-C file has errors, the compiler does not always produce informative error messages. This is because very often, the error will be caught and reported by the `language-c-quote` C parser. A common mistake in writing antiquoted-C is missing parentheses when the antiquoted type is a tuple. Another typical situation is that the user antiquotes the wrong Cogent type. In this case, the FFI compiler will readily accept the program but the downstream C compiler will complain. The user is then forced to hunt for the bug in the potentially long-winded preprocessed monolithic C file. This however is no worse than FFIs with name mangling—if a wrong name is used, the error can only be caught during linking.

As we have argued in Section 4.2, assigning names to all types at the language boundary is not viable in general. However, if there are only a very small number of types to be exported, this approach is lightweight and easy. Cogent does support user-designated type names, and if this approach is chosen, the FFI compilation pipeline can be left out completely. In a more general situation where antiquoted-C is required, transitioning from plain C to antiquoted-C is relatively straightforward, and the infectiousness of antiquoted-C ought not be a major concern. The user does not have to draw a firm line between vanilla C and antiquoted-C, if working within the parameters set by the C parser (Section 4.5.1). It is usually harmless to include plain C code in an antiquoted-C file. When vanilla C code is routed via the antiquoted-C compilation pipeline, it inevitably puts more load on the Cogent compiler and affects the compile-time performance, but the performance overhead is usually minor. It should be noted though, that the structure of the build scripts (e.g. Makefiles) may need to be adjusted accordingly. This is because the Cogent compiler behaves differently when antiquoted-C compilation is engaged. To compensate for the different behaviour, the user needs to organise the build steps correspondingly. These changes are only required when switching between the two different methods of interfacing with C code, hence not a routine in using antiquoted-C.

4.6.4 Other Target Languages

The C language is not the only target language that the Cogent compiler generates. It also produces Isabelle/HOL and Haskell embeddings (recall Chapter 2 and also see Chapter 5). When the user manually proves the refinement between the functional correctness specification and the Isabelle/HOL shallow embedding (see Figure 2.1), they will necessarily need to know the names of the Cogent types and functions in the embeddings. These names are unstable, meaningless, and are not known a priori.

Alas, the antiquoted-C idea cannot be directly applied to Isabelle/HOL for several reasons. Firstly, to the best of our knowledge, there is no general-purpose Isabelle/HOL parser library in Haskell that readily supports antiquotes. Isabelle is known to be difficult to parse in general, due to its support for user-defined inner syntax [Wenzel 2021, Chapter 8]. Secondly, because of the interactive nature of Isabelle, it is not practical to add a hypothetical antiquoted-Isabelle compiler into the development loop. In particular, once the Isabelle/HOL script is updated, the theorem prover will reprocess the entire session, totally jeopardising the interactive theorem proving experience. Thankfully, this is a less pressing issue than that of the C names. Unlike writing C code to interface with Cogent, the manual Isabelle/HOL proofs (recall Figure 2.1) are only possible to be developed when the embeddings have been produced. Therefore, the type and function names in the Isabelle/HOL embeddings do not necessarily need to be known a priori. Also, proof engineers oftentimes only attempt the proof when the Cogent code is relatively mature and stable, hence more stable Isabelle/HOL embeddings to work with. Although the generated names are less intelligible, they normally only need to work out the meaning of these names once. Dictating the names of the exported types partly helps improve readability.

Occasionally, the COGENT code is patched and the manual proofs need to be revised. It would be more convenient if parts of the manual proofs that are not directly affected by the code change could be recycled on these occasions. We implemented a feature in the compiler to retain name stability in Isabelle/HOL. When compiling COGENT code, the compiler's internal states are saved in a binary file called a *name cache*. The cache can be loaded in subsequent compilations, so that the names of all existing types and functions are consistent with earlier runs of the compiler. Proof engineers only need to spend their mental energy on deciphering the names of new types and functions.

Splicing COGENT code snippets into the HASKELL embedding is easier to achieve and is also more useful. Instead of using antiquotation, it can be implemented directly using HASKELL's quasiquotation mechanism [Mainland 2007], which then invokes the relevant COGENT compiler modules to generate HASKELL embedding for the COGENT code snippets.

4.6.5 A Recipe for Antiquoted FFI

We envision that the antiquotation approach to FFI design is applicable to other languages as well, when the name-based FFI is not desirable or infeasible to implement. We dissect the recipe for building such an antiquoted FFI.

Firstly, the host language's compilation target needs to coincide with the guest language, so that the compilation of the quoted host language and that of the guest language are independent of each other. This condition can be met in many real-world scenarios. Depending on the application domain, there are many languages that serve as common backend languages. Some examples include, C/C++ in systems programming, C# and JavaScript in web development, the Common Intermediate Language [ISO 2012, Partition III] (CIL) for Microsoft's .NET framework, and SQL in databases.

Secondly, the guest language has a readily available parser that supports foreign code splices. It is very likely that a parser already exists for the guest language, if it is a stable and well received language. Language features similar to HASKELL's quasiquotation have seen their appearance in other languages (Section 4.4.1). Augmenting an existing parser with antiquotation support requires some effort, but it can be amortised in the long term if the augmented parser is to be shared with the entire community for mutual benefits. Moreover, some programming languages already provide libraries for extending the syntax, for example camlp5 [INRIA 2017] for OCaml.

Thirdly, an AST traversal mechanism is available, and preferably with automation to save the users from writing boilerplate code. This is typically the case in modern mainstream programming languages.

Lastly, the compiler for the host language should be constructed in a modular fashion, so that the relevant modules can be reused in the FFI compiler. This is also in line with good software engineering practices.

This recipe offers an alternative approach to FFI design that does not rely on name-mangling. It only consists of a few ingredients, none of which is rare to find in practice. This recipe is not limited to FFIs; similar ideas can be applied to generate language embeddings (cf. Section 4.6.4).

4.7 Related Work

The work on FFIs is vastly abundant and we can by no means exhaustively survey them. However, to our knowledge, FFIs predominantly rely on name mangling to cross the language boundary. Even for strongly-typed languages with structural typing support (e.g. Ballerina [Ballerina 2022], Go [Go 2022]), their type systems often have some nominal aspects (or coexist with a full-blown nominal type system). Also, due to the underlying memory layout and runtime, they seldom allow structures to be exported directly.

The idea of splicing one language into another is not new. For example, LuCa [Tanimura and Iwasaki 2016] is a C language extension that allows Lua code to be embedded in C. LuCa is designed to overcome the difficulties in using the C API of Lua. In contrast, antiquoted-C gives an alternative to name mangling in FFIs. LuCa is arguably more flexible, in the sense that it allows the quoted Lua code in LuCa to backquote C expressions, while antiquoted-C does not. The LuCa compiler shares some commonality in the overall structure with the antiquoted-C compiler. But unlike the antiquoted-C compiler, which repurposes the `language-c-quote` parser and reuses the main Cogent compiler modules, the LuCa compiler has a dedicated parser, and performs semantic analysis on the LuCa AST. The LuCa-generated API calls have some non-negligible performance overhead.

Jeannie [Hirzel and Grimm 2007] can be considered an FFI language between Java and C. In Jeannie, users can nest Java and C code in each other. Jeannie programs can be compiled to JNI code. The objective of Jeannie is on language composition: Jeannie users can access features from both languages, with reduced overhead in writing FFI and resource management code. The compiler essentially consists of two independent compilation pipelines for the two languages, while sharing some common infrastructures. Because it has knowledge about both languages, it is capable of performing more checks on the programs, especially eliminating errors across the language boundary. The Jeannie compiler heavily relies on the visitor design pattern for AST traversal, and uses a catch-all visit method for the AST nodes that do not need to be processed. This is comparable to the generic programming techniques used in our FFI compiler.

The quasiquotation mechanism in Haskell is also used for accessing foreign languages. For example, `language-c-inline` [Chakravarty 2014] allows for inlined C and Objective-C code in Haskell, offering programmers an alternative to writing bindings for foreign functions.

4.8 Conclusion

In this chapter, we presented a novel application of HASKELL's antiquotation mechanism for building an FFI between the structurally typed, polymorphic functional language COGENT and C. The FFI language is a dialect of C, in which users can refer to COGENT types and expressions using the concrete COGENT syntax, rather than relying on exported (potentially mangled) names of the program objects. It offers language developers an alternative to name mangling in language interoperation. The FFI language, antiquoted-C, is intuitive to understand and easy to use, and its compiler only requires minimal engineering effort to develop. We envisioned how this approach to FFI design could be adopted by other languages and gave the recipe for its development.

Chapter 5

Property-Based Testing for Cogent

This chapter is derived from the following publications:

- ⋄ Zilin Chen, Liam O'Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. Oct. 28, 2017. "The Cogent Case for Property-Based Testing." In: *Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, Shanghai, China, 1–7. ISBN: 9781450351539. DOI: https://doi.org/10.1145/3144555.3144556

- ⋄ Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. Dec. 2022a. "Property-Based Testing: Climbing the Stairway to Verification." In: *ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2022). ACM, Auckland, New Zealand, 14 pages. DOI: 10.1145/3567512.3567520

The author of this thesis is the primary contributor to the technical development of the entire property-based testing framework, except for the implementation of the prototype domain-specific language described in Section 5.5.

Property-based testing (PBT) is a powerful tool that is widely available in many modern programming languages. It has been used to reduce the effort required for formal software verification. We demonstrate how PBT can be used in conjunction with formal verification to incrementally gain greater assurance in code correctness by integrating PBT into the verification framework of Cogent—a programming language equipped with a certifying compiler for developing high-assurance systems components. Specifically, for PBT and formal verification to work in tandem, we structure the tests in a fashion that mirrors the refinement infrastructure that is often used in formal verification: the behaviour of the system under test is modelled by a functional correctness specification, which mimics that of the formal proof, and we test the refinement relation between the implementation and the specification. We exhibit the additional benefits that this mutualism brings to developers and demonstrate the techniques we used in this style of PBT, by studying two concrete examples.

5.1 Introduction

Property-based testing (PBT), in the style of QuickCheck [Claessen and Hughes 2000], is a popular testing methodology and has tool support in many modern programming languages [MacIver 2016b]. In PBT, tests are specified in the form of logical properties which are automatically executed on randomly-generated inputs to find counter-examples. PBT is not only useful in finding bugs in programs, it has also been leveraged to reduce the effort in formal verification [Bulwahn 2012; Dybjer et al. 2003; Hriţcu et al. 2013; Lampropoulos and Pierce 2018]. Subjecting code to extensive PBT *prior* to verification reduces the number of defects and specification inconsistencies, thus reducing verification cost. A proof engineer can first test a property, and will only attempt to prove it after having gained reasonable confidence in its validity.

In program verification, it is common practice to prove the correctness of a program against a formal specification. The specification can be given in various forms (e.g. state machines, process calculi, modal logics), depending on the specific application domain. To show that the implementation conforms to the specification, the notion of *refinement* [R. J. R. Back 1988; Morgan 1990; Roever and Engelhardt 1998] is frequently used to establish the formal connection.

In this work, we explore the combination of PBT and refinement-based formal verification. We borrow from verification the *functional correctness specification* that is used to dictate the behaviour of the system in question, and give it to PBT. Instead of testing logical properties about the system, which is what PBT is typically designed for, we test the refinement relation between the implementation and the specification. Using logical properties to describe the behaviour of systems has been criticised for its practicality [Koopman, Achten, et al. 2012], especially if the full functional correctness of the system is desired. The high-level properties of a system can instead be proved on top of its functional specification.

We introduce PBT to our development loop, in parallel with the refinement-based verification framework. Specifically, we formulate the refinement between the implementation and the functional specification as the property to be tested, which is an under-explored application of PBT. Employing PBT in the formal verification context brings additional benefits beyond detecting bugs in the implementation of the systems.

In contrast to the high-effort all-or-nothing of a full functional correctness proof, PBT provides a continuum ranging from no assurance (no tests), to some assurance (good test coverage), to better assurance (some properties proved, some tested), all the way to high assurance (all properties proved). This allows users to make trade-offs between cost and assurance according to the criticality of a component.

Tests are more immune to program evolution than formal proofs. A proof may require significant changes whenever the code changes, even in scenarios where the specification remains the same (e.g. when an algorithm is optimised). On the contrary, PBT only requires developer input when the specification changes. Therefore, PBT can provide quick feedback on the likely correctness of the change, reducing code maintenance cost. Furthermore, all proofs depend on assumptions such as the correctness of the hardware or the external software involved. Some of these assumptions can be tested to increase our confidence in the correctness of the overall system.

It is usually not feasible to verify software that was not designed for verification, as the code has to be designed in a modular fashion, around clearly stated correctness properties. This means that it is vital to have an effective means for developers to express their design requirements, to experiment with and evaluate the designs, and to have a good set of design guidelines [Breitner et al. 2018] so that the programs they write are easy to specify and verify.

In large-scale software verification projects, such as seL4 [Klein, Elphinstone, et al. 2009], the systems experts and the verification experts are typically two separate teams. We posit PBT will enhance the communication between these two groups. We propose to use PBT specifications for this purpose. While the properties are similar to formal specifications, they represent tests and, as such, feel more familiar to software engineers than abstract proof requirements. Since PBT gives almost immediate benefit to software engineers, there is an incentive for them to design their code such that these properties are meaningful and easy to express, thereby structuring their code for formal specification, making it amenable to verification.

We examine these benefits by integrating PBT in the COGENT framework. The BilbyFs file system [Amani 2016], developed in COGENT, provides an example of this effect. The entire BilbyFs has been formally specified but only partially verified. By applying PBT, we uncovered bugs in the specification and the implementation of BilbyFs. PBT has therefore already reduced the cost of verifying the remainder of the system by uncovering mistakes early on.

To summarise, we make the following contributions:

- We demonstrate how to integrate PBT into a refinement verification framework by using COGENT as the target platform. Unlike previous use of PBT, we test against specifications that are defined as refinement properties (Section 5.3).
- We argue why PBT is suitable to be employed in parallel with formal verification, and explain the important role that PBT plays in the design and implementation of the systems in question (Section 5.4).
- We provide two concrete examples from the testing of components of the BilbyFs file system to demonstrate techniques that we used for specifying refinement relations, modularising the tests, using mocks, handling non-determinism, and efficiently generating test data (Section 5.5 and Section 5.6).
- We discuss the engineering implications of our approach and lessons learnt and proposals resulting from them (Section 5.7).

The full development can be found in [Z. Chen, Rizkallah, O'Connor, et al. 2022b] as a virtual machine image. It is a snapshot of the main COGENT project repository [The COGENT team 2023], and was submitted as the supplementary material to the SLE'22 paper [Z. Chen, Rizkallah, O'Connor, et al. 2022a], on which this chapter is based.

5.2 Background

5.2.1 Property-Based Testing and QuickCheck

PBT is a quick and effective method for detecting bugs and finding inconsistencies in specifications [Hughes 2016]. Similar to formal verification, PBT uses logical predicates to specify the desired behaviour of functions, by defining the allowed relations between inputs and outputs of the functions. It evaluates the properties on a large set of automatically generated input values in search for counter-examples.

While PBT is effective, it is not universally applicable. In practice, it is often hard to completely describe the behaviour of a system solely in terms of logical properties [Koopman, Achten, et al. 2012], albeit being a very useful method in certain contexts (e.g. as shown by Ridge et al. [2015]). In the context of formal verification, however, using a functional specification to describe the behaviour of a systems is very common. Thus proof engineers can first run extensive tests on the conjectures before attempting any proof development. This technique is not new and has witnessed great success in the verification community [Berghofer and Nipkow 2004].

QuickCheck [Claessen and Hughes 2000] is a combinator library in Haskell for PBT. While the QuickCheck functionality is now available in many programming languages [MacIver 2016b] and theorem provers [Bulwahn 2012; Dybjer et al. 2003; Lampropoulos and Pierce 2018], we interface Cogent to the Haskell QuickCheck library, as it is mature, feature-rich and integrates well with Cogent and C.

5.2.2 Data Refinement

Prior verification work in Cogent, e.g. of BilbyFs [Amani 2016], connects the functional specification to the Cogent implementation, and the Cogent implementation to the compiled C code [Rizkallah et al. 2016] by proving *refinement relations*. The notation of refinement is also central to our testing framework, in which they are expressed as QuickCheck properties. We use a textbook definition of refinement [Roever and Engelhardt 1998]. Informally, a program C is a *refinement* of a program A if every possible behaviour in the model of C is observable in that of A.

In an imperative setting, a simple model for both the abstract specification and the concrete implementation would be relations on states, describing every possible behaviour of the program as the manipulation of some global state. This means that if we prove a property about every execution for our abstract specification, we know that the property holds for all executions of our concrete implementation.

This state-based model for specifying the behaviours of systems is a very common paradigm in the world of model-based testing (MBT) [Gurbuz and Tekinerdogan 2018; Koopman, Achten, et al. 2012; Utting et al. 2012]. However, as mentioned in Section 5.1, COGENT's purely functional semantics provides a simple formal model of a program's behaviour; specifically, it enables reasoning about programs using *equational* principles. The equational semantics is fortunately widely available in PBT libraries, including QuickCheck.

Since COGENT is a purely functional, deterministic, total language, there is no global state, and all functions are modelled as plain mathematical functions. In such a scenario, the only state involved consists of the inputs and outputs to the function, simplifying the refinement statement. Given an abstract function abs $::\ X_a \rightarrow Y_a$, and a concrete COGENT function conc $::\ X_c \rightarrow Y_c$, then, assuming the existence of refinement relations R_X and R_Y, we can express the statement that conc refines abs as:

$$R_X\ i_a\ i_c \implies R_Y\ (\text{abs}\ i_a)\ (\text{conc}\ i_c)$$

This, however, places unnecessary constraints on our abstract specification. While COGENT is deterministic and total, our abstract specification need not be. In fact, it is often desirable to allow non-determinism to reduce the complexity of the abstract specification. In the context of testing, we are required to restrain the degree of non-determinism in the specification, for the sake of efficient execution of the test script. This, however, does not preclude us from having a non-deterministic specification.

We model non-determinism by allowing abstract functions to return a *set* of possible results. Then, our refinement statement merely requires the single concrete result to correspond to one of the possible abstract results:

$$R_X\ i_a\ i_c \implies \exists o_a \in \text{abs}\ i_a.\ R_Y\ o_a\ (\text{conc}\ i_c)$$

Defining a syntactic sugar for the *corres*pondence relation between the outputs:

$$\textit{corres}\ R\ a\ c \overset{\text{def}}{=} \exists o \in a.\ R\ o\ c$$

the refinement statement can be formulated as:

$$R_X\ i_a\ i_c \implies \textit{corres}\ R_Y\ (\text{abs}\ i_a)\ (\text{conc}\ i_c)$$

Theorems that capture correctness for COGENT systems typically have this *corres* format. We therefore aim to encode these as machine-testable properties in HASKELL.

5.3 The Cogent QuickCheck Framework

The integration of PBT into the COGENT framework mirrors the verification tasks, as shown in Figure 5.1. The developer manually writes a *HASKELL executable specification*, which plays a similar role to the Isabelle/HOL functional correctness specification. The compiler now generates

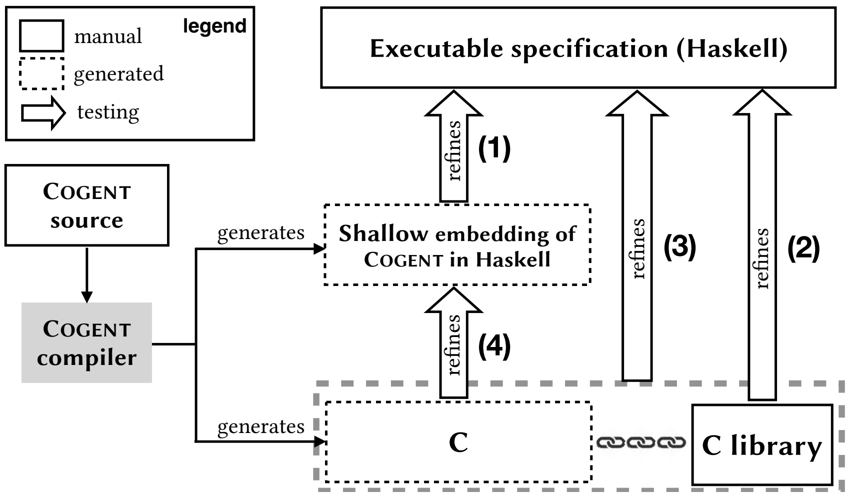


Figure 5.1: An overview of the COGENT QuickCheck framework

a *HASKELL shallow embedding* of the COGENT code for PBT. Although not formally connected, the HASKELL and Isabelle/HOL embeddings are very similar.

The framework supports testing the C implementation of ADTs against manually-written HASKELL, shown as arrow (2) in Figure 5.1; Section 5.5 provides an example. It furthermore supports testing the generated HASKELL embedding of the COGENT program, which is combined with the HASKELL definition of any ADTs that the COGENT program uses, against a manually written, more abstract HASKELL executable specification. This is depicted as arrow (1) in Figure 5.1 (the included ADTs are not shown in the figure); Section 5.6 provides a case-study. The ADTs to be invoked by the HASKELL shallow embedding can either be abstract or concrete, depending on whether the ADTs are also considered the system under test.

The refinement relation between the C code and the HASKELL embedding of the COGENT program (arrow (4)) can also be tested, although it does not concern us as much, since it is certified by the automatic proof. One scenario where this test can be carried out is during the development of the COGENT compiler, before the automatic refinement proof pipeline is fully restored.

The complete compiled executable, depicted in Figure 5.1 as the grey dotted box at the bottom, can be tested against the executable specification in theory, as indicated by arrow (3). However, we typically do not perform this test using the QuickCheck framework in COGENT, as the gap between their state spaces is usually too large to handle effectively. The final system can normally be deployed in the production environment and be tested against third-party test suites for their specific application domain. In the context of the BilbyFs, for example, there exists tools such as `fstest` [Behlendorf 2011].

The top-level Isabelle/HOL specification, written in higher-order logic, is not executable and may be highly non-deterministic in order to model externalities. For example, the memory allocator may non-deterministically fail to model out-of-memory errors, and disk device drivers may non-deterministically fail to model hardware errors.

The HASKELL specification must also model this non-determinism, but is more restrictive than the Isabelle/HOL specification, as it must be *executable*. Simulating a non-deterministic model can be exponential in time and space. To allow modelling a minimal amount of non-determinism in the HASKELL specification, the tester has to ensure that the search domain is finite and reasonably small by carefully examining the needed quantifiers in the specification.

For example, in the Isabelle/HOL abstract specification of BilbyFs, the `afs_get_current_time` function is defined as follows:

```
definition
  afs_get_current_time :: afs_state ⇒ (afs_state × TimeT) cogent_monad
where
  afs_get_current_time afs ≡ do
    time′ ← return (a_current_time afs);
    time ← select {x. x ≥ time′};
    return (afs⦇ a_current_time := time ⦈, time′)
   od
```

It picks non-deterministically a `time`, which is no earlier than the time stored in the file system state `afs`. This abstract specification is not suitable for testing, due to the infinitely large set of values. In contrast, on line 13 of Figure 5.3b, the specification non-deterministically chooses an error code from a small set of `{eIO, eNoMem, eInval, eBadF, eNoEnt}`. This moderate state explosion can potentially be handled by the testing framework, depending on the context in which the `afs_readpage` function is applied.

For this reason, the HASKELL executable specification is often strictly less abstract than the Isabelle/HOL functional specification. Currently, the two specifications are not formally connected, but they should bear a strong resemblance to each other and can be checked by manual inspection. It is not necessary to formally connect the two, as the gap only affects the quality of the tests. For instance, we cannot systematically evaluate the quality of tests (e.g. test coverage) with respect to the formal Isabelle/HOL specification, even though such tools may be readily available from the HASKELL QuickCheck library. Ideally, it would be convenient if one specification could be generated from the other one. Automatic refinement mechanisms that allow verified generation of HASKELL from Isabelle/HOL have been explored by Lammich [2013] and Lammich and Lochbihler [2018]. Generating the HASKELL specification from the Isabelle/HOL abstract specification is undoubtedly handy, when the Isabelle/HOL specification is developed prior to the HASKELL executable specification. This however is not always the case, and in fact in the workflow that we proposed, the HASKELL specification is used to guide the formulation of the Isabelle/HOL one.

5.4 PBT and Systems Design Go Hand-in-Hand

We argue that the employment of PBT and the design of the systems are interlinked with each other: appropriate systems design assures the effectiveness of testing, and the use of PBT encourages the programmers to design their systems in a fashion that is amenable to formal verification.

As a purely functional language, Cogent is in general well suited for PBT and verification: function values only depend on input values, with no hidden state. However, system state must be explicitly threaded through as an input and an output. If the functions are not designed appropriately, it is possible that a PBT test suite hardly yields counter-examples. Systems code often involves pretty large global states. However, a function typically only accesses a small fraction of the state. If the entire state is passed in, any random variations to the parts of the state that are not accessed by the function will have no effect on the behaviour of the function. In this case a large portion of the randomly generated test cases in PBT will not actually test anything and result in poor test coverage.

In practice, this means that to be suitable for PBT, the functions must be designed to keep the inputs minimal and relevant, which is unlikely when simply translating existing C code with global state into Cogent. While this may seem like a high price to pay, a verification-friendly design has the same requirement for modularity and compartmentalisation [Amani 2016; Amani and T. Murray 2015]. Thus PBT imposes few restrictions beyond existing requirements of verification, and instead helps guide the system design.

In the context of Cogent, the systems that the developers implement typically do not have a formal specification to start with. Systems programmers, together with verification engineers, not only need to ensure that they are implementing the systems right, but also to ensure that they are implementing the right systems. PBT provides guidance to them on how to structure the specification and the implementation. Having good design decisions, such as keeping the states threaded through small and relevant as we mentioned above, is doubly rewarding. It simplifies the manually defined refinement relations that correlate the concrete state with the abstract one, both in PBT and in verification.

Expressing design requirements for verification is an ongoing challenge, and we observed in the past that when verifying real-world systems, it is difficult for proof engineers to communicate these requirements effectively to the software engineers. The seL4 project [Klein, Elphinstone, et al. 2009] overcame this problem by using an executable Haskell specification of the system as an interface between these two groups [Klein, Andronick, Elphinstone, et al. 2014]. We posit that QuickCheck properties is a highly suitable language for communicating design requirements. They readily translate to formal specifications, but are expressed in a programming language, and thus familiar to software developers. As they lead to effective test generation, software developers get immediate benefit from using these specifications to structure their code to maximize their use, which consequently makes the code easier to verify.

```
prop_corres_wordarray_set_u8 :: Property
prop_corres_wordarray_set_u8 = monadicIO $
  forAllM gen_wordarray_set_u8_arg $ \args → run $ do
    let ia = mk_hs_wordarray_set_u8_arg args
        oa = uncurry4 hs_wordarray_set ia
    ic ← mk_c_wordarray_set_u8_arg args
    oc ← cogent_wordarray_set_u8 ic
    corresM' rel_wordarray_u8 oa oc
```

Figure 5.2: The refinement statement for `wordarray_set` (deallocation is omitted for simplicity).

5.5 Example: Testing Refinement of an ADT

We apply the Cogent QuickCheck framework to the `WordArray` library, which implements common functions manipulating arrays of machine words and is shared by all of our systems implementations. Most of these `WordArray` functions are implemented in C, and are invoked via the FFI mechanism available in Cogent.

We want to test whether each function observes the refinement property from Section 5.2.2. For example, the behaviour of the ADT function `wordarray_set`[1] (similar to `memset` in C)—which fills the first `n` elements starting at a certain index `frm` into a word array `arr` with a constant value `a`—is manually specified in Haskell as follows:

```
-- Haskell spec.
type WordArray a = [a]
hs_wordarray_set :: WordArray a → Word32 → Word32 → a → WordArray a
hs_wordarray_set arr frm n a =
  let len = length arr in
   if | frm > len = arr
      | frm + n > len = take frm arr ++ replicate (len - frm) a
      | otherwise = take frm arr ++ replicate n a ++ drop (frm + n) arr
```

In Cogent, the function is defined as a foreign function, whose definition is given in C:

```
-- declared in Cogent
type WordArray a
wordarray_set : (WordArray a, U32, U32, a) → WordArray a
-- implemented in C
```

While Cogent and Haskell both support polymorphism, C does not, and QuickCheck cannot perform genuine polymorphic testing [Bernardy, Jansson, et al. 2010]. In this example, we test

[1]The definition of this function in antiquoted-C has been given in Section 4.6.

the **U8** instance of the polymorphic `wordarray_set` function, whose refinement statement is given in Figure 5.2. It can be read roughly as: for any type-correct concrete input `ic` and its abstraction `ia`, check that the result of applying the concrete function `cogent_wordarray_set_u8` (i.e. `oc`) and that of the abstract function `hs_wordarray_set` (i.e. `oa`) are related via the refinement relation `rel_wordarray_u8`. The `corresM'` function is a monadic variant of our *corres* notation for situations where the specification is also deterministic.

Although the *corres* predicate contains an existential quantifier for the result of the non-deterministic abstract specification, our implementation does not require QuickCheck to guess the quantified value from an infeasibly large set. In Section 5.6, we show how to restrict the codomain of the abstract function to be relatively small for efficient testing. This enables our implementation to enumerate over all possible output values of the abstract function execution to find the existentially quantified value.

For the `WordArray` type, we relate the abstract input data and the concrete input data in the following way: we randomly generate test data on a middle-ground type, and then use two thin wrappers `mk_hs_wordarray_set_u8_arg` and `mk_c_wordarray_set_u8_arg` to convert the generated data to the types expected by the abstract and the concrete functions. The test data generation is not very involved, because the correspondence between the two types is straightforward, and the C type is not very convoluted in its underlying representation, in particular it does not heavily use pointers.

Although in the refinement statement, the refinement relation on the input data is expressed as a predicate, this is usually not the way to relate the input data in practice. Checking a predicate is often computationally less costly than computing the abstract input from the concrete one, but it requires two sets of random data generators and forces the two generators to be coupled. Otherwise the correspondence predicate is likely to reject the vast majority of the generated data, rendering the test very inefficient (see Section 5.7.4).

Broadly speaking, it is more convenient to relate the input data if we implement the refinement relation as an *abstraction function*, computing the abstract data according to its concrete counterpart. This is contrary to model-based testing approaches [Utting et al. 2012], in which the test cases are derived from the more abstract model. This is because refinement is usually not unique: the concrete input type normally contains more information than its abstraction. In order to derive a concrete input from the randomly generated abstract input, more data need to be created and this calls for another set of random data generators. On the other hand, when we generate a concrete input and abstract it, it only requires a potentially lossy abstraction function.

`WordArray` is such a case: we compute the abstract input data from the concrete one, relating them by construction. However, not all values of a C type are valid inputs: for instance, a null pointer does not correspond to a valid `WordArray`. To exclude invalid input data, we manually implement a test data generator `gen_wordarray_set_u8_args` which generates values that are isomorphic to valid concrete inputs only, and we convert them to Haskell inputs and C inputs using functions mk_hs_wordarray_set_u8_arg and mk_c_wordarray_set_u8_arg respectively.

The refinement statement as shown in Figure 5.2 is largely boilerplate code, therefore we implement a tool to generate this code [Downing 2021]. The tool consists of a small domain-specific language (DSL), in which the programmers can specify the function names, the definition of the abstraction function for the inputs and the refinement relation between the outputs, and other properties about the refinement statement, such as the determinism of the abstract function and whether the concrete function needs to operate under the IO monad. The DSL is written in JSON format, which can be easily parsed using third-party libraries such as aeson [O'Sullivan 2022]. A piece of sample code is give below.

```
{
  "name"  : "wordarray_set_u8",
  "monad" : true,
  "nondet": false,
  "absf"  : ... // (1) the abstraction function
  "rrel"  : ... // (2) ref. rel. between outputs
}
```

Haskell program texts can be embedded in the JSON structure as the values of the "absf" and "rrel" attributes ((1) and (2) in the code above). This allows the programmers to either call a Haskell function defined elsewhere, or directly write the definition in place. The lens style of code [Kmett 2022] is particularly suitable for this task.

In the refinement statement, the Cogent-compiled C code can be called from Haskell using its C FFI facility.

```
foreign import ccall unsafe "ffi_wordarray_set_u8"
  cogent_wordarray_set_u8 :: Ptr Ct5 → IO (Ptr CWordArray_u8)
```

The **foreign import ccall** declares that the Haskell function cogent_wordarray_set_u8 is the interface to the C function ffi_wordarray_set_u8. Ct5 and CWordArray_u8 are the Haskell representations of the C types for the function's input and output respectively. The Haskell representation of C types, marshalling functions, and foreign function calls are generated by the Cogent compiler and are further compiled by FFI tools such as hsc2hs [GHC User's Guide 2019] and c2hs [Chakravarty 1999].

Running a small number of randomly generated tests (by default 100 but this can be customised) by passing prop_corres_wordarray_set_u8 to the quickCheck function, we get:

```
*WordArray> quickCheck prop_corres_wordarray_set_u8
+++ OK, passed 100 tests.
```

We have specified most of the ADT functions for WordArrays and tested them. We found bugs in two C functions, which had not been uncovered by our earlier test suites nor the file systems built with them. The bugs went undetected as they involved invalid inputs and corner cases which were handled by the callers, whereas the HASKELL specification in our QuickCheck framework does not preclude these input values.

For example, for the `wordarray_copy` function that copies a number of bytes from one memory area to another (similar to `memcpy` in C), the old implementation implicitly assumed that the index into the source word array was always within bounds. This precondition was satisfied by our file system implementations, but it was unspecified. In fact, the `wordarray_copy` function, as part of a generic library, should not carry this implicit precondition in the first place. Otherwise it may introduce bugs to other systems that invoke this library function.

PBT also helped us uncover problems in the Isabelle/HOL ADT specifications, which had overly specific assumptions about inputs. While these assumptions are valid for the functions we verified, they do not hold in general. Thus, the specifications we had written did not represent a general purpose specification of the function.

The `WordArray` library in COGENT was initially axiomatised in the verification of the file systems [Amani, Hixon, et al. 2016], and then tested using the PBT framework, before they were finally formally verified [Cheung et al. 2022]. This is an example of how the developers can progressively increase their confidence in the correctness of the code by upgrading PBT to formal verification in a modular fashion.

5.6 Example: Testing a Top-Level File System Operation

Inspired by JFFS2 [Woodhouse 2001] and UBIFS [Hunter 2008], BilbyFs [Amani 2016] is a flash file system that was designed from scratch, focusing on modularity and verifiability; it has 19 top-level file system operations. We formally verified two functions in Isabelle/HOL and used them to demonstrate how COGENT facilitates equational reasoning. `fsop_sync`, a top-level function, consists of about 300 lines of COGENT code and took approximately 3.75 person months to verify with 5700 lines of proof. The other function, `iget`, directly called by the top-level `fsop_lookup` function, consists of approximately 200 lines of code, and took about one person month to verify with 1800 lines of proof.

We conduct PBT on one of BilbyFs's top-level function, `fsop_readpage`, which had previously been formally specified in Isabelle/HOL but not yet verified. Figure 5.3 shows the Isabelle/HOL specification as well as the manually written HASKELL executable specification; they are very similar in this case. Therefore, testing gives us reasonably high assurance of the implementation with respect to the Isabelle/HOL specification. As discussed in Section 5.3 , this is not always the case, making it occasionally more difficult to connect the two specifications, and sometimes requiring additional manual reasoning.

```
hs_fsop_readpage :: AfsState
                 → VfsInode
                 → OSPageOffset
                 → WordArray U8
                 → NonDet (Either ErrCode (WordArray U8))
hs_fsop_readpage afs vnode n buf =
  let size = vfs_inode_get_size vnode :: U64
      limit = size `shiftR` bilbyFsBlockShift
   in if | n > limit → return $ Left eNoEnt
         | n == limit && (size `mod` bilbyFsBlockSize == 0) →
             return $ Right buf
         | otherwise → return (Right $ fromJust (M.lookup (vfs_inode_get_ino
             inode) afs) !! n) <|>
                        (Left <$> [eIO, eNoMem, eInval, eBadF, eNoEnt])
```

(a) The HASKELL executable specification

```
definition
  afs_readpage :: afs_state
               ⇒ vnode
               ⇒ U64
               ⇒ U8 WordArray
               ⇒ (U8 WordArray × (unit, ErrCode) R) cogent_monad
where
  afs_readpage afs vnode n buf ≡
   if n > (v_size vnode >> unat bilbyFsBlockShift) then
    return (WordArrayT.make (replicate (unat bilbyFsBlockSize) 0), Error eNoEnt)
   else if (n = (v_size vnode >> unat bilbyFsBlockShift)) ∧ ((v_size vnode) mod
    (ucast bilbyFsBlockSize) = 0)
        then return (buf, Success ())
        else do err ← {eIO, eNoMem, eInval, eBadF, eNoEnt};
                return (WordArray.make (pad_block ((i_data (the $ updated_afs afs
     (v_ino vnode))) ! unat n) bilbyFsBlockSize), Success ()) ⊓
                return (buf, Error err)
            od
```

(b) The Isabelle/HOL functional specification

Figure 5.3: Functional specifications of the `fsop_readpage` function

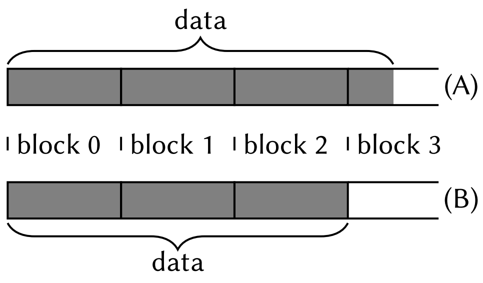


Figure 5.4: An example of the `read_page` algorithm. In case (A), `limit` = 3. When `n` = 0, 1, 2 we just read. When `n` = 3, because the size of the data is not perfectly aligned at the end, we still read. When `n` ≥ 4, we return the no-entry error. In case (B), `limit` = 3. When `n` = 3, that's the special case. We return the old buffer unmodified.

5.6.1 The HASKELL Executable Specification

In a nutshell, as shown in the HASKELL specification in Figure 5.3a, the function `hs_fsop_readpage` fetches a designated data block of a specific file to the buffer. `afs` is a map from inode numbers to files, each of which is represented as a list of blocks of data. The `hs_fsop_readpage` function looks up a file, whose inode number is given by `vnode`, in the map `afs`, and copies the `n`-th block of the file to `buf`. It returns non-deterministically an updated buffer or an error code.

As the first step, `hs_fsop_readpage` calculates the number of blocks that the wanted file occupies. If the block in question is out of bounds (`n > limit`), the function returns a no-entry error `eNoEnt`. If the file size is a multiple of the block size, `n` points to the last block in the wanted file, and the last block is empty (because the file data ends at the prior block boundary), then the function returns the original buffer as there is no data to read. Otherwise, `hs_fsop_readpage` reads the block by looking up the inode number in the map (see Figure 5.4 for a pictorial example).

This, however, is not the only possible correct behaviour. As the implementation has to access buffers and read from the physical medium, this may fail, in which case it should throw an error. We specify this as a non-deterministic behaviour. The specification states that the function can read a block or it can give one of the following five errors: `eIO`, `eNoMem`, `eInval`, `eBadF`, or `eNoEnt`. The `NonDet` monad used here is essentially a finite set containing all allowed behaviours. This monad is commonly used in proving refinement (e.g. [Amani 2016; Cock et al. 2008]). The alternative operator `(<|>)` acts as a non-deterministic choice, admitting the behaviour of either of its operands by taking the union of their behaviours.

5.6.2 Mock Implementations

It is not always feasible to test systems code in its exact production environment [Mostowski et al. 2017]. For instance, the `fsop_readpage` example has many low-level functions which call into the operating system's kernel, and it is currently not feasible to run QuickCheck tests in kernel mode. Instead of testing the monolithic object file obtained from compiling the C code, we mock up parts of the code in Haskell. A mock abstracts from low-level kernel calls and can be thought of as a *black box*, which provides to its caller the same observable effects as the actual implementation.

Mocks can also be used as substitutes for unimplemented functions, enabling systems developers to test functionality before they have a full system implementation. The use of mocks restricts the scope of debugging to a small number of functions, reducing the effort required to locate bugs.

The Cogent implementation of `fsop_readpage` calls a `read_block` function to fetch one block of file data, which in turn retrieves the data with the function `ostore_read`. The `read_block` function, which retrieves the file data from the physical medium, is conceptually simple. However it is complicated in its internal implementation, which involves a red-black tree lookup to locate the address, various layers of caching, and error-handling.

Because `read_block` relies on kernel-mode access to caches and complex data structures, it is a good candidate for a mock implementation. In addition to the `ostore_read` function, we also substitute some kernel ADT functions for mock implementations. For `WordArray` functions invoked by `fsop_readpage`, we use the Haskell models described in Section 5.5 as mocks.

The implementation of a mock is simplified by the fact that it does not need to provide the full functionality of the operation, as long as the callers in the specific test cases cannot observe the difference in its behaviours.

For example, the original Cogent signature of the function `ostore_read` is:

```
type RR x a e = (x, <Success a | Error e>)
ostore_read : (SysState, MountState!, OstoreState, ObjId)
            → RR (SysState, OstoreState) Obj ErrCode
```

This means that the function takes a quadruple as input, containing a read-only (denoted by the `!` operator) `MountState`, and returns the parametric `RR` type, which is further defined as a pair of a variant type `<Success a | Error e>` and a result type `x` that is common to both cases.

The purely functional nature of Cogent makes it easy for the developers to identify the observable behaviours—they are necessarily within the return type of a function. In the case of the `fsop_readpage` function, the only behaviours of `ostore_read` that can be observed are the returned `Obj` value or the error code `ErrCode`. Therefore, we can tailor the mock to the specific use case of `fsop_readpage`. The mock `ostore_read` function can be modelled as a simple map lookup: given a map `OstoreState` and a key of type `ObjId`, it returns the corresponding object or an error. The relevant Haskell definitions are given as follows (the use of the `Oracle` can be ignored for now; it will be explained shortly):

```
type OstoreState = Map ObjId Obj
data Res a e = Success a | Error e
```

```
ostore_read :: Oracle → (OstoreState, ObjId) → Res Obj ErrCode
ostore_read orc (ostore_st, oid) =
  if orc == 0 then
    case M.lookup oid ostore_st of
      Nothing  → Error eNoEnt
      Just obj → Success obj
  else Error orc
```

5.6.3 Oracles and Non-determinism

The Cogent implementation of `ostore_read` interacts with the physical media and kernel data structures, therefore its behaviour is dependent on that of the underlying systems and hardware. However, as we have introduced in Section 2.1, Cogent is a total, purely functional language, meaning that all functions in Cogent have to be deterministic. This non-determinism, therefore, has to be modelled by threading through a global state `SysState`, similar to how the `IO` monad in Haskell is implemented [GHC base 2022].

In the Haskell mock implementation of `ostore_read`, the non-determinism can be modelled in a different manner for simplicity. When testing this function independently, we emulate the non-determinism by adding an additional *oracle* input `orc`.[2] This oracle can be thought of as an input that models all external states on which the function depends, and the basis upon which all non-deterministic choices are made. A similar oracle is passed to our mocks of many `WordArray` functions, such as `wordarray_create`, which allocates memory and creates a fresh word array.

We have seen how oracles can be used in mock implementations to emulate non-determinism. The oracle technique can be similarly applied to the Haskell executable specification. When we test that an oracle-carrying mock refines a non-deterministic specification, the specification can abstract over the values that the oracle can possibly possess with the `NonDet` monad. If the specification is to be made more precise, we can also introduce an oracle to it to ascribe the source of the non-determinism. In this case, care needs to be taken to ensure that the oracle in the specification and that in the implementation are in synchronisation, so that they do not make conflicting choices.

Using oracles in the specification can be problematic if the implementation is not a mock, and is genuinely non-deterministic due to, say, the hardware or the operating system. There is in general no way to predict accurately which execution path the concrete implementation will take (e.g. `malloc` failures). Hence, the specification and the implementation can make inconsistent choices when they encounter non-determinism, which can lead to spurious test failures. In this case, we have to step back and use a non-deterministic specification with the `NonDet` monad

[2] In our implementation, we pass the oracle around using GHC Haskell's implicit parameter extension [Lewis et al. 2000], making it more transparent to users. In the presentation of this chapter, however, we pass them explicitly for clarity.

instead. This, however, has a negative cosmetic effect on the entire HASKELL specification, as the `NonDet` monad is infectious and it would render all ancestor functions in the call graph monadic.

To address this problem, we can establish an equivalence relation between the non-deterministic specification using `NonDet` and the oracle-based deterministic one by testing. Concretely, we test that the two specifications return the same set of results, by enumerating every possible oracle in the deterministic specification, collecting a finite set of results, and then checking that set against the set of results produced by the non-deterministic specification.

The `NonDet` monad and the oracle approach are two extremes of the spectrum, and the developers can choose a suitable degree of non-determinism by combining these two to meet the needs of a specific test case.

5.6.4 Test Data Generation

When using mocks, not only can we choose simpler algorithms to implement the same features, but we can also fine-tune test data generators to restrict the domain of inputs given to the mock, allowing us to implement only a partial mock of the original code.

When testing a cluster of functions and the mock function's input depends on the output of other functions, the aforementioned partial mock should be used with care. The input to a function can be directly controlled by the tester by defining appropriate test data generators, while the output of a function is not as easy to predict as it may have been heavily processed and manipulated by the functions. When such an input value reaches the partial mock implementation, it is less obvious to know a priori whether it falls into the unhandled sub-domain of the mock. It poses a greater challenge in writing good test data generators to ensure the pre-condition of the mock function is met even after the randomly generated data have flowed through other functions in the system under test. More general remarks on this point can be found in Section 5.7.4.

Domain-specific knowledge can be leveraged to write good test data generators, which makes the checking process more efficient and practical to use. For example, when we generate the `OstoreState`, all entries we generate belong to the same inode. In reality, there are many data objects for other files, or other types of objects; but none of these facts will be observed by its caller. This in turn simplifies the implementation of the mock and the abstraction functions.

5.6.5 Results

The shape of the top-level refinement statement for the HASKELL shallow embedding of the COGENT `fsop_readpage` function (shown below) closely resembles that of the `WordArray` example. An oracle is also generated and passed to the HASKELL embedding, which will be further passed to the mock implementation of `ostore_read` as discussed earlier.

```
prop_corres_fsop_readpage :: Property
prop_corres_fsop_readpage =
forAll gen_fsop_readpage_arg $ \ic →
  forAll gen_oracle $ \o →
    let ia = abs_fsop_readpage_arg ic
        oa = uncurry4 hs_fsop_readpage ia
        oc = fsop_readpage o ic
     in corres rel_fsop_readpage_ret oa oc
```

From the counter-examples produced by QuickCheck, we found that the Haskell executable specification was flawed, which in turn exposed a problem with the Isabelle/HOL abstract specification, from which the Haskell specification was derived. These specifications did not take errors returned from `ostore_read` into account. Testing the above property helped us rectify this mis-specification.

5.7 Design Decisions and Key Takeaways

We have showcased two examples of using our PBT framework in Cogent. Since the examples we examined are from a real file system, they have given us some good insights into how much boilerplate code is required and which components of the testing infrastructure can be automatically generated. They have also served as a vehicle for experimenting on correctly integrating these different components.

5.7.1 Modular Testing and Whole-System Testing

Our QuickCheck machinery does not require the user to test the entire system at once. Instead, the user may test refinement for each function or for a cluster of functions at a time. Typically, the ADTs implemented in C form a common module, shared across many systems. Accordingly, our framework allows developers to test the ADTs in isolation, with no regard to how they are used within systems. This modularity is aided by Cogent's functional semantics.

For the `fsop_readpage` example, we chose to employ modular testing as opposed to whole-system testing, which would have required extra effort in developing the infrastructure to run tests in kernel mode (see Section 5.7.3). Whole-system testing allows for more abstract top-level properties to be specified: for instance, that read and write are inverse operations.[3] Alternatively, these logical properties can be tested on top of the executable specification rather than directly on the concrete implementation.

As our specification becomes more abstract, the single step simulation style of refinement becomes less relevant, because several low-level functions may be specified as a single function

[3]It might be surprising to some readers that we claim that this property is suitable for whole-system testing, instead of PBT. After all, this kind of round-trip properties are typical in the realm of PBT. The reason is that, systems software, or file systems in this case, are not implemented cleanly in a purely functional manner. They always involve heavy I/O, kernel interaction, locking, etc., which cannot be precisely and concisely specified in the functional specification and thus fall under the global state.

on the abstract level. Modular testing, on the other hand, is more comprehensive, as it also examines the interfaces among different components in a system. In this case, it uncovered issues in the `WordArray` implementation that whole-system testing would have, and indeed had, missed.

5.7.2 Functional Specification Versus Logical Properties

Traditional PBT tests specifications against a set of logical properties (e.g. *get* and *set* are inverse operations on `WordArrays`). We instead test functions against a full executable specification that models the functions. This is conceptually similar to model-based testing [Koopman, Achten, et al. 2012] (also see Section 5.8). The functional specification is most akin to the functional notions of model paradigm as classified in the work by Utting et al. [2012, § 3.3].

Our ADT functions, which are implemented in C, can be modeled using readily available HASKELL library functions or can be easily defined in terms of library functions. Moreover, low-level file system operations are often pretty concrete and only perform simple tasks, making it hard to specify traditional properties about them that are intuitive or easily comprehensible. In both cases, it is easier to define a model rather than a set of properties that define the behaviour of such functions [Koopman, Achten, et al. 2012].

In addition, a model of an ADT serves as a mock implementation when testing functions that use these ADTs. This enables modular testing. For instance, in our `fsop_readpage` case study, we used the previously defined HASKELL model of the `WordArray` functions as mocks.

Furthermore, such models additionally serve as a communication interface between system programmers and proof engineers. The models are key to designing verification-friendly systems programs, whereas logical properties alone fall short for this purpose.

5.7.3 Testing Kernel Modules

A file system is typically compiled as a kernel module and runs in kernel mode, while our test framework runs in user mode. To handle this discrepancy, for our prototype, we have ported our file systems code to run in user mode, using mocks to simulate the kernel API. Emulating the kernel is common practice in systems programming, with libraries such as FUSE [FUSE 2019] facilitating the user-space execution of kernel code. However, these tools expect a complete kernel module, thus precluding the use of mocks or other user-land code during testing. A possible alternative to explore is using a system such as KML [Maeda 2015], House [Hallgren et al. 2005] or HaLVM [HaLVM n.d.] to run PBT in kernel mode. This would allow the testing environment to have a closer resemblance to the real run-time environment of the software. Other potential solutions include the use of unikernels (or libraryOS) [Raza et al. 2019]. We leave them for future investigation.

5.7.4 Test Generation Strategies

There are two main factors to consider when generating test data for PBT. The first is how to sample the data: In this work, we choose user-guided random test generation à la QuickCheck. Exhaustive testing (for small values) is another popular strategy and has gained great popularity, e.g. SmallCheck for HASKELL [Runciman et al. 2008]. However, the small scope hypothesis on which SmallCheck is based does not hold in general in the context of systems software[4]. For instance, integer overflow, which is a common bug in systems code, can hardly be triggered by small values. The second main concern is the effective generation of test data which satisfies the premises of the properties. If the property is of the form $p \implies q$ and the premise p is very strong (i.e. difficult to satisfy), and the test data is not sampled with great care, then a lot of them will falsify the premise and thus be discarded in the test, rendering the test very inefficient. All refinement statements in this work have the form $p \implies q$ with a strong precondition p. The Luck framework by Lampropoulos, Gallois-Wong, et al. [2017] proposes coupling the predicates of the property and the test data generation, which could simplify writing custom test data generators. There is a rich body of research devoted to test generation techniques [Duregård et al. 2012]. In the future, we plan to explore more options to automate our test data generators.

Domain-specific knowledge can be leveraged to generate test data more efficiently. Currently encoding this knowledge in the test data generator is a manual effort by the testers, and it is decoupled from the program under test. User annotations on the program could be one way to specify domain-specific knowledge, which can then be used for generating tests. This user annotation can be given as refinement types, so that the test generators remain type-based. More precise types like refinement types improves the overall productivity of testing as well, as the input domain becomes smaller. In Chapter 6, we explore refinement types.

Higher-order functions are essential to functional languages, but testing high-order functions usually requires some extra mechanism (e.g. [Koopman and Plasmeijer 2006; Xia 2022]). COGENT supports higher-order functions and our testing framework supports generating functions. However, it takes a different approach to the QuickCheck library. QuickCheck essentially generates a representation of an arbitrary function according to its domain and codomain, whereas our framework exploits the technique with which the COGENT compiler generates functions in C. Instead of compiling function objects to function pointers in C, the COGENT compiler generates a numeric token for each declared function in the same program and uses static dispatch functions to realise high-order function calls, as we have seen in Section 4.3.3. This design allows us to simply choose an integer number randomly as the representation of the function object. It makes the tests more efficient, as the test driver only tests the functions that can be passed as arguments to higher-order functions. The limitation is that higher-order functions cannot be tested in isolation from the rest of the program.

[4]The small scope hypothesis is stated in Runciman et al. [2008]'s paper as: "(1) If a program fails to meet its specification in some cases, it almost always fails in some simple case. Or in contrapositive form: (2) If a program does not fail in any simple case, it hardly ever fails in any case."

5.7.5 Shrinking

Counter-example shrinking reduces the size of counter-examples before reporting them to testers, which plays an important role in helping the tester effectively understand and fix bugs. For example, the Haskell QuickCheck comes with a customisable shrinking library with a default shrinking algorithm for many datatypes. A rich body of research can be found on more advanced shrinking algorithms. For example, test data shrinking that preserve invariants about the generated data has been explored in [MacIver 2016a; Stanley 2019]. But due to the lack of recursively defined datatypes in Cogent and thus in the Cogent-powered file systems, the effectiveness of shrinking is dubious, as the size of the input data chiefly comes from the sheer complexity of the (non-recursive) datatypes, rather than from recursion. Shrinking is nevertheless useful for testing ADTs, but basic shrinking strategies work reasonably well in our context.

5.8 Related Work

QuickCheck has been used for testing a variety of high-level properties, such as information flow control [Amorim et al. 2014; Hriţcu et al. 2013], mutual exclusion [Claessen, Palka, et al. 2009], and the functional correctness of AUTOSAR components [Arts et al. 2015; Mostowski et al. 2017]. To the best of our knowledge, our framework is the first to use PBT for testing refinement-based functional correctness statements.

The `hs-to-coq` tool [Spector-Zabusky et al. 2018] translates Haskell code into the Coq proof assistant [Bertot and Castéran 2004]. Breitner et al. [2018] used it to verify parts of Haskell's container library in Coq. In addition to proving the functional correctness of various functions in the library, they also verified that the QuickCheck properties that the library is tested against are correct. By contrast, our QuickCheck properties are refinement properties that directly resemble the those used for full verification. Verifying these properties is already a substantial step towards proving functional correctness, and in some cases directly implies functional correctness.

A version of QuickCheck is available as a built-in tool in Isabelle/HOL and is used for quickly finding counter-examples to proposed lemmas [Berghofer and Nipkow 2004; Bulwahn 2012]. We chose to build on Haskell's QuickCheck rather than Isabelle/HOL's QuickCheck because it is easier for Cogent programmers to use a testing framework that lies in the ecosystem of a functional programming language rather than interact with a theorem prover. Haskell acts as a good communication medium between programmers and proof engineers [Breitner et al. 2018; Derrin et al. 2006]. Moreover, due to Isabelle/HOL's interactive nature, testers would have to wait for Isabelle to re-establish all the processed script and proven facts once *anything* changes in the theory files, before they are able to test. For example, when the definition of `SysState` (see Section 5.6) changes in BilbyFs, it can easily take up to several dozen minutes for Isabelle to reach the same conjecture that the tester is examining.[5] A similar wait can also be expected even though the changes are irrelevant to the testing task, which means during testing the developers cannot do little in parallel. Even if Isabelle's `quick-and-dirty` mode is used, which skips proofs, testers would still have to wait for Isabelle/HOL to process definitions. In fact, a large portion of the time is spent on reading in the deep embeddings of the COGENT program into Isabelle, due to the large terms generated by the COGENT compiler. This would cause a significant and unnecessary slowdown to their productivity, and destroy the user experience.

The SPARK language [SPARK Pro 2014], a formally defined subset of Ada, also uses a combination of testing and verification to facilitate the development of reliable software. SPARK developers can attach contracts, that is, specifications of pre- and postconditions, to critical procedures. Tools of the framework can use these contracts as input to automatically test the procedures, or attempt to formally prove that the implementation observes these contracts. Ada language features which are hard to verify, such as side-effects in expressions, access types, allocators, exception handling and many others, are not permitted in SPARK. Others are, but can lead to gaps in the verification. In contrast to COGENT, SPARK does not aim at fully verified systems from high-level specification to machine code, but at selectively verifying safety critical components.

DoubleCheck [Eastlund 2009] integrates PBT into Dracula [Vaillancourt et al. 2006], a pedagogic programming environment which enables students to develop programs and then prove theorems about them in ACL2 [Kaufmann and Moore 2018], a theorem prover based on term rewriting. As with our work, the motivation of this integration is to facilitate formal verification, though its focus is on education, not on producing verified real-world applications.

In the PBT framework we presented, as we test the refinement statement between the implementation and the HASKELL executable specification, which can be considered a *conformance relation*, it does appear that we are instead conducting model-based testing [Tretmans 2011; Utting et al. 2012]. Our approach does indeed share a lot in common with MBT, but we identify our approach as PBT for the following reasons. Firstly, in MBT, the starting point of testing is a model of the software under test. On the contrary, as we have demonstrated, testing in our framework does not necessarily have an existing model to start with. In developing formally verifiable operating systems components, which is the application domain that concerns us, it is of paramount importance to find the right balance between verifiability and performance. PBT gives developers insights in both aspects. Therefore, testing plays a role in the design of the system, and subsequently its specification. This is similar to the iterative development process reported in the seL4 formal verification work [Heiser, Andronick, et al. 2010]. Secondly, test cases are systematically and algorithmically generated from the model in MBT. Test inputs are typically concretised from the abstract test suite and the test results are abstracted to be validated against the model by an adapter. In contrast, as we have shown in the examples, the test cases

[5]The experiment is done on a Dell Precision T1700 machine with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 32GiB DDR3 1600MHz RAM, running Debian 4.9.130-2 x86_64 GNU/Linux.

are not generated from the model. Test data is directly produced on the concrete level. Lastly, from the tooling perspective, our approach uses a PBT library QuickCheck as the core of the testing infrastructure.

5.9 Conclusion

In this chapter, we showed how we augmented the Cogent verification framework with PBT. Testing and formal verification complement each other, which is well acknowledged among researchers and developers. In this work, we further demonstrated this common belief in the specific context of PBT and interactive theorem proving. The central idea is to mirror the refinement proof in testing, using a functional specification as the model instead of a set of logical properties as commonly done in PBT.

Using this method, we tested an abstract data type from a library, as well as an operation of a real-world file system. The tests exposed several bugs in the ADT implementation and uncovered errors in the specification of the ADT and of the file system.

Besides the main purpose of testing—detecting bugs—we exhibited other benefits of employing PBT. They include reduced effort in formal verification, structured verification-ready specification and implementation, and precise and effective communication media among developers. We believe PBT offers developers the opportunity to gradually tackle the verification challenge in large and complex systems development, serving as a helpful stepping stone in the endeavour into full formal verification of high assurance software.

Chapter 6

A Hoare-Logic Style Refinement Types Formalisation

This chapter is derived from the following publication:

- Zilin Chen. Sept. 2022. "A Hoare Logic Style Refinement Types Formalisation." In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (TyDe '22). ACM, Ljubljana, Slovenia, 14 pages. DOI: 10.1145/3546196.3550162

Cogent is designed for formally verifying low-level systems code. It abstracts the semantics of low-level C code to a purely functional semantics, on which proof engineers reason about program behaviours. This proof is established fully manually (except the automation from the Isabelle/HOL tactics), and it is typically done by proof engineers asynchronously, independent of program development. Refinement types are a lightweight yet expressive tool for specifying and reasoning about programs. Employing a refinement type system in Cogent allows more theorems to be specified by systems developers in the program as types, automatically proved synchronously with program development during typechecking. As a first step towards integrating refinement types into Cogent, we explore refinement typechecking and verification condition generation from a more theoretical angle, in a generic setting independent of Cogent.

The connection between refinement types and Hoare logic has long been recognised but the discussion remains largely informal. In this chapter, we present a Hoare triple style Agda formalisation of a refinement type system on a simply-typed λ-calculus restricted to first-order functions. In our formalisation, we interpret the object language as shallow Agda terms and use Agda's type system as the underlying logic for the type refinement. To deterministically typecheck a program with refinement types, we reduce it to the computation of the weakest precondition and define a verification condition generator which aggregates all the proof obligations that need to be fulfilled to witness the well-typedness of the program.

6.1 Introduction

Program verification has been widely adopted by software developers in the past few decades, and is becoming standard practice in the development of safety-critical software. Refinement types[1], due to its tight integration into the programming language, has seen a surge in popularity among language designers and users in recent years [Lehmann, Kunkel, et al. 2021; Lehmann and Tanter 2017; Pavlinovic et al. 2021; Polikarpova et al. 2016; Swamy et al. 2016; Vazou 2016]. It is effective yet easy to harness for programmers who are less skillful at using manual verification tools, such as interactive theorem provers. Programmers can annotate their code with types that include predicates which further restrict the inhabitants of that type. For instance, $\{\nu : \mathbb{N} \mid \nu > 0\}$ is a type that only includes positive natural numbers. We typically call the type being refined, namely $\mathbb{N}$ here, the *base type*, and the logical formula the *refinement predicate*. The typechecker of a refinement type system will produce verification conditions, in the form of logical entailments, to justify the specified refinements. In the typechecker, a Satisfiability Modulo Theories (SMT) solver is responsible for automatically discharging these verification conditions. Although SMT-solving is undecidable in general, language designers typically design their type systems carefully so that all allowable refinement predicates fall under a decidable fragment of the logic, rendering SMT-solving and consequently typechecking decidable.

As we have seen in earlier chapters, Cogent's refinement verification framework provides users with a nice purely functional semantics of the Cogent program, while they can still enjoy the runtime characteristics of the target C language. Reasoning about the behaviour of a Cogent program against a formal specification in Isabelle/HOL is made easy by this purely functional abstraction: equational reasoning techniques can be applied in the proof. This proof, however, still has to be constructed manually in an interactive manner. In Amani, Hixon, et al. [2016]'s work, it is reported that the effort for verifying two BilbyFs functions `sync()` and `iget()` was roughly 9.25 person months, and produced roughly 13,000 lines of proof for the 1,350 lines of Cogent code. The amount of manual verification work is still significant, if a full system is to be verified.

Extending Cogent with a refinement type system can alleviate this burden on developers. Specifically, we propose adding a refinement type system to Cogent in the following way, in addition to its current verification framework: Cogent developers can opt to write specifications as refinement types in Cogent, then the refinement type system will generate verification conditions, which the SMT-solver will try to prove. The refinement predicate logic can be very expressive and does not have to be decidable. The SMT-solver will try to solve as many theorems as it practically can; the rest will be generated as lemmas in an interactive theorem prover, say Isabelle/HOL, and verification engineers can manually verify them as they already do with Cogent's manual proofs. The employment of a mixture of SMT-solving and interactive theorem

[1]The term *refinement types* is unfortunately overloaded. In this chapter, we discuss refinement types in the sense of *subset types* [Greenberg 2015].

proving is similar to F* [Swamy et al. 2016].

Augmenting Cogent with a refinement type system has other potential benefits. In Chapter 5, we proposed the use of Haskell's QuickCheck library to test Cogent programs. One key component in the QuickCheck framework is to generate test data automatically. The addition of refinement types facilitates test data generation. As we would have more precise types, the type-based test data generators can be tailored to only generate valid random input values. Similar ideas have also been previously explored by Seidel et al. [2015] and Zhu et al. [2015].

Adding a refinement type system to Cogent is not straightforward. Indeed, a refinement type system is already complicated on it own, especially if we want to formalise it in order to include it in Cogent's formal verification framework. Typically, a refinement type system supports *dependent functions*, which are similar to those in a dependent type system [Martin-Löf 1984]. A dependent function allows the refinement predicate of the return type to refer to the value of the function's arguments. Such term-dependency also results in the typing contexts being telescopic, meaning that a type in the context can refer to variables in earlier entries of that context.

Another source of complexity in refinement type systems is solving the logical entailment which determines the subtyping relation between two types. Usually, some syntactic or semantic rewriting tactics will be employed to carefully transform the entailment into a certain form that facilitates the automatic discharge of proof obligations using an SMT-solver. To ensure that SMT-solving is decidable, language designers typically need to restrict the logic used for expressing refinement predicates. For instance, in Liquid Haskell [Vazou et al. 2014], the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) is chosen to ensure decidability.

In the rich literature on refinement types, while there are notable exceptions (e.g. [Lehmann and Tanter 2016]), due to the complexity of refinement type systems, the development remains largely informal[2] and rather ad hoc. For instance, the typing rules of each variant of a refinement type system can be subtly different, whereas the underlying reasons for the difference are not always systematically analysed and clearly attributed.

The connections between refinement types and Hoare logic have long been recognised. For example, the work by Jhala and Vazou [2021] indicates that "refinement types can be viewed as a generalization of Floyd-Hoare style program logics." The monolithic assertions on the whole program states in Hoare logic can be decomposed into more fine-grained refinements on the values of individual terms. Pre- and post-conditions correspond directly to functions' input and output types.

In a blog post [Jhala 2019], Jhala further explains why refinement types are different (and in some aspects, superior to) Hoare logic, with the punchline that "types decompose quantified assertions into quantifier-free refinements", which is a recipe for rendering the logics decidable

[2]Informal in the sense that they are lacking machine-checked proofs.

for SMT-solvers.

The formal connections between refinement types and Hoare logic deserve more systematic studies. In this chapter, we present a unifying paradigm—a Hoare logic style formalisation of a refinement type system on a simply-typed λ-calculus with primitive types, products and restricted to first-order functions. Formalising refinement types in the Hoare logic style not only allows us to study the connections between these two systems, it also makes the formalisation easier by avoiding the aforementioned complications in refinement type systems. The formalisation is done in Agda [Norell 2009, 2007], a dependently-typed proof assistant. In our formalisation, shallow Agda terms are used to denote the semantics of the object language and Agda's type system is used as the underlying logic for the refinement predicates.

In a nutshell, we formulate the typing judgement of the refinement type system as $\Gamma\{\phi\} \vdash e : T\{\lambda\nu.\psi\}$. When reading it as a regular typing rule, the typing context is split into two parts: Γ is a list of term variables associated with their base types, and ϕ contains all the refinement predicates about these variables in the context. The base type T and the predicate $\lambda\nu.\psi$ form the refinement type $\{\nu : T \mid \psi\}$ that is assigned to e. On the other hand, if we read the rule as a Hoare triple, e is the program and ϕ and ψ are the pre- and post-conditions of the execution of e under the context Γ.

When we make the analogy between refinement type systems and Hoare logic, another analogy naturally arises. The type analysis with a refinement type system correlates to the weakest precondition calculation in Hoare logic. In fact, the idea of using the weakest precondition for refinement type inference is not new. Knowles and Flanagan [2007] have proposed, as future work, calculating the weakest precondition for refinement type reconstruction. In this chapter, we explore how to use backward reasoning for typechecking, with our machine-checked formalisation in Agda.

Specifically, this chapter presents the following technical contributions:

- We formalise a refinement type system (Section 6.3 and Section 6.4) à la Hoare logic and prove, among other meta-theoretical results, that the type system is sound and complete with respect to the denotational semantics (Section 6.5).
- We define a naïve weakest precondition function wp in lieu of a typechecking algorithm and prove that it is sound and complete with respect to the typing rules (Section 6.6).
- We revise the formalisation above and present a variant of the refinement type system which preserves the contracts imposed by functions (i.e. λ-abstractions). This requires a more sophisticated weakest precondition function pre and a verification condition generator vc. We again establish the soundness and completeness results of pre and vc (Section 6.7).

All the formalisation is developed in and checked by Agda (version 2.6.2.1). In fact, the main body of this chapter is generated from a literate Agda file that contains all the formal development, including the proofs of all the theorems presented in this chapter. The source file of this chapter can be obtained online (`https://github.com/zilinc/ref-hoare`).

6.2 The Key Idea

Typically, a refinement type can be encoded as a Σ-type in a dependently-typed host language. For example, in Agda's standard library, a refinement type is defined as a record consisting of a value and an irrelevant proof of a property P over the value.[3]

Embedding and working with a dependently-typed language is often quite tedious. Encoding such an object language in another dependently-typed language can typically be done with the help of inductive-recursive techniques [Dybjer 2000]. The dependent object language features telescopic contexts in the typing rules. As a result, it poses extra challenges in manipulating typing contexts and in performing type inference, because the dependency induces specific topological orders in solving type constraints.

Realising the connections between refinement types and Hoare logic can be a rescue. When assigning a refinement type to a function (we assume, without loss of generality, that a function only takes one argument), the refinement predicate on the argument asserts the properties that the input possesses, and the predicate on the result type needs to be satisfied by the output. This mimics the structure of a Hoare triple: the predicates on the input and on the output correspond to the pre- and post-conditions respectively.

Another correlation is that, as we have alluded to earlier, in a traditional refinement typing judgement $\overrightarrow{x_i : \tau_i} \vdash e : \tau$ (we use an overhead $\overrightarrow{\text{arrow}}$ to denote an ordered vector, and an overhead $\overline{\text{line}}$ for an unordered list; τ_i and τ here are refinement types), the refinement predicates in $\overline{\tau_i}$ correlate to the precondition in a Hoare triple, and the predicate in τ corresponds to the postcondition. Concretely, if each τ_i is in the form $\{\nu : B_i \mid \phi_i\}$, we can take the conjunction of all the ϕ_i to form a proposition about $\overline{x_i}$, which becomes the precondition describing the program state (i.e. the typing context) under which e is executed. Similarly, if τ is $\{\nu : B \mid \psi\}$, then ψ is the postcondition that the evaluation result ν of e must satisfy.

The Hoare triple view of refinement types has many advantages. Firstly, it separates the checking or inference of the base types and that of the refinement predicates, which is common practice in languages with refinement types (e.g. [Knowles and Flanagan 2007; Pavlinovic et al. 2021; Rondon et al. 2008]). The base type information ensures that the refinement predicates are well-typed. Secondly, the separation of types and predicates means that there is no longer any term-dependency in types, and there is no telescopic contexts any more. It makes the formalisation and the reasoning of the system drastically simpler. In particular, the typing contexts no longer need to maintain any particular order.

In this chapter, we study a small simply-typed λ-calculus with primitive types, products, and only first-order functions. We assume that all programs are well-typed under the simple

[3]`https://github.com/agda/agda-stdlib/blob/367e3d6/src/Data/Refinement.agda`

type system and only focus on the type refinement. We require all functions (λ-abstractions) to be annotated with refinement types and they are the only places where type annotations are needed. We only typecheck a program against the given annotations, without elaborating the entire syntax tree with refinement types. We reduce the typechecking problem to the computation of the weakest precondition of a program and define a verification condition generator which aggregates all the proof obligations that need to be fulfilled to witness the well-typedness of the program. The proof of the verification conditions is left to the users, who serve as an oracle for solving all logic puzzles. Therefore we do not concern with the decidability of the refinement logic.

6.3 The Base Language λ^B

Our journey starts with a simply-typed λ-calculus λ^B without any refinement. The syntax of the λ^B is shown in Figure 6.1. It has ground types of unit ($\mathbb{1}$), Booleans ($\mathbb{2}$) and natural numbers ($\mathbb{N}$), and product types. These types are called base types, meaning that they are the types that can be refined, i.e. they can appear in the base type position B in a typical refinement type $\{\nu : B \mid \phi\}$. The language is restricted to first-order functions by excluding function arrows from the first-class base types. The term language is very standard, consisting of variables (x), constants of the ground types, pairs, projections (π_1 and π_2), function applications (denoted by juxtaposition), if-conditionals, non-recursive local let-bindings, and some arithmetic and logical operators on natural numbers and Booleans.

Although λ-abstractions can only be directly applied, we do not eschew them in the syntax. This allows us to define top-level functions, which can be reused. This design decision is primarily only for practical convenience and is otherwise unimportant. The Agda formalisation follows this design; it handles function and non-function terms separately. Whenever possible, however, we merge the two when we present them in this chapter.

Since the typing rule for λ^B is very standard, we directly show its Agda embedding and use it as a tutorial on how we construct the language in Agda. We use an encoding derived from McBride's Kipling language [McBride 2010], which allows us to index the syntax of the object language with its type in Agda. Therefore, the object term language is type correct by construction, up to simple types.

In Figure 6.2, we define a universe **U** of codes for base types, and a decoding function $\mathscr{E}[\![\cdot]\!]_{\mathsf{Ty}}$ (`⟦_⟧τ` in Agda) which maps the syntax to the corresponding Agda types. We do not include a code constructor for function types; a (non-dependent) function type is interpreted according to its input and output types.

McBride [2010] uses inductive-recursive definitions [Dybjer 2000] for embedding his dependently-typed object language in Agda, which is a pretty standard technique (e.g. [Chapman 2009; Danielsson 2006]). In our base language (and also later with refinement types), since the term-dependency in types has been eliminated by the Hoare logic style formulation, the

base types	B, S, T	$::=$	$\mathbb{1} \mid \mathbb{2} \mid \mathbb{N} \mid S \times T$
func. types		$\ni$	$S \longrightarrow T$
expressions	e	$::=$	$x \mid () \mid \mathsf{true} \mid \mathsf{false}$
		$\mid$	$\mathsf{ze} \mid \mathsf{su}\ e$
		$\mid$	$(e, e) \mid \pi_1\ e \mid \pi_2\ e \mid f\ e$
		$\mid$	$\textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2$
		$\mid$	$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$
		$\mid$	$e_1 \oplus e_2$
binary operators		$\ni$	$\oplus$
functions	f	$::=$	$\lambda x.e$
contexts	Γ	$::=$	$\cdot \mid \Gamma, x : S$

Figure 6.1: Syntax of the language λ^B

```
data U : Set where
  `1´ `2´ `ℕ´ : U
  _`×´_ : U → U → U
```

```
⟦_⟧τ : U → Set
⟦ `1´ ⟧τ = ⊤
⟦ `2´ ⟧τ = Bool
⟦ `ℕ´ ⟧τ = ℕ
⟦ S `×´ T ⟧τ = ⟦ S ⟧τ × ⟦ T ⟧τ
```

Figure 6.2: The Agda definition of the base types and their interpretation

```
data Cx : Set where                    ⟦_⟧C : Cx → Set
  `E´ : Cx                             ⟦ `E´ ⟧C = ⊤
  _▸_ : Cx → U → Cx                    ⟦ Γ ▸ S ⟧C = ⟦ Γ ⟧C × ⟦ S ⟧τ
```

Figure 6.3: The syntax and the denotation of the typing context

```
data _∋_ : (Γ : Cx)(T : U) → Set where
  top : ∀{Γ}{T} → Γ ▸ T ∋ T
  pop : ∀{Γ}{S T} → Γ ∋ T → Γ ▸ S ∋ T

⟦_⟧∋ : ∀{Γ}{T} → Γ ∋ T → (γ : ⟦ Γ ⟧C) → ⟦ T ⟧τ
⟦ top ⟧∋ (_ , t) = t
⟦ pop i ⟧∋ (γ , _) = ⟦ i ⟧∋ γ
```

Figure 6.4: Variable indexing in contexts Γ and γ

inductive-recursive definition of the universe à la Tarski and its interpretation is not needed. Nevertheless, we choose to use the vocabulary from that lines of work since the formalisation is heavily inspired by them.

With the denotation of types, we define what it means for a denotational value to possess a type, following the work on semantic typing [Milner 1978]. A denotational value v possesses a type T, written $\vDash v : T$, if v is a member of the semantic domain corresponding to the type T. This is to say, v is a shallow Agda value of type $\mathscr{E}[\![T]\!]_{\mathsf{Ty}}$.

Next, we define the typing context for the language λ^B, and the denotation of the context in terms of a nested tuple of shallow Agda values (see Figure 6.3). The denotation of the typing context gives us a *semantic environment* γ, mapping variables to their denotational values in Agda. A semantic environment γ *respects* the typing context Γ if for all $x \in \mathrm{dom}(\Gamma)$, $\vDash \gamma(x) : \Gamma(x)$.

Variables in λ^B are nameless and are referenced by their de Bruijn indices in the context, with the rightmost (also outermost) element bound most closely. Unlike in Kipling [McBride 2010], the direction to which the context grows is largely irrelevant, since the context is not telescopic. The variable indexing for the typing context Γ and the semantic context γ are defined in Figure 6.4 respectively.

Before we continue, we introduce a few combinators that are helpful in simplifying the presentation. ^{k} and ^{s} are the **K** and **S** combinators from the SKI calculus. Infix operators ˆ and ˇ are synonyms to the currying and uncurrying functions respectively.

The syntax of the language is defined in Figure 6.5, and the interpretation functions ⟦_⟧⊢ and ⟦_⟧⊢⃗ are given in Figure 6.6. The deep syntax of the object language is indexed by the

```
mutual
  data _⊢_ (Γ : Cx) : U → Set where
    VAR  : ∀{T} → Γ ∋ T → Γ ⊢ T
    UNIT : Γ ⊢ `1´
    TT   : Γ ⊢ `2´
    FF   : Γ ⊢ `2´
    ZE   : Γ ⊢ `ℕ´
    SU   : Γ ⊢ `ℕ´ → Γ ⊢ `ℕ´
    IF   : ∀{T} → Γ ⊢ `2´ → Γ ⊢ T → Γ ⊢ T → Γ ⊢ T
    LET  : ∀{S T} → Γ ⊢ S → Γ ▸ S ⊢ T → Γ ⊢ T
    PRD  : ∀{S T} → Γ ⊢ S → Γ ⊢ T → Γ ⊢ (S `×´ T)
    FST  : ∀{S T} → Γ ⊢ S `×´ T → Γ ⊢ S
    SND  : ∀{S T} → Γ ⊢ S `×´ T → Γ ⊢ T
    APP  : ∀{S T} → (Γ ⊢ S ⟶ T) → Γ ⊢ S → Γ ⊢ T
    BOP  : (o : ⊕) → Γ ⊢ →⊕ o → Γ ⊢ →⊕ o → Γ ⊢ ⊕→ o

  data _⊢_⟶_ (Γ : Cx) : U → U → Set where
    FUN : ∀{S T} → Γ ▸ S ⊢ T → Γ ⊢ S ⟶ T
```

Figure 6.5: The Agda embedding of λ^B. ⊕ is the deep syntax for binary operators; →⊕ and ⊕→ return the input and output types of a binary operator respectively.

typing context and the deep type. It therefore guarantees that the deep terms are type correct by construction. There is little surprise in the definition of the typing rules. We only mention that FUN has the same type as a normal first-class λ-abstraction does. It can be constructed under any context Γ and does not need to be closed. A function application can be represented equivalently as a let-binding up to simple types, but they have different refinement typing rules, as we will see later in this chapter.

The interpretation of the term language is entirely standard, mapping object language terms to values of their corresponding Agda types. On paper, we write $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$ for the denotation function, which takes a deep term and a semantic environment and returns the Agda denotation.

As a simple example, if we want to define a top-level function

$$f_0 : \mathbb{N} \longrightarrow \mathbb{N}$$
$$f_0 = \lambda x.\, x + 1$$

it can be done in Agda as

```
f₀ : ∀{Γ} → Γ ⊢ `ℕ´ ⟶ `ℕ´
```

```
mutual
  ⟦_⟧⊢ : ∀{Γ}{T} → Γ ⊢ T → ⟦ Γ ⟧C → ⟦ T ⟧τ
  ⟦ VAR x ⟧⊢ = ⟦ x ⟧∋
  ⟦ UNIT ⟧⊢ = ᵏ tt
  ⟦ TT ⟧⊢ = ᵏ true
  ⟦ FF ⟧⊢ = ᵏ false
  ⟦ ZE ⟧⊢ = ᵏ 0
  ⟦ SU e ⟧⊢ = ᵏ suc ˢ ⟦ e ⟧⊢
  ⟦ IF c e₁ e₂ ⟧⊢ = (if_then_else_ ∘ ⟦ c ⟧⊢) ˢ ⟦ e₁ ⟧⊢ ˢ ⟦ e₂ ⟧⊢
  ⟦ LET e₁ e₂ ⟧⊢ = ^ ⟦ e₂ ⟧⊢ ˢ ⟦ e₁ ⟧⊢
  ⟦ PRD e₁ e₂ ⟧⊢ = < ⟦ e₁ ⟧⊢ , ⟦ e₂ ⟧⊢ >
  ⟦ FST e ⟧⊢ = proj₁ ∘ ⟦ e ⟧⊢
  ⟦ SND e ⟧⊢ = proj₂ ∘ ⟦ e ⟧⊢
  ⟦ APP f e ⟧⊢ = ⟦ f ⟧⊢⃗ ˢ ⟦ e ⟧⊢
  ⟦ BOP o e₁ e₂ ⟧⊢ γ = ⟦ e₁ ⟧⊢ γ ⟦ o ⟧⊢⊕ ⟦ e₂ ⟧⊢ γ

  ⟦_⟧⊢⃗ : ∀{Γ}{S T} → Γ ⊢ S ⟶ T → ⟦ Γ ⟧C → ⟦ S ⟧τ → ⟦ T ⟧τ
  ⟦ FUN e ⟧⊢⃗ = ^ ⟦ e ⟧⊢
```

Figure 6.6: The Agda definition of the interpretation function for λ^B. `_⟦ o ⟧⊢⊕_` interprets the deep operator *o* as its Agda counterpart.

```
f₀ = FUN (BOP [+] (VAR top) ONE)
```

where `ONE` is defined as `SU ZE`. Note that the function is defined under any context Γ. The denotation of the `f₀` function under any valid semantic environment γ is:

```
⟦f₀⟧⊢ : ∀{Γ}{γ : ⟦ Γ ⟧C} → ℕ → ℕ
⟦f₀⟧⊢ {γ = γ} = ⟦ f₀ ⟧⊢⃗ γ
```

Evaluating this term in Agda results in a λ-term: `λ x → x + 1`, independent of the environment γ.

6.4 Refinement Typed Language λ^R

We introduce a refinement typed language λ^R that is obtained by equipping λ^B with refinement predicates. We first present the syntax of the language in Figure 6.7.[4] The upcast operator for non-function expressions is used to make any refinement subtyping explicit in the typing tree.

[4]A remark on the notation: when we talk about the dependent function types in the object languages, we use a slightly longer function arrow ⟶ as a reminder that it is not a first-class type constructor. The typesetting is only

$$
\begin{array}{llcll}
\text{ref. types} & \tau & ::= & \{\nu : B \mid \phi\} & \\
\text{func. types} & & \ni & x : \tau \longrightarrow \tau & \text{(dep. functions)} \\
\text{expressions} & \hat{e} & ::= & \ldots & \text{(same as } \lambda^B\text{)} \\
 & & \mid & \hat{e} :: \tau & \text{(upcast)} \\
\text{functions} & \hat{f} & ::= & \lambda x.\hat{e} & \\
\text{ref. contexts} & \hat{\Gamma} & ::= & \Gamma; \phi & \\
\text{predicates} & \phi, \xi, \psi & ::= & \ldots & \text{(a logic of choice)}
\end{array}
$$

Figure 6.7: Syntax of the language λ^R

In contrast to the traditional formulation of refinement type systems, where a typing context is defined as $\overrightarrow{x_i : \tau_i}$, we split it into a base typing context Γ and a refinement predicate ϕ that can be constructed by taking the conjunction of all predicates in $\overline{\tau_i}$. This formulation is arguably more flexible to work with. It does not enforce any ordering in the context entries, and it is easier to add path sensitive constraints that are not bound to a program variable. For example, when typechecking an expression **if** c **then** e_1 **else** e_2, we need to add the constraint c to the context when we zoom in on e_1. This can typically be done by introducing a fresh (ghost) variable x of an arbitrary base type, such as $x : \{\nu : \mathbb{1} \mid c\}$, where $\nu \notin \mathsf{FV}(c)$, and add it to the typing context. In our system, additional conjuncts can be added to the predicate ϕ directly.

Refinement predicates are shallowly embedded as Agda terms of type `Set`. We also define a substitution function in Agda which allows us to substitute the top-most variable in a predicate ϕ with an expression e:

```
_[_]s : ∀{Γ}{T} → (ϕ : ⟦ Γ ▸ T ⟧C → Set) → (e : Γ ⊢ T)
      → (⟦ Γ ⟧C → Set)
ϕ [ e ]s = ^ ϕ ˢ ⟦ e ⟧⊢
```

In Figure 6.8, we show the well-formedness rules for the refinement contexts and for the refinement types. They are checked by Agda's type system and are therefore implicit in the Agda formalisation. The typing rules can be found in Figure 6.9. Most of the typing rules are straightforward and work in a similar manner to their counterparts in a more traditional formalisation of refinement types. We only elaborate on a few of them.

Variables The VAR$^{\text{R}}$ rule infers the most precise type—the singleton type—for a variable x. In many other calculi (e.g. [Jhala and Vazou 2021; Knowles and Flanagan 2009; Ou et al. 2004]), a

subtly different from the normal function arrow → and in fact its semantics overlaps with that of the normal function arrow. In reading and understanding the rules, their difference can usually be dismissed.

$$\boxed{\hat{\Gamma}\ \mathbf{wf}}$$

$$\frac{\mathsf{FV}(\phi) \subseteq \mathsf{dom}(\Gamma)}{\Gamma; \phi\ \mathbf{wf}}\text{Ctx-Wf}$$

$$\boxed{\Gamma \vdash \{\nu : B \mid \psi\}\ \mathbf{wf}}$$

$$\frac{\mathsf{FV}(\psi) \subseteq \mathsf{dom}(\Gamma) \cup \{\nu\}}{\Gamma \vdash \{\nu : B \mid \psi\}\ \mathbf{wf}}\text{RefType-Wf}$$

Figure 6.8: Well-formedness rules for contexts and types

different variant of the selfification rule is used for variables:[5]

$$\frac{(x : \{\nu : B \mid \phi\}) \in \Gamma_\tau}{\Gamma_\tau \vdash x : \{\nu : B \mid \phi \wedge \nu \equiv x\}}\text{Self}$$

We choose not to include the ϕ in the inferred type of x, as such information can be recovered from the typing context via the subtyping rule SUB$^{\mathrm{R}}$.

Constants For value constants ((), true, false, ze) and function constants ($\oplus$, (,), π_1, π_2, su), the typing rules always infer the exact type for the result. As with the VAR$^{\mathrm{R}}$ rule, we do not carry over the refinement predicates in the premises to the inferred type in the conclusion. Again, no information is lost during this process, as they can be recovered later from the context when needed.

Function applications The typing rule for function applications is pretty standard. In the work of Knowles and Flanagan [2009], a compositional version of this rule is used instead. To summarise the idea, consider the typical function application rule, which has the following form:

$$\frac{\Gamma_\tau \vdash f : (x{:}\tau_1) \to \tau_2 \qquad \Gamma_\tau \vdash e : \tau_1' \qquad \Gamma_\tau \vdash \tau_1' \sqsubseteq \tau_1}{\Gamma_\tau \vdash f\ e : \tau_2[e/x]}\text{App}$$

In the refinement in the conclusion, the term e is substituted for x. This would circumvent the type abstraction τ_1' of e, exposing the implementation details of the argument to the resulting refinement type $\tau_2[e/x]$. It also renders the type $\tau_2[e/x]$ arbitrarily large due to the presence of e. To rectify this problem, Knowles and Flanagan [2009] propose the result type to be existential: $\exists x : \tau_1'.\, \tau_2$. Which application rule to include largely depends on the language design. We use the traditional one here and the compositional one later in this chapter to contrast the two.

[5]We use the subscript in Γ_τ to mean the more traditional $\overrightarrow{x_i : \tau_i}$ context, where each τ_i is a refinement type.

$$\boxed{\hat\Gamma \vdash_R \hat e : \tau}$$

$$\frac{(x : T) \in \Gamma}{\Gamma;\phi \vdash_R x : \{\nu : T \mid \nu = x\}}\,\textsf{VAR}^{\textsf{R}} \qquad \frac{}{\hat\Gamma \vdash_R () : \{\nu : \mathbb{1} \mid \nu = ()\}}\,\textsf{UNIT}^{\textsf{R}} \qquad \frac{b \in \{\text{true}, \text{false}\}}{\hat\Gamma \vdash_R b : \{\nu : \mathbb{2} \mid \nu = b\}}\,\textsf{TT}^{\textsf{R}}/\textsf{FF}^{\textsf{R}}$$

$$\frac{}{\hat\Gamma \vdash_R \text{ze} : \{\nu : \mathbb{N} \mid \nu = 0\}}\,\textsf{ZE}^{\textsf{R}} \qquad \frac{\hat\Gamma \vdash_R \hat e : \{\nu : \mathbb{N} \mid \psi\}}{\hat\Gamma \vdash_R \text{su}\ \hat e : \{\nu : \mathbb{N} \mid \nu = \text{suc}(\hat e)\}}\,\textsf{SU}^{\textsf{R}}$$

$$\frac{\begin{array}{c}\Gamma;\phi \vdash_R \hat c : \{\nu : \mathbb{2} \mid \psi'\} \\ \Gamma;\phi \wedge \hat c \vdash_R \hat e_1 : \{\nu : T \mid \psi\} \qquad \Gamma;\phi \wedge \neg\hat c \vdash_R \hat e_2 : \{\nu : T \mid \psi\}\end{array}}{\Gamma;\phi \vdash_R \textbf{if}\ \hat c\ \textbf{then}\ \hat e_1\ \textbf{else}\ \hat e_2 : \{\nu : T \mid \psi\}}\,\textsf{IF}^{\textsf{R}}$$

$$\frac{\begin{array}{c}\Gamma;\phi \vdash_R \hat e_1 : \{x : S \mid \xi\} \qquad \Gamma \vdash \{\nu : T \mid \psi\}\ \textbf{wf} \\ \Gamma, x : S;\phi \wedge \xi \vdash_R \hat e_2 : \{\nu : T \mid \psi\}\end{array}}{\Gamma;\phi \vdash_R \textbf{let}\ x = \hat e_1\ \textbf{in}\ \hat e_2 : \{\nu : T \mid \psi\}}\,\textsf{LET}^{\textsf{R}}$$

$$\frac{\hat\Gamma \vdash_R \hat e_1 : \{\nu : S \mid \psi_1\} \qquad \hat\Gamma \vdash_R \hat e_2 : \{\nu : T \mid \psi_2\}}{\hat\Gamma \vdash_R (\hat e_1, \hat e_2) : \{\nu : S \times T \mid \nu = (\hat e_1, \hat e_2)\}}\,\textsf{PRD}^{\textsf{R}}$$

$$\frac{\hat\Gamma \vdash_R \hat e : \{\nu : T_1 \times T_2 \mid \psi\} \qquad i \in \{1, 2\}}{\hat\Gamma \vdash_R \pi_i\ \hat e : \{\nu : T_i \mid \nu = \pi_i\ \hat e\}}\,\textsf{FST}^{\textsf{R}}/\textsf{SND}^{\textsf{R}}$$

$$\frac{\begin{array}{c}\Gamma;\phi \vdash_R \hat f : x{:}\{\nu : S \mid \xi\} \longrightarrow \{\nu : T \mid \psi\} \\ x \notin \text{Dom}(\Gamma) \qquad \Gamma;\phi \vdash_R \hat e : \{\nu : S \mid \xi\}\end{array}}{\Gamma;\phi \vdash_R \hat f\ \hat e : \{\nu : T \mid \psi[\hat e/x]\}}\,\textsf{APP}^{\textsf{R}}$$

$$\frac{\begin{array}{c}\text{ty}(\oplus) = B_1 \to B_1 \to B_2 \\ \hat\Gamma \vdash_R \hat e_1 : \{\nu : B_1 \mid \psi_1\} \qquad \hat\Gamma \vdash_R \hat e_2 : \{\nu : B_1 \mid \psi_2\}\end{array}}{\hat\Gamma \vdash_R \hat e_1 \oplus \hat e_2 : \{\nu : B_2 \mid \nu = \hat e_1 \oplus \hat e_2\}}\,\textsf{BOP}^{\textsf{R}}$$

$$\frac{\Gamma;\phi \vdash_R \hat e : \{\nu : S \mid \psi\} \qquad \Gamma, x : S;\phi \vDash \psi \Rightarrow \psi'}{\Gamma;\phi \vdash_R \hat e :: \{\nu : S \mid \psi'\}}\,\textsf{SUB}^{\textsf{R}}$$

$$\frac{\Gamma;\phi \vdash_R \hat e : \{\nu : S \mid \psi\} \qquad \Gamma \vDash \phi' \Rightarrow \phi}{\Gamma;\phi' \vdash_R \hat e : \{\nu : S \mid \psi\}}\,\textsf{WEAK}^{\textsf{R}}$$

$$\boxed{\hat\Gamma \vdash_R \hat f : x{:}\tau_1 \longrightarrow \tau_2}$$

$$\frac{\Gamma, x : S;\xi \vdash_R \hat e : \{\nu : T \mid \psi\}}{\Gamma;\phi \vdash_R \lambda x.e : x{:}\{\nu : S \mid \xi\} \longrightarrow \{\nu : T \mid \psi\}}\,\textsf{FUN}^{\textsf{R}}$$

Figure 6.9: Static semantics of λ^R

Jhala and Vazou [2021] stick to the regular function application rule, but with some extra restrictions. They require the function argument to be in A-normal form (ANF) [Sabry and Matthias Felleisen 1992], i.e. the argument being a variable instead of an arbitrary expression. This reduces the load on the SMT-solver and helps them remain decidable in the refinement logic. We do not need the ANF restriction in our system for decidability, and the argument term will always be fully reduced in Agda while conducting the meta-theoretical proofs.

Let-bindings In the LET$^{\mathrm{R}}$ rule, the well-formedness condition $\Gamma \vdash \{\nu : T \mid \psi\}$ **wf** implies that ψ does not mention the locally-bound variable x, preventing the local binder from creeping into the resulting type of the let-expression. The let-binding and the function application rule give similar power in reasoning, thanks to the SUB$^{\mathrm{R}}$ rule. The structure of the proofs are slightly different though because the SUB$^{\mathrm{R}}$ nodes need to be placed at different positions.

Subtyping and weakening Key to a refinement type system is the subtyping relation between types. Typically, the (partly syntactic) subtyping judgement looks like:

$$\frac{\Gamma_\tau, x : B \vDash \phi \Rightarrow \phi'}{\Gamma_\tau \vdash \{\nu : B \mid \phi\} \sqsubseteq \{\nu : B \mid \phi'\}}\,\textsf{Sub}$$

$$\frac{\Gamma_\tau \vdash \sigma_2 \sqsubseteq \sigma_1 \qquad \Gamma_\tau, x : \sigma_2 \vdash \tau_1 \sqsubseteq \tau_2}{\Gamma_\tau \vdash x : \sigma_1 \to \tau_1 \sqsubseteq x : \sigma_2 \to \tau_2}\,\textsf{Sub-Fun}$$

In our language, since we only support first-order functions, the subtyping rule for functions is not needed. It can be achieved by promoting the argument and the result of a function application separately. Since function types are excluded from the universe **U**, subtyping can be defined on the entire domain of **U**, and in a fully semantic manner. We directly define the subtyping-style rules (SUB$^{\mathrm{R}}$, WEAK$^{\mathrm{R}}$) in terms of a logical entailment: $\phi \vDash \psi \Rightarrow \psi' \overset{def}{=} \forall \gamma\, x.\, \phi\, \gamma \wedge \psi\, (\gamma, x) \to \psi'\, (\gamma, x)$.

If we allowed refinement predicates over function spaces, it would require a full semantic subtyping relation [Bierman et al. 2010; Castagna and Frisch 2005] that also works on the function space. This has been shown to be possible, e.g. interpreting the types in a set-theoretic fashion as in Castagna and Frisch [2005]'s work. It is however far from trivial to encode a set theory that can be used for the interpretation of functions in Agda's type system (e.g. [Kono 2022] is an attempt to define Zermelo–Fraenkel set theory **ZF** in Agda).

The subtyping rule (SUB$^{\mathrm{R}}$) and the weakening rule (WEAK$^{\mathrm{R}}$) roughly correspond to the left- and right- consequence rules of Hoare logic respectively. All the subtyping in our system is explicit. For instance, unlike rule App above, in order to apply a function, we have to explicitly promote the argument with a SUB$^{\mathrm{R}}$ node, if its type is not already matching the argument type expected by the function. As a notational convenience, in the typing rules we write $\hat{\Gamma} \vdash_R \hat{e} :: \tau$ to mean $\hat{\Gamma} \vdash_R \hat{e} :: \tau : \tau$, as the inferred type is always identical to the one that is promoted to.

Comparing the SUB$^{\text{R}}$ rule with the right-consequence rule in Hoare logic, which reads

$$\frac{\{P\}\,s\,\{Q\} \qquad Q \to Q'}{\{P\}\,s\,\{Q'\}}\,\textsf{Cons-R}$$

we can notice that in the SUB$^{\text{R}}$ rule, the implication says $\phi \vDash \psi \Rightarrow \psi'$. In Cons-R, in contrast, the precondition P is not involved in the implication. This is because of the nature of the underlying language. In an imperative language to which the Hoare logic is applied, P and Q are predicates over the program states. A variable assignment statement or reference update operation will change the state, and therefore invalidate the proposition P. The newly established proposition about the updated program state is then captured by Q after the execution of the statement s. In our purely functional language λ^R, ϕ is a predicate over the typing context Γ, and a typing judgement does not invalidate ϕ. Moreover, in practice, when assigning a refinement type to an expression, the refinement predicate typically only concerns the term being typed, and does not talk about variables in Γ that are not directly relevant. Therefore it is technically possible not to require ϕ in the implication, but it renders the system more cumbersome to use.

As for weakening, in contrast to the more canonical structural rule [Lehmann and Tanter 2016]:

$$\frac{\vdash \Gamma_{\tau_1}, \Gamma_{\tau_2}, \Gamma_{\tau_3} \qquad \Gamma_{\tau_1}, \Gamma_{\tau_3} \vdash e : \tau}{\Gamma_{\tau_1}, \Gamma_{\tau_2}, \Gamma_{\tau_3} \vdash e : \tau}$$

the WEAK$^{\text{R}}$ rule only changes the predicates in the context and does not allow for adding new binders to the simply-typed context Γ. It compares favourably to those with a more syntactic refinement-typing context. For a traditional refinement-typing context $\overrightarrow{x_i : \tau_i}$, if the weakening lemma is to be defined in a general enough manner to allow weakening to happen arbitrarily in the middle of the context, some tactics will be required to syntactically rearrange the context to make the weakening rule applicable. Our weakening rule is purely semantic and therefore does not require syntactic rewriting before it can be applied.

Functions The FUN$^{\text{R}}$ rule can be used to construct a λ-abstraction under any context Γ and the λ-term does not need to be closed. The function body $\hat{e}$ is typed under the extended context $\Gamma, x : S$, but the predicate part does not include ϕ. This does not cause any problems because ξ is itself a predicate over the context and the function argument, and also if more information about the context needs to be drawn, it can be done via the SUB$^{\text{R}}$ rule at a later stage.

The pen-and-paper formalisation above leaves some aspects informal for presentation purposes. One instance of the discrepancy is that, when we type the term $\hat{e}_1 + \hat{e}_2$, the resulting predicate is $\nu = \hat{e}_1 + \hat{e}_2$. What has been implicit in the rules is the reflection of object terms into the logical system.

In our formal development, the underlying logical system is Agda's type system, therefore we want to reflect the refinement-typed term language into the Agda land. We do it as a two-step process: we first map the refinement-typed language to the simply-typed language by erasure, and then reflect the simply-typed terms into logic using the already-defined interpretation function $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$, with which we interpret the object language as Agda terms.

The erasure function $\ulcorner\cdot\urcorner^{\mathsf{R}}$ removes all refinement type information from a refinement-typed term (also, typing tree) and returns a corresponding simply-typed term (also, typing tree). Essentially, the erasure function removes the refinement predicates, and any explicit upcast ($\mathsf{SUB}^{\mathsf{R}}$) nodes from the typing tree.

Now we can define the deep syntax of the λ^R language along with its typing rules in Agda. When an expression $\hat{e}$ in the object language has an Agda type `Γ { ϕ }⊢ T { ψ }`, it means that under context Γ which satisfies the precondition ϕ, the expression $\hat{e}$ can be assigned a refinement type $\{\nu : T \mid \psi\}$. For functions, we have `Γ { ϕ }⊢ S { ξ }⟶ T { ψ }` which keeps track of the predicates on the context Γ, on the argument and on the result respectively. The datatypes and the erasure function $\ulcorner\cdot\urcorner^{\mathsf{R}}$ are inductive-recursively defined. The Agda definition of the syntax (and the typing rules) of λ^R is given in Figure 6.9.

The context weakening rule $\mathsf{WEAK}^{\mathsf{R}}$ in Figure 6.9 is in fact admissible in our system, therefore it is not included as a primitive construct in the formal definition of the language.

Lemma 6.1 (Weakening is admissible)**.** *For any typing tree $\Gamma;\phi \vdash_R \hat{e} : \tau$, if $\phi' \Rightarrow \phi$ under any semantic environment γ that respects Γ, then there exists a typing tree with the stronger context $\Gamma;\phi'$, such that $\Gamma;\phi' \vdash_R \hat{e}' : \tau$ and $\ulcorner\hat{e}\urcorner^R = \ulcorner\hat{e}'\urcorner^R$.*

Continuing on the `f₀` function defined in Section 6.3, if we want to lift it to a function definition in λ^R, we will need to add refinements and insert explicit upcast nodes:

$$f_0^{\mathsf{R}} : (x : \{\nu : \mathbb{N} \mid \nu < 2\}) \longrightarrow \{\nu : \mathbb{N} \mid \nu < 4\}$$
$$f_0^{\mathsf{R}} = \lambda x.\, (x + 1 :: \{\nu : \mathbb{N} \mid \nu < 4\})$$

In Agda, it is defined as follows:

```
f₀ᴿ : `E´ { ᵏ T }⊢ `ℕ´ { (_< 2) ∘ proj₂ }⟶
                   `ℕ´ { (_< 4) ∘ proj₂ }
f₀ᴿ = FUNᴿ (SUBᴿ (BOPᴿ [+] (VARᴿ top) ONEᴿ) _ prf)
```

The upcast node `SUBᴿ` needs to be accompanied by an evidence (i.e. an Agda proof term `prf` whose definition is omitted) to witness the semantic entailment $x < 2 \vDash \nu = x + 1 \Rightarrow \nu < 4$. To demonstrate the function application of `f₀ᴿ`, we define the following expression:

$$ex_0^{\mathsf{R}} : \{\nu : \mathbb{N} \mid \nu < 5\}$$
$$ex_0^{\mathsf{R}} = f_0^{\mathsf{R}}\ (1 :: \{\nu : \mathbb{N} \mid \nu < 2\}) :: \{\nu : \mathbb{N} \mid \nu < 5\}$$

The inner upcast is for promoting the argument 1, which is inferred the exact type $\{\nu : \mathbb{N} \mid \nu = 1\}$, to match f_0^{R}'s argument type. The outer upcast is for promoting the result of the application from

```
mutual
  data _{_}⊢_{_} (Γ : Cx) : (⟦ Γ ⟧C → Set) → (T : U)
                             → (⟦ Γ ▸ T ⟧C → Set) → Set₁ where
    VARᴿ  : ∀{ϕ}{T} → (i : Γ ∋ T) → Γ { ϕ }⊢ T { (λ (γ , ν) → ν ≡ ⟦ i ⟧∋ γ) }
    UNITᴿ : ∀{ϕ} → (Γ { ϕ }⊢ `1´ { (λ γ → proj₂ γ ≡ tt) })
    TTᴿ   : ∀{ϕ} → Γ { ϕ }⊢ `2´ { (λ γ → proj₂ γ ≡ true ) }
    FFᴿ   : ∀{ϕ} → Γ { ϕ }⊢ `2´ { (λ γ → proj₂ γ ≡ false) }
    ZEᴿ   : ∀{ϕ} → Γ { ϕ }⊢ `ℕ´ { (λ γ → proj₂ γ ≡ 0 ) }
    SUᴿ   : ∀{ϕ}{ψ}
          → (n : Γ { ϕ }⊢ `ℕ´ { ψ })
          → Γ { ϕ }⊢ `ℕ´ { (λ γ → proj₂ γ ≡ suc (⟦ ⌜ n ⌝ᴿ ⟧⊢ (proj₁ γ))) }
    IFᴿ   : ∀{ϕ}{ξ}{T}{ψ}
          → (c : Γ { ϕ }⊢ `2´ { ξ })
          → (Γ { (λ γ → ϕ γ × ⟦ ⌜ c ⌝ᴿ ⟧⊢ γ ≡ true ) }⊢ T { ψ })
          → (Γ { (λ γ → ϕ γ × ⟦ ⌜ c ⌝ᴿ ⟧⊢ γ ≡ false) }⊢ T { ψ })
          → Γ { ϕ }⊢ T { ψ }
    LETᴿ  : ∀{S}{T}{ϕ}{ξ : ⟦ Γ ▸ S ⟧C → Set}{ψ : ⟦ Γ ▸ T ⟧C → Set}
          → (Γ { ϕ }⊢ S { ξ })
          → ((Γ ▸ S) { (λ (γ , s) → ϕ γ × ξ (γ , s)) }⊢ T { (λ ((γ , _) , ν) → ψ (γ , ν)) })
          → Γ { ϕ }⊢ T { ψ }
    PRDᴿ  : ∀{S T}{ϕ}{ψ₁ ψ₂}
          → (e₁ : Γ { ϕ }⊢ S { ψ₁ })
          → (e₂ : Γ { ϕ }⊢ T { ψ₂ })
          → Γ { ϕ }⊢ S `×´ T { (λ (γ , ν) → ν ≡ < ⟦ ⌜ e₁ ⌝ᴿ ⟧⊢ , ⟦ ⌜ e₂ ⌝ᴿ ⟧⊢ > γ) }
    FSTᴿ  : ∀{S T}{ϕ}{ψ}
          → (e : Γ { ϕ }⊢ S `×´ T { ψ })
          → Γ { ϕ }⊢ S { (λ (γ , ν) → ν ≡ proj₁ (⟦ ⌜ e ⌝ᴿ ⟧⊢ γ)) }
    SNDᴿ  : ∀{S T}{ϕ}{ψ}
          → (e : Γ { ϕ }⊢ S `×´ T { ψ })
          → Γ { ϕ }⊢ T { (λ (γ , ν) → ν ≡ proj₂ (⟦ ⌜ e ⌝ᴿ ⟧⊢ γ)) }
    APPᴿ  : ∀{S}{T}{ϕ}{ξ}{ψ}
          → (f : Γ { ϕ }⊢ S { ξ }⟶ T { ψ })
          → (e : Γ { ϕ }⊢ S { ξ })
          → Γ { ϕ }⊢ T { (λ (γ , t) → ψ ((γ , ⟦ ⌜ e ⌝ᴿ ⟧⊢ γ) , t)) }
  BOPᴿ : ∀{ϕ}(o : ⊕){ψ₁ ψ₂}
         → (e₁ : Γ { ϕ }⊢ →⊕ o { ψ₁ })
         → (e₂ : Γ { ϕ }⊢ →⊕ o { ψ₂ })
         → Γ { ϕ }⊢ ⊕→ o { (λ (γ , ν) → ν ≡ (⟦ ⌜ e₁ ⌝ᴿ ⟧⊢ γ ⟦ o ⟧⊢⊕ ⟦ ⌜ e₂ ⌝ᴿ ⟧⊢ γ)) }
  SUBᴿ   : ∀{T}{ϕ : ⟦ Γ ⟧C → Set}{ψ : ⟦ Γ ▸ T ⟧C → Set}
         → (e : Γ { ϕ }⊢ T { ψ })
         → (ψ′ : ⟦ Γ ▸ T ⟧C → Set)
         → ϕ ⊨ ψ ⇒ ψ′
         → Γ { ϕ }⊢ T { ψ′ }


data _{_}⊢_{_}⟶_{_} (Γ : Cx) : (⟦ Γ ⟧C → Set) → (S : U)
                              → (⟦ Γ ▸ S ⟧C → Set) → (T : U)
                              → (⟦ Γ ▸ S ▸ T ⟧C → Set) → Set₁ where
  FUNᴿ   : ∀{S T}{ϕ}{ξ}{ψ}
         → Γ ▸ S { ξ }⊢ T { ψ }
         → Γ { ϕ }⊢ S { ξ }⟶ T { ψ }
```

Figure 6.9: The syntax of the λ^R language

$\{\nu : \mathbb{N} \mid \nu < 4\}$ to $\{\nu : \mathbb{N} \mid \nu < 5\}$. In Agda, two proof terms need to be constructed for the upcast nodes in order to show that the argument and the result of the application are both type correct:

```
ex₀ᴿ : `E′ { ᵏ T }⊢ `ℕ′ { (_< 5) ∘ proj₂ }
ex₀ᴿ = SUBᴿ (APPᴿ {ψ = (_< 4) ∘ proj₂} f₀ᴿ
                  (SUBᴿ ONEᴿ _ λ _ _ _ → s≡1⇒s<2))
            _ λ _ _ _ → t<4⇒t<5
```

6.5 Meta-Properties of λ^R

Instead of proving the textbook type soundness theorems (progress and preservation) [Harper 2016; Wright and M. Felleisen 1994] that rest upon subject reduction, we instead get for free the semantic type soundness theorem à la Milner [Milner 1978] for the base language λ^B because of the way the term language is embedded and interpreted in Agda.

Theorem 6.2 (Semantic soundness). *If $\Gamma \vdash e : T$ and the semantic environment γ respects the typing environment Γ, then $\vDash \mathscr{E}[\![e]\!]_{Tm}\gamma : T$.*

We take the same semantic approach to type soundness and establish the the refinement soundness theorem for λ^R. We use the notation $\phi \vDash \psi$ for the semantic entailment relation in the underlying logic, which, in our case, is Agda's type system. To relate a semantic environment γ to a refinement typing context $\hat{\Gamma}$, we proceed with the following definitions.

Definition 6.1. *A semantic environment γ satisfies a predicate ϕ, if $FV(\phi) \subseteq dom(\gamma)$ and $\varnothing \vDash \phi\ \gamma$. We write $\phi\ \gamma$ to mean $\phi[\overline{\gamma(x_i)/x_i}]$ for all free variables x_i in ϕ.*

Definition 6.2. *A semantic environment γ respects a refinement typing context $\Gamma; \phi$ if γ respects Γ and satisfies ϕ.*

We define what it means for a denotational value to possess a refinement type, and extend the notion of semantic typing to refinement types.

Definition 6.3. *A value v posesses a refinement type $\{\nu : T \mid \psi\}$, written $\vDash v : \{\nu : T \mid \psi\}$, if $\vDash v : T$ and $\varnothing \vDash \psi[v/\nu]$.*

Definition 6.4 (Refinement semantic typing)**.** *$\hat{\Gamma} \vDash \hat{e} : \tau$ if $\vDash \mathscr{E}[\![\ulcorner \hat{e} \urcorner^R]\!]_{Tm}\gamma : \tau$ for all γ that respects $\hat{\Gamma}$.*

With the notion of refinement semantic typing, we can state the refinement (semantic) type soundness theorem as follows.

Theorem 6.3 (Refinement soundness)**.** *If $\hat{\Gamma} \vdash_R \hat{e} : \tau$ then $\hat{\Gamma} \vDash e : \tau$.*

Proof. By induction on the structure of $\hat{\Gamma} \vdash_R \hat{e} : \tau$. □

The converse of this theorem is also true. It states the completeness of our refinement type system with respect to semantic typing.

Theorem 6.4 (Refinement completeness)**.** *If $\Gamma \vdash e : T$ and for all semantic context γ that respects $\hat{\Gamma}$, $\vDash \mathscr{E}[\![e]\!]_{Tm}\gamma : \{\nu : T \mid \psi\}$, then there exists a refinement typing $\hat{\Gamma} \vdash_R \hat{e} : \{\nu : T \mid \psi\}$ such that $\ulcorner \hat{e} \urcorner^R = e$.*

Proof. By induction on the structure of $\Gamma \vdash e : T$. □

Note that for the completeness theorem, since we only need to construct one such refinement typed expression (or equivalently, typing tree), the proof is not unique, in light of the SUB^R and WEAK^R rules.

With the refinement soundness and completeness theorems, we can deduce a few direct but useful corollaries.

Corollary 6.5. *Refinement soundness holds for closed terms.*

Proof. By setting $\Gamma = \varnothing$ in Theorem 6.3. □

Corollary 6.6. *For refinement typing judgements, the predicate ϕ over the context is an invariant, namely, $\Gamma; \phi \vdash_R \hat{e} : \{\nu : T \mid \lambda\nu.\,\phi\}$.*

Proof. A direct consequence of Theorem 6.4. □

Corollary 6.7 (Consistency). *It is impossible to assign a void refinement type to an expression* $\hat{\Gamma} \vdash_R \hat{e} : \{\nu : T \mid \mathsf{false}\}$*, if there exists a semantic environment* γ *that respects* $\hat{\Gamma}$*.*

Proof. A direct consequence of Theorem 6.3. □

6.6 Typechecking λ^R by Weakest Precondition

A simple typechecking algorithm can be given to the λ^R language, in terms of the weakest precondition predicate transformer [Dijkstra 1975]. Since all functions in λ^R are required to be annotated with types, it is possible to assume the postcondition of a function and compute the weakest precondition. If the given precondition in the type signature entails the weakest precondition, we know that the program is well-typed according to the specification (i.e. the type signature).

In an imperative language, when a variable x gets assigned, the Hoare triple is $\{Q[e/x]\}\ x := e\ \{Q\}$, which means that the weakest precondition can be obtained by simply substituting the variable x in Q by the expression e. The Hoare logic style typing judgement $\Gamma\{\phi\} \vdash e : T\{\psi\}$ in our purely functional language can be considered as assigning the value of e to a fresh variable ν. Therefore the weakest precondition function $\mathrm{wp}\ \psi\ e$ can be defined analogously as a substitution of the top binder ν in ψ with the value of e, resulting in a predicate over a semantic environment γ that respects Γ.

```
wp : ∀{Γ}{T} → (⟦ Γ ▸ T ⟧C → Set) → Γ ⊢ T → (⟦ Γ ⟧C → Set)
wp ψ e = ψ [ e ]s
```

The completeness and soundness of the wp function with respect to the typing rules of λ^R are direct corollaries of the refinement soundness and completeness theorems (Theorem 6.3 and Theorem 6.4) respectively.

Theorem 6.8 (Completeness of wp w.r.t. λ^R typing). *If* $\Gamma;\phi \vdash_R \hat{e} : \{\nu : T \mid \psi\}$*, then* $\phi\ \gamma \Rightarrow$ *wp* $\psi\ \ulcorner\hat{e}\urcorner^R\ \gamma$ *for any semantic environment* γ *that respects* Γ*.*

Theorem 6.9 (Soundness of wp w.r.t. λ^R typing). *For an expression* $\Gamma \vdash e : T$ *in* λ^B *and a predicate* ψ*, there must exist a type derivation* $\Gamma;$ *wp* $\psi\ e \vdash_R \hat{e} : \{\nu : T \mid \psi\}$ *such that* $\ulcorner\hat{e}\urcorner^R = e$*.*

The wp function checks that, when a type signature is given to an expression e, it can infer the weakest precondition under which e is typeable. Writing in natural deduction style, the algorithmic typing rule looks like:

$$\frac{\cdots}{\Gamma;\phi \mapsto e : \{\nu : T \mid \psi\}}$$

Contrary to regular algorithmic typing rules (e.g. in bidirectional typing [Dunfield and Krishnaswami 2021]), where the context and the expression are typically inputs, and the type is either input or output depending on whether it performs type checking or synthesise, in our formulation, the expression and the type are the inputs and (the predicate part of) the context is the output.

The wp function only checks whether an expression is well-typed by inferring the weakest context, but it does not elaborate the typing tree by annotating each sub-expression with a type, nor does it automatically construct proof terms. Despite the limitation, this method can still be applied to program verification tasks in which the exact refinement typing tree need not be constructed, or when the automatic construction of proof terms is not required. Cogent is a natural candidate application for this typing algorithm. In Cogent's verification framework, a fully elaborated refinement typing tree will not be necessary, and the functional correctness of a system is manually proved in Isabelle/HOL. Since proof engineers are already engaged, we do not have to rely on an SMT-solver to construct the all the proof objects. Proof engineers can manually discharge the proof obligations, especially the difficult ones that an SMT-solver cannot efficiently solve. Therefore the logic supported by the refinement predicates does not necessarily need to be restricted to a decidable fraction, as is required by many refinement type systems. A decidable logic is a bonus, but not a prerequisite.

6.7 Function Contracts with λ^C

As we have seen, wp is easy to define and works uniformly across all terms. Yet it has a very unfortunate drawback—it is defined on the simply-typed language and is oblivious to the function signatures, and consequently does not preserve function contracts. This is however not the only problem. In the definition of wp, when it is applied to an expression containing function applications, instead of β-reducing the applications, it should inspect the function's type signature and produce verification conditions for the function contracts. To this end, the denotation function $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$ and subsequently the refinement typing rules also need to be revised.

We define a variant of the language λ^C, in which the function contracts are respected.[6] It is worth mentioning that the language is not yet compositional in the sense of [Knowles and Flanagan 2009], as the weakest precondition computation still draws information from the implementation of expressions, which we will see later in this section.

6.7.1 The λ^C Language

The syntax of λ^C is the same as λ^R, and its typing rules are very similar to those of λ^R as well. Despite the fact that the λ^C language will reflect its term language to Agda in a slightly different way (more details in Section 6.7.2), we only make two changes in the typing rules for λ^C:

[6]The superscript C in λ^C means "contract".

$$\boxed{\hat{\Gamma} \vdash_C \hat{e} : \tau}$$

$$\frac{\Gamma; \phi \vdash_C \hat{e}_1 : \{x : S \mid \xi\} \qquad \Gamma \vdash \{\nu : T \mid \psi\}\ \mathbf{wf} \qquad \Gamma, x : S; \phi \wedge x = \hat{e}_1 \vdash_C \hat{e}_2 : \{\nu : T \mid \psi\}}{\Gamma; \phi \vdash_C \mathbf{let}\ x = \hat{e}_1\ \mathbf{in}\ \hat{e}_2 : \{\nu : T \mid \psi\}}\ \text{LET}^{\text{C}}$$

$$\frac{\Gamma; \phi \vdash_C \hat{f} : x{:}\{\nu : S \mid \xi\} \longrightarrow \{\nu : T \mid \psi\} \qquad x \notin \mathsf{Dom}(\Gamma) \qquad \Gamma; \phi \vdash_C \hat{e} : \{\nu : S \mid \xi\}}{\Gamma; \phi \vdash_C \hat{f}\ \hat{e} : \{\nu : T \mid \exists x : \xi[x/\nu].\psi\}}\ \text{APP}^{\text{C}}$$

As suggested by Knowles and Flanagan [2009], the result of a function application can be made existential for retaining the abstraction over the function's argument. This idea is implemented as the rule APP$^{\text{C}}$. The choice of using this favour of function application is purely incidental—offering a contrast to the other variant used in λ^R. In practice, we believe both rules have their place in a system. The existential version is significantly limited in the conclusions that it can lead to, and renders some basic functions useless. For instance, we define an inc function as follows:

$$\mathsf{inc} : (x : \mathbb{N}) \longrightarrow \{\nu : \mathbb{N} \mid \nu = x + 1\}$$
$$\mathsf{inc} = \lambda x.\, x + 1$$

The function's output is already giving the exact type of the result. With the APP$^{\text{C}}$ rule, we cannot deduce that inc 0 is 1, which is intuitively very obvious. In fact, if the input type of inc is kept unrefined, then we can hardly draw any conclusion about the result of this function. This behaviour can be problematic when users define, say, arithmetic operations as functions.

The LET$^{\text{C}}$ rule differs from LET$^{\text{R}}$ in a way that the precondition of the expression e_2 is ϕ in conjunction with the exact refinement $x = \hat{e}_1$ for the new binder x instead of the arbitrary postcondition ξ of e_1. Intuitively, because the exact type of the local binder is added to the context when typechecking e_2, when we compute the weakest precondition of the let-expression (later in Figure 6.10), we can assume the trivial postcondition $\lambda\_.\,\mathsf{true}$ of e_1. This makes the LET$^{\text{C}}$ rule significantly easier to work with. Unfortunately, we do not yet have formal evidence to conclude with full confidence whether the LET$^{\text{R}}$ rule can be used in this system instead of LET$^{\text{C}}$ or not. Also note that, LET$^{\text{C}}$ gives different reasoning power than APP$^{\text{C}}$ does, and they nicely complement each other in the system.

6.7.2 Annotated Base Language λ^A

To typecheck of λ^C, we define λ^A, a variant of the base language λ^B. It differs in that the functions are accompanied by type signatures. We denote function expressions in λ^A as $f :: (x : \xi) \longrightarrow \psi$, instead of a bare unrefined f.

To establish the connection between λ^C and λ^A, an erasure function $\ulcorner \cdot \urcorner^{\text{C}}$ is defined, taking a λ^C term to the corresponding λ^A term. It preserves the function's type annotations in λ^C, so that we know that when a λ^A term is typed, the functions are typed in accordance with their type signatures.

One reason why λ^R does not preserve the function contracts is because the way we interpret function calls. Imagine an expression $((f :: (x : \xi) \longrightarrow \psi)\ 1) + 2$ in the language λ^C where $f \equiv \lambda x. x + 1$, $\xi \equiv \lambda x. x < 2$ and $\psi \equiv \lambda \nu. \nu < 4$, which is well-typed. Ideally, the only knowledge that we can learn about the function application should be drawn from its type signature, namely $\lambda \nu. \nu < 4$ here. Therefore, the most precise type we can assign to the whole expression is $\{\nu : \mathbb{N} \mid \nu < 6\}$. However, according to the typing rule ADD$^{\mathrm{C}}$, the inferred refinement predicate of the result of the addition will be an Agda term $\mathscr{E}[\![f\ 1]\!]_{\mathsf{Tm}}\gamma + \mathscr{E}[\![2]\!]_{\mathsf{Tm}}\gamma$ for any γ. As the predicate reduces in Agda, it means that we can in fact conclude that the result is equal to 4, which is more precise than what the function contract tells us—we again lost the abstraction over f.

To fix the problem, we revise the definition of $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$. Instead of interpreting functions as their Agda shallow embedding, we `postulate` the interpretation of functions as δ:

```
postulate
  δ : ∀{Γ}{S T}{ξ}{ψ} → Γ ⊢ᴬ S { ξ }⟶ T { ψ } → ⟦ Γ ⟧C → ⟦ S ⟧τ → ⟦ T ⟧τ
```

It allows us to reflect functions in the object language into the logic as uninterpreted functions. In the example above, it will stop the shallow postcondition from reducing to 4, retaining a symbolic representation of the function f. We define a new interpretation function $\mathscr{E}[\![\cdot]\!]^A_{\mathsf{Tm}}$ for λ^A terms (`⟦_⟧⊢ᴬ` and `⟦_⟧⊢ᴬ⃗` in the Agda formalisation). It is defined in the same way as $\mathscr{E}[\![\cdot]\!]_{\mathsf{Tm}}$, with the exception of functions:

```
⟦ f ⟧⊢ᴬ⃗ = δ f
```

On the other hand, when we type any expressions in the language λ^C, we need to add the known function contracts to the precondition ϕ. The function contract can be extracted automatically by a `mkC` function defined as follows:

```
mkC : ∀{Γ}{S T}{ξ}{ψ} → Γ ⊢ᴬ S { ξ }⟶ T { ψ } → Set
mkC {Γ = Γ}{S = S}{ξ = ξ}{ψ = ψ} f =
  {γ : ⟦ Γ ⟧C} → (x : ⟦ S ⟧τ) → ξ (γ , x) → ψ ((γ , x) , δ f γ x)
```

6.7.3 Typechecking λ^A

In order to typecheck λ^A, which is a language that is already well-typed with respect to simple types, and all functions are annotated with refinement types, we want to have a similar deterministic procedure as we had in Section 6.6. Unfortunately, in the presence of the function boundaries, the weakest precondition computation cannot be done simply by substituting in the expressions.

We borrow the ideas from computing weakest preconditions for imperative languages with loops. Specifically, we follow the development found in Nipkow and Klein [2014, §12.4]'s book. In standard Hoare logic, it is widely known that the loop-invariant for a WHILE-loop cannot be computed using the weakest precondition function wp [Dijkstra 1975], as the function is recursive and may not terminate. In Nipkow and Klein [2014]'s work, for Isabelle/HOL to deterministically generate the verification condition for a Hoare triple, it requires the users to provide annotations for loop-invariants. It then divides the standard wp function into two functions: pre and vc. The former computes the weakest precondition nearly as wp, except that in the case of a WHILE-loop, it returns the annotated invariant immediately. The latter then checks that the provided invariants indeed make sense. Intuitively, for a WHILE-loop, it checks that the invariant I together with the loop condition implies the precondition of the loop body, which needs to preserve I afterwards, and that I together with the negation of the loop-condition implies the postcondition. In all other cases, the vc function simply recurses down the sub-statements and aggregates verification conditions.

Although there is no recursion—the functional counterparts to loops of an imperative language—in our language, the situation with functions is somewhat similar to WHILE-loops. We also cannot compute the weakest precondition according to the expressions, but have to rely on user annotations, for a different reason. We can also divide the wp computation into pre and vc. The function pre immediately returns the precondition of a function, which is the refinement predicate of the argument type. Then vc additionally validates the provided function signatures. In particular, we need to check that in a function application: (1) the function's actual argument is of a supertype to the prescribed input type; (2) the function's prescribed output type implies the postcondition of the function application inferred from the program context. Additionally, vc needs to recurse down the syntax tree and gather verification conditions from sub-expressions, and, in particular, descend into the function definition to check that it meets the given type signature. The definitions of the pre and the vc functions are shown in Figure 6.10 and Figure 6.11 respectively.[7]

Unlike the development in the book of Nipkow and Klein [2014], in our language λ^A, the definition of pre deviates from wp by quite a long way. For example, the typing rule for su looks like:

$$\frac{\Gamma; \phi \vdash_C : \hat{e} : \{\nu : \mathbb{N} \mid \xi\}}{\Gamma; \phi \vdash_C : \mathsf{su}\ \hat{e} : \{\nu : \mathbb{N} \mid \nu = \mathsf{suc}\ \hat{e}\}}\ \textsf{SU}^{\textsf{C}}$$

Intuitively, when we run the wp backwards on su $\hat{e}$ with a postcondition ψ, it results in $\psi[\mathscr{E}[\![\mathsf{suc}\ \hat{e}]\!]^A_{\mathsf{Tm}}\gamma/\nu]$ for a semantic environement γ. The inferred refinement ξ of $\hat{e}$ in the premise is arbitrary and appears to be irrelevant to the computation of the weakest precondition of the whole term. Therefore we can set ξ to be the trivial refinement (true) and there is nothing to be assumed about the context to refine $\hat{e}$. This is however not the case in the presence of

[7] $\cap$ is the intersection of predicates defined in Agda's standard library as: $P \cap Q = \lambda\gamma \to P\,\gamma \times Q\,\gamma$.

```
pre : ∀{Γ}{T}(ψ : ⟦ Γ ▸ T ⟧C → Set) → (e : Γ ⊢ᴬ T) → (⟦ Γ ⟧C → Set)
pre→ : ∀{Γ}{S T}{ξ}{ψ} → Γ ⊢ᴬ S { ξ }⟶ T { ψ } → (⟦ Γ ▸ S ⟧C → Set)

pre ψ (SUᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ SUᴬ e ]sᶜ
pre ψ (IFᴬ c e₁ e₂) = pre (ᵏ ⊤) c
  ∩ (if_then_else_ ∘ ⟦ c ⟧⊢ᴬ) ˢ pre ψ e₁ ˢ pre ψ e₂
pre ψ (LETᴬ e₁ e₂) = pre (ᵏ ⊤) e₁
  ∩ ^ (pre (λ ((γ , _) , t) → ψ (γ , t)) e₂) ˢ ⟦ e₁ ⟧⊢ᴬ
pre ψ (PRDᴬ e₁ e₂) = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ PRDᴬ e₁ e₂ ]sᶜ
pre ψ (FSTᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ FSTᴬ e ]sᶜ
pre ψ (SNDᴬ e) = pre (ᵏ ⊤) e ∩ ψ [ SNDᴬ e ]sᶜ
pre _ (APPᴬ {ξ = ξ}{ψ = ψ} f e) = pre ξ e
pre ψ (BOPᴬ o e₁ e₂) = pre (ᵏ ⊤) e₁ ∩ pre (ᵏ ⊤) e₂ ∩ ψ [ BOPᴬ o e₁ e₂ ]sᶜ
pre ψ e = ψ [ e ]sᶜ -- It's just subst for the rest

pre→ {ξ = ξ}{ψ = ψ} (FUNᴬ e) = ξ ∩ pre ψ e
```

Figure 6.10: The Agda definition of pre

function contracts. In general, a trivial postcondition does not entail a trivial precondition: pre ϕ $(\lambda\_.\,\text{true})$ $\hat{e} \neq (\lambda\_.\,\text{true})$. For instance, if $\hat{e}$ is a function application, then we also need to compute the weakest precondition for the argument to satisfy the contract.

Our vc function also differs slightly from its counterpart in the imperative setting: it additionally takes the precondition as an argument. This is because in a purely functional language, we do not carry over all the information in the precondition to the postcondition, as the precondition is an invariant (recall that in the subtyping rule SUB$^{\mathsf{R}}$, the entailment is $\phi \vDash \psi \Rightarrow \psi'$).

To see it in action, we consider the following definitions again:

$$f_0^{\mathsf{A}} = (\lambda x.\, x+1) :: \{\nu : \mathbb{N} \mid \nu < 2\} \longrightarrow \{\nu : \mathbb{N} \mid \nu < 4\}$$
$$ex_2^{\mathsf{A}} = (f_0^{\mathsf{A}}\ 1) + 2$$

If we assign ex_2^{A} a postcondition $\lambda\nu.\,\nu < 6$, then pre computes the weakest precondition to be $1 < 2 \wedge \delta(f_0^{\mathsf{A}}, 1) + 2 < 6$. It checks the argument 1 against f_0^{A}'s input type, and the whole expression against the given postcondition. The vc function validates that f_0^{A} correctly implements its specification as the type signature sets out.

6.7.4 Meta-properties of pre and vc

We first state monotonicity lemmas of pre and vc.

```
vc  : ∀{Γ}{T} → (⟦ Γ ⟧C → Set) → (⟦ Γ ▸ T ⟧C → Set) → Γ ⊢ᴬ T → Set
vc⃗ : ∀{Γ}{S T}{ξ}{ψ} → (⟦ Γ ⟧C → Set) → Γ ⊢ᴬ S { ξ }⟶ T { ψ } → Set

vc ϕ ψ (SUᴬ e) = vc ϕ (ᵏ ⊤) e
vc ϕ ψ (IFᴬ c e₁ e₂) = vc ϕ (ᵏ ⊤) c
                      × vc (λ γ → ϕ γ × ⟦ c ⟧⊢ᴬ γ ≡ true) ψ e₁
                      × vc (λ γ → ϕ γ × ⟦ c ⟧⊢ᴬ γ ≡ false) ψ e₂
vc ϕ ψ (LETᴬ e₁ e₂) = vc ϕ (ᵏ ⊤) e₁
                     × vc (λ (γ , s) → ϕ γ × s ≡ ⟦ e₁ ⟧⊢ᴬ γ)
                          (λ ((γ , _) , t) → ψ (γ , t)) e₂
vc ϕ ψ (PRDᴬ e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂
vc ϕ ψ (FSTᴬ e) = vc ϕ (ᵏ ⊤) e
vc ϕ ψ (SNDᴬ e) = vc ϕ (ᵏ ⊤) e
vc {Γ} ϕ ψ′ (APPᴬ {S = S}{T = T}{ξ = ξ}{ψ = ψ} f e)
  = vc⃗ ϕ f
  × vc ϕ ξ e
  × (∀(γ : ⟦ Γ ⟧C)(s : ⟦ S ⟧τ)(t : ⟦ T ⟧τ)
    → ϕ γ → ξ (γ , s) → ψ ((γ , s) , t) → ψ′ (γ , t))
vc ϕ ψ (BOPᴬ o e₁ e₂) = vc ϕ (ᵏ ⊤) e₁ × vc ϕ (ᵏ ⊤) e₂
vc _ _ _ = ⊤

vc⃗ {Γ = Γ}{S = S}{T = T} ϕ (FUNᴬ {ξ = ξ}{ψ = ψ} e)
  = (∀(γ : ⟦ Γ ⟧C)(s : ⟦ S ⟧τ) → ϕ γ → ξ (γ , s) → pre ψ e (γ , s))
  × vc (λ (γ , s) → ϕ γ × ξ (γ , s)) ψ e
```

Figure 6.11: The Agda definition of vc

Lemma 6.10 (pre is monotone). *For an annotated expression $\Gamma \vdash_A e : T$ in λ^A, if a predicate ψ_1 implies ψ_2, then pre ψ_1 e implies pre ψ_2 e.*

Proof. By induction on the structure of $\Gamma \vdash_A e : T$. □

Lemma 6.11 (vc is monotone). *For an annotated expression $\Gamma \vdash_A e : T$ in λ^A, if a predicate ϕ_2 implies ϕ_1, and under the stronger precondition ϕ_2, a postcondition ψ_1 implies ψ_2, then vc ϕ_1 ψ_1 e implies vc ϕ_2 ψ_2 e.*

Proof. By induction on the structure of $\Gamma \vdash_A e : T$. □

With the monotonicity lemmas, we can finally prove the soundness and completeness of pre and vc with respect to the typing rules of λ^C.

Theorem 6.12 (Completeness of pre and vc w.r.t. λ^C typing rules)**.** *If* $\Gamma; \phi \vdash_C \hat{e} : \{\nu : T \mid \psi\}$*, then* $vc\ \phi\ \psi\ \ulcorner\hat{e}\urcorner^C$ *and* $\phi\ \gamma \Rightarrow pre\ \psi\ \ulcorner\hat{e}\urcorner^C\ \gamma$ *for any semantic environment* γ *that respects* Γ*.*

Proof. We separately proof the completeness of the preand vcfunctions. Both of them can be proved by induction on the structure of $\Gamma; \phi \vdash_C \hat{e} : \{\nu : T \mid \psi\}$, with the help of Theorem 6.10 and Theorem 6.11 respectively. □

Corollary 6.13. *For an expression* $\Gamma \vdash_A e : T$ *in* λ^A*, if* $vc\ \phi\ \psi\ e$ *and* $\phi\ \gamma \Rightarrow pre\ \psi\ e\ \gamma$ *for any semantic environment* γ *that respects* Γ*, then there is a type derivation* $\Gamma; \phi \vdash_C \hat{e} : \{\nu : T \mid \psi\}$ *such that* $\ulcorner\hat{e}\urcorner^C = e$*.*

Proof. By induction on the structure of $\Gamma \vdash_A e : T$. □

Theorem 6.14 (Soundness of pre and vc w.r.t. λ^C typing rules)**.** *For an expression* $\Gamma \vdash_A e : T$ *in* λ^A*, if* $vc\ (pre\ \psi\ e)\ \psi\ e$*, then there is a type derivation* $\Gamma; pre\ \psi\ e \vdash_C \hat{e} : \{\nu : T \mid \psi\}$ *such that* $\ulcorner\hat{e}\urcorner^C = e$*.*

Proof. A direct consequence of Theorem 6.13. □

6.8 Related Work, Future Work and Conclusion

There is a very long line of prior work on refinement types, e.g. [Knowles and Flanagan 2009; Lehmann and Tanter 2017; Pavlinovic et al. 2021; Rondon et al. 2008; Vazou 2016], just to name a few. We find the work by Lehmann and Tanter [2016] most comparable. They define the language and the logical formulae fully deeply in Coq, and assumes an oracle that can answer the questions about logical entailment. In our formalisation, we interpret the language as shallow Agda terms, and the underlying logic is Agda's type system. Programmers serve as an oracle to construct proof terms. Knowles and Flanagan [2007]'s work is also closely related. It develops a decidable type reconstruction algorithm which preserves the typeability of a program. Their type reconstruction is highly influenced by the strongest postcondition predicate transformation. Prior work on formalising Hoare logic or separation logic is also very rich [Charguéraud 2020; Michael J. C. Gordon 1989; Jung, Krebbers, et al. 2018; Kleymann 1998; Nipkow and Klein 2014; Schirmer 2005; Shankar 2018], but they typically lack a formal connection with refinement types.

F* [Swamy et al. 2016] is a verification-oriented programming language, and it features a refinement type system. F* users can mix interactive theorem proving and SMT-based automatic verification methods to prove properties about their programs. One of the main research contributions of the F* work is its support for an extensible lattice of monadic effects. In that regard, F* is more expressive than Cogent, especially in its capability of handling effectful, exceptional and divergent programs; Cogent is a pure and total functional language and focuses on uniqueness types. The refinement types in both languages are formalised differently. In F*, expressions are given monadic types that are indexed by predicate transformers. Our work explores the connection between Hoare logic and refinement typing rules, especially how to encode the preconditions in the typing contexts. We also use our Agda formalisation to study different designs of refinement typing rules. It is yet to be seen whether F*'s type system subsumes the core calculus we formalised, if we restrict F*'s monadic effects to PURE—the monad for pure, recursive functions. The total correctness meta-theorem of F* is very similar to our refinement soundness results.

* * *

Admittedly, our attempt in formalising refinement type systems is still in its infancy. We list a few directions for future exploration.

Language features The languages that we presented in this paper are very preliminary. They do not yet support higher-order functions. Variants of Hoare style logics that deal with higher-order language features [Charlton 2011; Jung, Krebbers, et al. 2018; Régis-Gianas and Pottier 2008; Schwinghammer et al. 2009; Yoshida et al. 2007] will shed light on the extension to higher-order functions. It remains to be seen which techniques are compatible with the way in which the language is embedded and interpreted in Agda. General recursion is also missing from our formalisation. We surmise that recursion can be handled analogously to how a WHILE-loop is dealt with in Hoare logic. Hoare logic style reasoning turns out to be instrumental in languages with side-effects or concurrency. How to extend the unifying paradigm to languages with such features is also an open question. An equivalent question is how to formulate proof systems that support these features in terms of refinement type systems.

Delaying proof obligations As we have seen in the examples, constructing a typing tree for a program requires the developer to fill the holes with proof terms. The typechecking algorithm with pre and vc collects the proof obligations along the typing tree. This is effectively deferring the proofs to a later stage. It shares the same spirit as the Delay applicative functor by O'Connor [2019a]. It is yet to be seen how it can be applied in the construction of the typing trees in our formalisation.

Compositionality We stated in Section 6.7 that the λ^C language is not yet fully compositional in the sense of [Knowles and Flanagan 2009]. The interpretation function $\mathscr{E}[\![\cdot]\!]^{A}_{\mathsf{Tm}}$ is used in the definition of pre, and that effectively leaks the behaviour of the program to the reasoning thereof, penetrating the layer of abstraction provided by types. We dealt with it for functions: the implementation details of the function body and those of the argument are hidden from the reasoning process. We would like to further extend the compositional reasoning to other language constructs (via user type annotations) in future work.

Other program logics Lastly, in our formalisation, we use Hoare logic as the foundation for the typing rules. There are other flavours of program logics, most notably the dual of Hoare logic—Reverse Hoare Logic [de Vries and Koutavas 2011] and Incorrectness Logic [O'Hearn 2019]. We are intrigued to see if we can mount these logics onto our system, and how it interacts with a functional language that is, say, impure or concurrent.

* * *

In this chapter, we presented a simple yet novel Agda formalisation of refinement types on a small first-order functional language in the style of Hoare logic. It provides a testbed for studying the formal connections between refinement types and Hoare logic. We believe that our work is a valuable addition to the formal investigation into refinement types, and we hope that this work will foster more research into machine-checked formalisations of refinement type systems, and the connection with other logical systems such as Hoare style logics.

This work is a first step towards integrating refinement types into the Cogent framework. The Cogent typechecker can generate verification conditions for refinement typing as Isabelle/HOL theorems, and they can be manually proved by verification engineers with the help of SMT-solvers. It does not necessarily increase the amount of verification work, as the extra theorems originated from the refinement type system replace those previously in the manual functional correctness proof. Prior research has provided answers to the question of how to couple dependent types with linear types [Atkey 2018; McBride 2016]. We expect the interaction between refinement types and uniqueness types to resemble that. The extra expressive power from refinement types can be beneficial to Cogent users. They will allow the programmers to specify and document their designs more easily and precisely, as well as to reduce the amount of dead code in the programs. They also build an extra channel for programmers and verification engineers to communicate, as refinement types in programs directly become proof obligations. We posit that this addition is a perfect fit for Cogent.

Chapter 7

Conclusion

Developing low-level systems code and formally proving interesting properties about programs are individually challenging tasks. Combining the two—verifying low-level systems code—is even harder, let alone doing it at scale. Each of systems programming and formal verification has its own curriculum, with unique ideology and methodology. The work done by Amani [2016], O'Connor [2019b] and colleagues yielded the COGENT language and its verification framework, which was a major theoretical breakthrough in the search for a solution to this problem. While practising the COGENT methodology in real-world development, we identified several shortcomings of the COGENT framework:

- the compiled algebraic data types did not meet performance and conformance expectations;
- the language interface between COGENT and C was inadequate for real-world applications;
- COGENT gave developers little guidance or early feedback on the design and verifiability of their systems;
- a communication gap between systems programmers and verification engineers could be seen.

These shortcomings range from theoretical limitations to highly pragmatic matters. This thesis attempts to address these most pressing issues of COGENT that could be observed when it was deployed in a more realistic context. Also, as we have pointed out throughout the thesis, many of these solutions (and the techniques used in them) that we have presented can be applied more widely to solve similar problems, out of the context of COGENT. We showcased their construction and application, using COGENT as a vehicle for demonstration.

In this thesis, we presented COGENT, a high-level functional language for low-level systems programming and a code/proof co-generation framework for formally verifying systems programs. We extended the core COGENT language and enriched its ecosystem to meet the needs of systems programmers in real-world development. These additions are essential to make COGENT a practical programming language for developing high-assurance systems software.

In Chapter 2, we introduced the COGENT language, its uniqueness type system, the compilation pipeline and its verification framework. COGENT is the research platform on which we experimented with extensions and explored new ideas to answer the research questions we asked in Chapter 1.

One major theoretical contribution of prior work on COGENT is that it elevated the abstraction level on which programmers write systems code to a purely functional semantics. The purely functional abstraction, however, imposed serious limitations on output programs, as they typically did not respect the memory layout used by existing hardware and protocols, which could impair the design, performance, compatibility and usefulness of the resulting systems. In Chapter 3, we presented a data layout description language DARGENT and its data layout refinement framework to address this problem. DARGENT provides users with the much-needed features to fine-tune the memory representation of algebraic data types. We demonstrated, with real-world examples, that DARGENT is indispensable for writing high-performance, compliant low-level programs.

Transitioning from a purely research platform to a real-world application revealed limitations in COGENT's simplistic design. The structural type system used by COGENT became a bottleneck for its deployment at scale, as it hindered the integration of COGENT and C programs, which is essential for any non-trivial COGENT programs. In Chapter 4, we represented antiquoted-C—our solution to the design and implementation of COGENT's foreign function interface. Antiquoted-C is an intuitive and simple language interface, allowing COGENT and C to interoperate smoothly. We explored a different point in the design space that does not rely on name-mangling. Antiquoted-C composes existing tools in a novel way and thus requires minimal engineering effort to build.

Another challenge that arose when using COGENT to verify large-scale programs was due to the binary nature of formally verified software: fully verified or not. This is not only a huge gap in the trustworthiness of the software, but also an appreciable mental gap for the developers. More interim milestones for developers would help them pursue fully verified software in a more progressive and tractable fashion. To this end, in Chapter 5, we added a property-based testing framework to the COGENT ecosystem, which nicely complements COGENT's verification framework. Instead of testing logical properties directly, we used property-based testing in a novel way to test refinement relations. We argued that doing so helps the developers quickly reach the design that is amenable to formal verification. It also enables an incremental approach to full verification. With case studies, we showed the techniques we used for specifying the tests in various scenarios.

To open up avenues for more automation opportunities during formal verification of systems programs, which would further strengthen COGENT as a verification tool in real-world deployment, we experimented with integrating refinement types to COGENT. In Chapter 6, we explored refinement types from the theoretical end, by formalising a refinement type system in Agda. The encoding we used is unconventional: we represented the refinement types in terms of Hoare logic. It results in an encoding of the object language that is non-dependent and can be easily typechecked with backward reasoning. This exploration is a first step towards integrating refinement types into Cogent's type system, which we leave for future work.

The research on verifying low-level systems code has not finished, and there are more technologies that we want to experiment with using Cogent as a platform. In Chapter 3, we have seen that in several cases, the expressiveness of the Cogent language becomes the bottleneck. A richer type system that supports dependent types, finer-grained capabilities and first-class pointers is a big step forward. Concurrency has always been a question, a big one indeed, that can be asked about nearly any new research results. Cogent is no exception. Cogent itself relies on a single-threaded C semantics and does not support concurrency in its current form. A concurrent variant of the C semantics is formulated in the work of Amani, Andronick, et al. [2017]. A memory model for Cogent programs is an interesting topic for future research. Also relevant is the semantics of the Dargent extension in a multi-threaded setting, and the memory access patterns on modern architectures. All of these are exciting and useful research questions that are worth pursuing further. Adding refinement types is another lines of future work. As we have mentioned throughout the thesis, refinement types can have substantial impact on many other features of Cogent. Its interaction with Dargent layout, test data generation and the overall verification workflow have not been adequately studied in this thesis and they are all interesting territories in the design space. We plan to tackle these research challenges in the future.

Bibliography

Michael D. Adams and Thomas M. DuBuisson. 2012. “Template Your Boilerplate: Using Template Haskell for Efficient Generic Programming.” In: *Proceedings of the 2012 Haskell Symposium* (Haskell ’12). Association for Computing Machinery, Copenhagen, Denmark, 13–24. ISBN: 9781450315746. DOI: 10.1145/2364506.2364509.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, & Tools with Gradiance.* (2nd ed.). Addison-Wesley Publishing Company, USA. ISBN: 0321547985, 9780321547989.

Eyad Alkassar and Mark A. Hillebrand. Oct. 2008. “Formal Functional Verification of Device Drivers.” In: *Verified Software: Theories, Tools and Experiments* (Lecture Notes in Computer Science). Ed. by Natarajan Shankar and Jim Woodcock. Vol. 5295. Springer, Toronto, Canada, 225–239.

José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. 2011. “An Overview of Formal Methods Tools and Techniques.” In: *Rigorous Software Development: An Introduction to Program Verification.* Springer London, London, 15–44. ISBN: 978-0-85729-018-2. DOI: 10.1007/978-0-85729-018-2_2.

Sidney Amani. Aug. 2016. “A Methodology for Trustworthy File Systems.” PhD Thesis. CSE, UNSW, Sydney, Australia.

Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joey Tuong. Jan. 2017. “COMPLX: a verification framework for concurrent imperative programs.” In: *International Conference on Certified Programs and Proofs.* SIGPLAN Notices, Paris, France, 138–150. DOI: https://doi.org/10.1145/3018610.3018627.

Sidney Amani, Peter Chubb, Alastair Donaldson, Alexander Legg, Leonid Ryzhyk, and Yanjin Zhu. Nov. 2012. “Automatic Verification of Message-Based Device Drivers.” In: *Systems Software Verification.* Sydney, Australia, 1–14.

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. Apr. 2016. “Cogent: Verifying High-Assurance File System Implementations.” In: *International Conference on Architectural Support for Programming Languages and Operating Systems.* Atlanta, GA, USA, 175–188. DOI: 10.1145/2872362.2872404.

Sidney Amani and Toby Murray. Nov. 2015. “Specifying a Realistic File System.” In: *Workshop on Models for Formal Analysis of Real Systems.* Suva, Fiji, 1–9.

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. Jan. 2014. “A Verified Information-Flow Architecture.” In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* San Diego, CA, USA, 165–178.

June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He (Jason) Zhang, and Liming Zhu. June 2012. “Large-Scale Formal Verification in Practice: A Process Perspective.” In: *International Conference on Software Engineering*. ACM, Zurich, Switzerland, 1002–1011.

Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. Apr. 2015. “Testing AUTOSAR software with QuickCheck.” In: *International Conference on Software Testing, Verification and Validation (ICST) Workshops*. Graz, AT, 1–4. DOI: 10.1109/ICSTW.2015.7107466.

Robert Atkey. 2018. “Syntax and Semantics of Quantitative Type Theory.” In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (LICS ’18). ACM, Oxford, United Kingdom, 56–65. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209189.

Godmar Back. 2002. “DataScript – A Specification and Scripting Language for Binary Data.” In: *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering* (Lecture Notes in Computer Science). Ed. by Don Batory, Charles Consel, and Walid Taha. Vol. 2487. Springer, 66–77.

R. J. R. Back. Aug. 1988. “A calculus of refinements for program derivations.” *Acta Informatica*, 25, 6, 593–624. DOI: 10.1007/BF00291051.

Henry G. Baker. Aug. 1992. “Lively Linear Lisp: “Look Ma, No Garbage!”” *SIGPLAN Not.*, 27, 8, 89–98. DOI: 10.1145/142137.142162.

Ballerina FFI. Retrieved August 2022 from https://ballerina.io/learn/java-interoperability/ballerina-ffi/.

Julian Bangert and Nickolai Zeldovich. 2014. “Nail: A Practical Tool for Parsing and Generating Data Formats.” In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Broomfield, CO, 615–628. ISBN: 978-1-931971-16-4. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert.

Erik Barendsen and Sjaak Smetsers. 1993. “Conventional and Uniqueness Typing in Graph Rewrite Systems.” In: *Foundations of Software Technology and Theoretical Computer Science* (Lecture Notes in Computer Science). Vol. 761, 41–51.

base: Basic libraries. Retrieved August 2022 from https://hackage.haskell.org/package/base.

Alan Bawden. 1999. “Quasiquotation in Lisp.” In: *Partial Evaluation and Semantic-Based Program Manipulation*, 4–12.

Brian Behlendorf. 2011. *POSIX Filesystem Test Suite*. Retrieved August 2022 from https://github.com/zfsonlinux/fstest.

Stefan Berghofer and Tobias Nipkow. 2004. “Random Testing in Isabelle/HOL.” In: *Proceedings of the Software Engineering and Formal Methods, Second International Conference* (SEFM ’04). IEEE Computer Society, Washington, DC, USA, 230–239. ISBN: 0-7695-2222-X. DOI: 10.1109/SEFM.2004.36.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Dec. 2017. “Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language.” *Proc. ACM Program. Lang.*, 2, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Article 5, 5:1–5:29. DOI: 10.1145/3158093.

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. 2010. “Testing Polymorphic Properties.” In: *Programming Languages and Systems*. Ed. by Andrew D. Gordon. Springer Berlin Heidelberg, Berlin, Heidelberg, 125–144. ISBN: 978-3-642-11957-6.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer. ISBN: 3-540-20854-2. DOI: 10.1007/978-3-662-07964-5.

Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. 2010. “Semantic Subtyping with an SMT Solver.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (ICFP ’10). Association for Computing Machinery, Baltimore, Maryland, USA, 105–116. ISBN: 9781605587943. DOI: 10.1145/1863543.1863560.

Simon Biggs, Damon Lee, and Gernot Heiser. Aug. 2018. “The Jury Is In: Monolithic OS Design Is Flawed.” In: *Asia-Pacific Workshop on Systems (APSys)*. ACM SIGOPS, Korea, 7 pages. DOI: https://doi.org/10.1145/3265723.3265733.

Dines Bjørner and Klaus Havelund. 2014. “40 Years of Formal Methods - Some Obstacles and Some Possibilities?” In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings* (Lecture Notes in Computer Science). Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Springer, 42–61. DOI: 10.1007/978-3-319-06410-9_4.

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. “A Formally Verified Compiler for Lustre.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2017). Association for Computing Machinery, Barcelona, Spain, 586–601. ISBN: 9781450349888. DOI: 10.1145/3062341.3062358.

Jonathan Bowen and Victoria Stavridou. July 1993. “Safety-critical systems, formal methods and standards.” English. *Software Engineering Journal*, 8, 189–209(20), 4. https://digital-library.theiet.org/content/journals/10.1049/sej.1993.0025.

Edwin C. Brady. 2011. “IDRIS — Systems Programming Meets Full Dependent Types.” In: *Proceedings of the 2011 ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*. ACM, Austin, Texas, USA, 43–54. ISBN: 978-1-4503-0487-0. DOI: 10.1145/1929529.1929536.

Edwin C. Brady. Sept. 2013. “Idris, a general-purpose dependently typed programming language: Design and implementation.” *Journal of Functional Programming*, 23, 552–593, 05. DOI: 10.1017/S095679681300018X.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. “Ready, set, verify! applying hs-to-coq to real-world Haskell code (experience report).” *PACMPL*, 2, ICFP, 89:1–89:16.

Neil C.C. Brown and Adam T. Sampson. 2009. “Alloy: Fast Generic Transformations for Haskell.” In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Haskell ’09). Association for Computing Machinery, Edinburgh, Scotland, 105–116. ISBN: 9781605585086. DOI: 10.1145/1596638.1596652.

Lukas Bulwahn. 2012. “The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing Under One Roof.” In: *International Conference on Certified Programs and Proofs*. Springer-Verlag, Kyoto, Japan, 92–108. ISBN: 978-3-642-35307-9. DOI: 10.1007/978-3-642-35308-6_10.

CakeML: A Verified Implementation of ML. Retrieved September 2022 from https://cakeml.org/.

Luca Cardelli and Peter Wegner. Dec. 1985. “On Understanding Types, Data Abstraction, and Polymorphism.” *ACM Comput. Surv.*, 17, 4, 471–523. DOI: 10.1145/6041.6042.

Giuseppe Castagna and Alain Frisch. 2005. “A Gentle Introduction to Semantic Subtyping.” In: *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP ’05). Association for Computing Machinery, Lisbon, Portugal, 198–199. ISBN: 1595930906. DOI: 10.1145/1069774.1069793.

cgo Documentation. Retrieved August 2022 from https://pkg.go.dev/cmd/cgo.

Manuel M. T. Chakravarty. 1999. “C → Haskell, or Yet Another Interfacing Tool.” In: *Implementation of Functional Languages, 11th International Workshop, IFL’99, Lochem, The Netherlands, September 7-10, 1999, Selected Papers*, 131–148. DOI: 10.1007/10722298_8.

Manuel M. T. Chakravarty. 2014. “Foreign inline code: systems demonstration.” In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014.* Ed. by Wouter Swierstra. ACM, 119–120. DOI: 10.1145/2633357.2633372.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. “Associated types with class.” In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL ’05). ACM, Long Beach, California, USA, 1–13. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040306.

James Chapman. Jan. 2009. “Type Theory Should Eat Itself.” *Electron. Notes Theor. Comput. Sci.*, 228, 21–36. DOI: 10.1016/j.entcs.2008.12.114.

Arthur Charguéraud. Aug. 2020. “Separation Logic for Sequential Programs (Functional Pearl).” *Proc. ACM Program. Lang.*, 4, International Conference on Functional Programming, Article 116, 34 pages. DOI: 10.1145/3408998.

Nathaniel Charlton. 2011. “Hoare Logic for Higher Order Store Using Simple Semantics.” In: *Logic, Language, Information and Computation.* Ed. by Lev D. Beklemishev and Ruy de Queiroz. Springer Berlin Heidelberg, Berlin, Heidelberg, 52–66. ISBN: 978-3-642-20920-8.

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. “Toward compositional verification of interruptible OS kernels and device drivers.” In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 431–447. DOI: 10.1145/2908080.2908101.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Oct. 2015. “Using Crash Hoare Logic for Certifying the FSCQ File System.” In: *ACM Symposium on Operating Systems Principles.* Monterey, CA, 18–37.

Zilin Chen. Sept. 2022. “A Hoare Logic Style Refinement Types Formalisation.” In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (TyDe ’22). ACM, Ljubljana, Slovenia, 14 pages. DOI: 10.1145/3546196.3550162.

Zilin Chen. 2017. “COGENT$^{\Uparrow}$: Giving Systems Engineers A Stepping Stone (Extended abstract).” In: *The workshop on Type-Driven Development* (TyDe’17). Oxford, UK. Retrieved April 2019 from https://www.cse.unsw.edu.au/~zilinc/tyde17.pdf.

Zilin Chen, Matt Di Meglio, Liam O’Connor, Partha Susarla Ajay, Christine Rizkallah, and Gabriele Keller. Jan. 2019. *A Data Layout Description Language for Cogent.* at PriSC. Lisbon, Portugal.

Zilin Chen, Ambroise Lafont, Liam O’Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Oct. 2022. *Dargent: A Silver Bullet for Verified Data Layout Refinement (Artefact).* POPL23 artefact. DOI: 10.5281/zenodo.7220452.

Zilin Chen, Ambroise Lafont, Liam O’Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Jan. 2023. “DARGENT: A Silver Bullet for Verified Data Layout Refinement.” *Proc. ACM Program. Lang.*, 7, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Article 47, 27 pages. DOI: 10.1145/3571240.

Zilin Chen, Liam O’Connor, Gabriele Keller, Gerwin Klein, and Gernot Heiser. Oct. 28, 2017. “The Cogent Case for Property-Based Testing.” In: *Workshop on Programming Languages and Operating Systems (PLOS).* ACM, Shanghai, China, 1–7. ISBN: 9781450351539. DOI: https://doi.org/10.1145/3144555.3144556.

Zilin Chen and Christine Rizkallah. 2022. *Why It’s Nice to be Quoted: A COGENT FFI.* Unpublished.

Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. Dec. 2022a. "Property-Based Testing: Climbing the Stairway to Verification." In: *ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2022). ACM, Auckland, New Zealand, 14 pages. DOI: 10.1145/3567512.3567520.

Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. Oct. 2022b. *Property-Based Testing: Climbing the Stairway to Verification (Artefact).* SLE2022 Artefact. DOI: 10.5281/zenodo.7248640.

Louis Cheung, Liam O'Connor, and Christine Rizkallah. 2022. "Overcoming Restraint: Composing Verification of Foreign Functions with Cogent." In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2022). ACM, Philadelphia, PA, USA, 13–26. ISBN: 9781450391825. DOI: 10.1145/3497775.3503686.

Adam Chlipala. 2015. "From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification." In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '15). Association for Computing Machinery, Mumbai, India, 609–622. ISBN: 9781450333009. DOI: 10.1145/2676726.2677003.

Koen Claessen and John Hughes. 2000. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs." In: *Proceedings of the 5th International Conference on Functional Programming*, 268–279.

Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. "Finding Race Conditions in Erlang with QuickCheck and PULSE." In: *International Conference on Functional Programming*. ACM, Edinburgh, Scotland, 149–160. ISBN: 978-1-60558-332-7. DOI: 10.1145/1596550.1596574.

Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. 2021. "Layered Formal Verification of a TCP Stack." In: *2021 IEEE Secure Development Conference (SecDev)*, 86–93. DOI: 10.1109/SecDev51306.2021.00028.

David Cock, Gerwin Klein, and Thomas Sewell. Aug. 2008. "Secure Microkernels, State Monads and Scalable Refinement." In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, Sofiène Tahar. Springer, Montreal, Canada, 167–182. DOI: 10.1007/978-3-540-71067-7_16.

Code quotations. Retrieved February 2022 from https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations.

Karl Cronburg and Samuel Z. Guyer. 2019. "Floorplan: Spatial Layout in Memory Management Systems." In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (GPCE 2019). Association for Computing Machinery, Athens, Greece, 81–93. ISBN: 9781450369800. DOI: 10.1145/3357765.3359519.

Nils Anders Danielsson. 2006. "A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family." In: *Proceedings of the 2006 International Conference on Types for Proofs and Programs* (TYPES'06). Springer-Verlag, Nottingham, UK, 93–109. ISBN: 3540744630.

Matthew Danish and Hongwei Xi. Apr. 2014. "Using Lightweight Theorem Proving in an Asynchronous Systems Context." In: *Proceedings of the 6th NASA Formal Methods Symposium* (Lecture Notes in Computer Science). Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 8430. Springer, Houston, TX, USA, 158–172. DOI: 10.1007/978-3-319-06200-6_12.

Edsko de Vries and Vasileios Koutavas. 2011. "Reverse Hoare Logic." In: *Software Engineering and Formal Methods*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171. ISBN: 978-3-642-24690-6.

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2007. "Uniqueness Typing Redefined." In: *Implementation and Application of Functional Languages*. Ed. by Zoltán Horváth, Viktória Zsók, and Andrew Butterfield. Springer Berlin Heidelberg, Berlin, Heidelberg, 181–198. ISBN: 978-3-540-74130-5.

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2008. "Uniqueness Typing Simplified." In: *Implementation and Application of Functional Languages*. Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Springer Berlin Heidelberg, Berlin, Heidelberg, 201–218. ISBN: 978-3-540-85373-2.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Jan. 2015. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant." In: *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India.

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. July 2019. "Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats." *Proc. ACM Program. Lang.*, 3, International Conference on Functional Programming, Article 82, 29 pages. DOI: 10.1145/3341686.

Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Sept. 2006. "Running the Manual: An Approach to High-Assurance Microkernel Development." In: *Proceedings of the ACM SIGPLAN Haskell Workshop*. Portland, OR, USA.

Iavor S. Diatchki and Mark P. Jones. 2006. "Strongly Typed Memory Areas Programming Systems-Level Data Structures in a Functional Language." In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell* (Haskell '06). Association for Computing Machinery, Portland, Oregon, USA, 72–83. ISBN: 1595934898. DOI: 10.1145/1159842.1159851.

Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. "High-Level Views on Low-Level Representations." In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (ICFP '05). Association for Computing Machinery, Tallinn, Estonia, 168–179. ISBN: 1595930647. DOI: 10.1145/1086365.1086387.

Edsger W. Dijkstra. Aug. 1975. "Guarded commands, nondeterminacy and formal derivation of programs." *Communications of the ACM*, 18, 8, 453–457. DOI: 10.1145/360933.360975.

Oscar Downing. 2021. "Enhancements to the COGENT Property-Based Testing Framework." Undergraduate Thesis. CSE, UNSW, Sydney, Australia. https://people.eng.unimelb.edu.au/rizkallahc/theses/oscar-downing-honours-thesis.pdf.

Jianjun Duan and John Regehr. Oct. 2010. "Correctness Proofs for Device Drivers in Embedded Systems." In: *Systems Software Verification*. USENIX Association, Vancouver, BC, CA.

Jana Dunfield and Neel Krishnaswami. May 2021. "Bidirectional Typing." *ACM Comput. Surv.*, 54, 5, Article 98, 38 pages. DOI: 10.1145/3450952.

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. "Feat: Functional Enumeration of Algebraic Types." In: *Proceedings of the 2012 Haskell Symposium* (Haskell '12). ACM, Copenhagen, Denmark, 61–72. ISBN: 978-1-4503-1574-6. DOI: 10.1145/2364506.2364515.

Peter Dybjer. 2000. "A general formulation of simultaneous inductive-recursive definitions in type theory." *Journal of Symbolic Logic*, 65, 2, 525–549. DOI: 10.2307/2586554.

Peter Dybjer, Haiyan Qiao, and Makoto Takeyama. 2003. "Combining Testing and Proving in Dependent Type Theory." In: *Theorem Proving in Higher Order Logics*. Ed. by David Basin and Burkhart Wolff. Springer Berlin Heidelberg, Berlin, Heidelberg, 188–203. ISBN: 978-3-540-45130-3.

Carl Eastlund. 2009. “DoubleCheck Your Theorems.” In: *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and Its Applications* (ACL2 ’09). ACM, Boston, Massachusetts, USA, 42–46. ISBN: 9781-60558-742-4. DOI: 10.1145/1637837.1637844.

Rob Ennals, Richard Sharp, and Alan Mycroft. 2004. “Linear Types for Packet Processing.” In: *Proceedings of the 13th European Symposium on Programming* (Lecture Notes in Computer Science). Ed. by David Schmidt. Vol. 2986. Springer, 204–218.

F^*. Retrieved September 2022 from https://www.fstar-lang.org/.

Federal Aviation Administration. 2018. “Airworthiness Directives; The Boeing Company Airplanes.” In: Federal Register Volume 83, Number 200. FR Doc No: 2018-22152. https://drs.faa.gov/browse/excelExternalWindow/7332A6AC61F5CB30862583280049940E.0001.

Kathleen Fisher and Robert Gruber. 2005. “PADS: a domain-specific language for processing ad hoc data.” In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI ’05). ACM, Chicago, IL, USA, 295–304. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065046.

Kathleen Fisher and David Walker. 2011. “The PADS Project: An Overview.” In: *International Conference on Database Theory*. ACM, Uppsala, Sweden, 11–17.

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. “Linear Regions Are All You Need.” In: *Programming Languages and Systems*. Ed. by Peter Sestoft. Springer Berlin Heidelberg, Berlin, Heidelberg, 7–21. ISBN: 978-3-540-33096-7.

Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. “You Can Have It All: Abstraction and Good Cache Performance.” In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward! 2017). Association for Computing Machinery, Vancouver, BC, Canada, 148–167. ISBN: 9781450355308. DOI: 10.1145/3133850.3133861.

Juliana Franco, Alexandros Tasos, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. 2019. *Safely Abstracting Memory Layouts*. DOI: 10.48550/ARXIV.1901.08006.

David Gay and Alex Aiken. 2001. “Language Support for Regions.” In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (PLDI ’01). Association for Computing Machinery, Snowbird, Utah, USA, 70–80. ISBN: 1581134142. DOI: 10.1145/378795.378815.

Marcell van Geest and Wouter Swierstra. 2017. “Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data.” In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development* (TyDe 2017). Association for Computing Machinery, Oxford, UK, 30–40. ISBN: 9781450351836. DOI: 10.1145/3122975.3122979.

GHC User’s Guide Documentation, Version 8.6.4. Retrieved April 2019 from https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf.

Jean-Yves Girard. 1987. “Linear Logic.” *Theoretical Computer Science*, 50, 1–102.

M. J. C. Gordon and T. F. Melham. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, USA. ISBN: 0521441897.

Michael J. C. Gordon. 1989. “Mechanizing Programming Logics in Higher Order Logic.” In: *Current Trends in Hardware Verification and Automated Theorem Proving*. Ed. by Graham Birtwistle and P. A. Subrahmanyam. Springer New York, New York, NY, 387–439. ISBN: 978-1-4612-3658-0. DOI: 10.1007/978-1-4612-3658-0_10.

David Greenaway. Mar. 2015. “Automated proof-producing abstraction of C code.” PhD Thesis. CSE, UNSW, Sydney, Australia.

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. June 2014. “Don’t Sweat the Small Stuff: Formal Verification of C Code Without the Pain.” In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh, UK, 429–439. DOI: 10.1145/2594291.2594296.

Michael Greenberg. 2015. *A refinement type by any other name*. http://www.weaselhat.com/2015/03/16/a-refinement-type-by-any-other-name/.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Nov. 2016. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.” In: *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Savannah, GA, US, 653–669.

Havva Gulay Gurbuz and Bedir Tekinerdogan. Dec. 2018. “Model-based testing for software safety: a systematic mapping study.” *Software Quality Journal*, 26, 4, 1327–1372. DOI: 10.1007/s11219-017-9386-2.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Mar. 1996. “Type classes in Haskell.” *ACM Transactions on Programming Languages and Systems*, 18, 2, 109–138.

Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. Sept. 2005. “A principled approach to operating system construction in Haskell.” In: *Proceedings of the 10th International Conference on Functional Programming*. Tallinn, Estonia, 116–128.

Robert Harper. 2016. *Practical Foundations for Programming Languages*. (2nd ed.). Cambridge University Press, USA. ISBN: 1107150302.

Dana Harrington. 2006. “Uniqueness logic.” *Theoretical Computer Science*, 354, 1, 24–41. Algebraic Methods in Language Processing. DOI: https://doi.org/10.1016/j.tcs.2005.11.006.

Gernot Heiser, June Andronick, Kevin Elphinstone, Gerwin Klein, Ihor Kuz, and Leonid Ryzhyk. Oct. 2010. “The Road to Trustworthy Systems.” In: *ACM Workshop on Scalable Trusted Computing (ACMSTC)*. ACM, Chicago, IL, USA, 3–10.

Gernot Heiser and Kevin Elphinstone. Apr. 2016. “L4 Microkernels: The Lessons from 20 Years of Research and Deployment.” *ACM Transactions on Computer Systems*, 34, 1, 1:1–1:29. DOI: 10.1145/2893177.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2007. “Comparing Approaches to Generic Programming in Haskell.” In: *Datatype-Generic Programming*. Ed. by Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring. Springer Berlin Heidelberg, Berlin, Heidelberg, 72–149. ISBN: 978-3-540-76786-2.

Martin Hirzel and Robert Grimm. 2007. “Jeannie: Granting Java Native Interface Developers Their Wishes.” In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’07). Association for Computing Machinery, Montreal, Quebec, Canada, 19–38. ISBN: 9781595937865. DOI: 10.1145/1297027.1297030.

Martin Hofmann. 2000. “A Type System for Bounded Space and Functional In-Place Update–Extended Abstract.” In: (Lecture Notes in Computer Science). Ed. by Gert Smolka. Vol. 1782. Springer, 165–179.

Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. “Testing Noninterference, Quickly.” In: *International Conference on Functional Programming*. Boston, Massachusetts, USA, 455–468. DOI: 10.1145/2500365.2500574.

John Hughes. 2016. "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane." In: *A List of Successes That Can Change the World*. Lecture Notes in Computer Science. Vol. 9600. Ed. by S. Lindley et al. Springer, 169–186. DOI: 10.1007/978-3-319-30936-1_9.

Adrian Hunter. 2008. *A Brief Introduction to the Design of UBIFS*. Retrieved November 2018 from http://linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

INRIA. 2017. *Camlp5 Documentation*. Retrieved February 2023 from https://camlp5.readthedocs.io/en/latest/index.html.

ISO. Feb. 2012. *ISO/IEC 23271:2012: Information technology — Common Language Infrastructure (CLI)*. International Organization for Standardization, Geneva, Switzerland, 546.

ISO. Dec. 1999. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, 538.

Java Native Interface Specification. Retrieved May 2022 from https://docs.oracle.com/en/java/javase/18/docs/specs/jni/index.html.

Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. 2008. "Libraries for Generic Programming in Haskell." In: *Proceedings of the 6th International Conference on Advanced Functional Programming* (AFP'08). Springer-Verlag, Heijen, The Netherlands, 165–229. ISBN: 3642046517.

Ranjit Jhala. 2019. *Liquid Types vs. Floyd-Hoare Logic*. Retrieved May 2022 from https://ucsd-progsys.github.io/liquidhaskell-blog/2019/10/20/why-types.lhs/.

Ranjit Jhala and Niki Vazou. 2021. "Refinement Types: A Tutorial." *Foundations and Trends in Programming Languages*, 6, 3–4, 159–317. DOI: 10.1561/2500000032.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Dec. 2017. "RustBelt: Securing the Foundations of the Rust Programming Language." *Proc. ACM Program. Lang.*, 2, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Article 66, 34 pages. DOI: 10.1145/3158154.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." *Journal of Functional Programming*, 28, e20. DOI: 10.1017/S0956796818000151.

Matt Kaufmann and J Strother Moore. 2018. *ACL2*. Retrieved April 2019 from http://www.cs.utexas.edu/users/moore/acl2/.

Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. Nov. 2013. "File Systems Deserve Verification Too!" In: *Workshop on Programming Languages and Operating Systems (PLOS)*. Farmington, Pennsylvania, USA, 1–7. DOI: 10.1145/2525528.2525530.

Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language*. Retrieved November 2022 from https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Feb. 2014. "Comprehensive Formal Verification of an OS Microkernel." *ACM Transactions on Computer Systems*, 32, 1, 2:1–2:70. DOI: 10.1145/2560537.

Gerwin Klein, June Andronick, Gabriele Keller, Daniel Matichuk, Toby Murray, and Liam O'Connor. Sept. 2017. "Provably Trustworthy Systems." *Philosophical Transactions of the Royal Society A*, 375, 1–23, 2104. DOI: https://doi.org/10.1098/rsta.2015.0404.

Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Aug. 2009. "Experience Report: seL4 — Formally Verifying a High-Performance Microkernel." In: *Proceedings of the 14th International Conference on Functional Programming*. ACM, Edinburgh, UK, 91–96.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Oct. 2009. "seL4: Formal Verification of an OS Kernel." In: *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, USA, 207–220.

Gerwin Klein, Thomas Sewell, and Simon Winwood. Mar. 2010. "Refinement in the formal verification of seL4." In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David Hardin. Springer, 323–339. ISBN: 978-1-4419-1538-2.

Thomas Kleymann. 1998. "Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs." PhD Thesis. School of Informatics, College of Science and Engineering, Edinburgh, UK.

Edward A. Kmett. 2022. *lens: Lenses, Folds and Traversals*. Retrieved August 2022 from `https://hackage.haskell.org/package/lens`.

Kenneth Knowles and Cormac Flanagan. 2009. "Compositional Reasoning and Decidable Checking for Dependent Contract Types." In: *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification* (PLPV '09). Association for Computing Machinery, Savannah, GA, USA, 27–38. ISBN: 9781605583303. DOI: `10.1145/1481848.1481853`.

Kenneth Knowles and Cormac Flanagan. 2007. "Type Reconstruction for General Refinement Types." In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, 505–519. DOI: `10.1007/978-3-540-71316-6_34`.

Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. 1999. "A Readable TCP in the Prolac Protocol Language." In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM '99). Association for Computing Machinery, Cambridge, Massachusetts, USA, 3–13. ISBN: 1581131356. DOI: `10.1145/316188.316200`.

Shinji Kono. 2022. *Constructing ZF Set Theory in Agda*. Retrieved May 2022 from `https://github.com/shinji-kono/zf-in-agda`.

Pieter Koopman, Peter Achten, and Rinus Plasmeijer. 2012. "Model Based Testing with Logical Properties versus State Machines." In: *Implementation and Application of Functional Languages*. Ed. by Andy Gill and Jurriaan Hage. Springer Berlin Heidelberg, Berlin, Heidelberg, 116–133. ISBN: 978-3-642-34407-7.

Pieter Koopman and Rinus Plasmeijer. 2006. "Automatic Testing of Higher Order Functions." In: *Programming Languages and Systems*. Ed. by Naoki Kobayashi. Springer Berlin Heidelberg, Berlin, Heidelberg, 148–164. ISBN: 978-3-540-48938-2.

Robbert Krebbers. 2015. "The C Standard formalised in Coq." Ph.D. Dissertation. Radboud University Nijmegen.

Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. Jan. 2014. "CakeML: A Verified Implementation of ML." In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Ed. by Peter Sewell. ACM Press, San Diego, 179–191. DOI: `10.1145/2535838.2535841`.

Ralf Lämmel and Simon Peyton Jones. 2005. "Scrap your boilerplate with class: extensible generic functions." In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (ICFP '05). ACM, Tallinn, Estonia, 204–215. ISBN: 1-59593-064-7. DOI: `10.1145/1086365.1086391`.

Ralf Lämmel and Simon Peyton Jones. 2003. “Scrap your boilerplate: a practical design pattern for generic programming.” In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (TLDI ’03). ACM, New Orleans, Louisiana, USA, 26–37. ISBN: 1-58113-649-8. DOI: 10.1145/604174.604179.

Peter Lammich. 2013. “Automatic Data Refinement.” In: *Proceedings of the 4th International Conference on Interactive Theorem Proving*. Lecture Notes in Computer Science. Vol. 7998. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Springer, 84–99. ISBN: 978-3-642-39633-5. DOI: 10.1007/978-3-642-39634-2_9.

Peter Lammich and Andreas Lochbihler. Mar. 2018. “Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches.” *Journal of Automated Reasoning*. DOI: 10.1007/s10817-018-9461-9.

Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. “Beginner’s Luck: A Language for Property-based Generators.” In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Paris, France, 114–129. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009868.

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*. Retrieved April 2019 from https://softwarefoundations.cis.upenn.edu/qc-current/index.html.

Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. July 2021. “STORM: Refinement Types for Secure Web Applications.” In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. ISBN: 978-1-939133-22-9. https://www.usenix.org/conference/osdi21/presentation/lehmann.

Nico Lehmann and Éric Tanter. 2016. “Formalizing Simple Refinement Types in Coq.” In: *The Second International Workshop on Coq for PL* (CoqPL’16). St. Petersburg, Florida, United States.

Nico Lehmann and Éric Tanter. 2017. “Gradual Refinement Types.” In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Paris, France, 775–788. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009856.

Daan Leijen and Erik Meijer. July 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/.

K. Rustan M. Leino. Oct. 2010. “Dafny: An Automatic Program Verifier for Functional Correctness.” In: *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (Lecture Notes in Computer Science). Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Springer, Yogyakarta, Indonesia, 348–370. ISBN: 978-3-642-17510-7. DOI: 10.1007/978-3-642-17511-4_20.

Xavier Leroy. 2009. “Formal verification of a realistic compiler.” *Communications of the ACM*, 52, 7, 107–115. DOI: 10.1145/1538788.1538814.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. “Implicit Parameters: Dynamic Scoping with Static Types.” In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Boston, MA, USA, 108–118. ISBN: 1-58113-125-9. DOI: 10.1145/325694.325708.

Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Apr. 2022. “Linear Types for Large-Scale Systems Verification.” *Proc. ACM Program. Lang.*, 6, OOPSLA1, Article 69, 28 pages. DOI: 10.1145/3527313.

LibOpenCM3. Retrieved August 2022 from http://libopencm3.org/.

David R. MacIver. 2016a. *Integrated vs Type-based Shrinking*. Article. Retrieved April 2019 from http://hypothesis.works/articles/integrated-shrinking.

David R. MacIver. 2016b. *QuickCheck in Every Language*. Retrieved April 2019 from https://hypothesis.works/articles/quickcheck-in-every-language.

David MacQueen. 1984. "Modules for Standard ML." In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (LFP '84). Association for Computing Machinery, Austin, Texas, USA, 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036.

Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. 2007. "Melange: Creating a "Functional" Internet." In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (EuroSys '07). Association for Computing Machinery, Lisbon, Portugal, 101–114. ISBN: 9781595936363. DOI: 10.1145/1272996.1273009.

Toshiyuki Maeda. 2015. *Kernel Mode Linux: Execute user processes in kernel mode*. Retrieved April 2019 from http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. "A Generic Deriving Mechanism for Haskell." In: *Proceedings of the Third ACM Haskell Symposium on Haskell* (Haskell '10). Association for Computing Machinery, Baltimore, Maryland, USA, 37–48. ISBN: 9781450302524. DOI: 10.1145/1863523.1863529.

Geoffrey Mainland. 2021. *language-c-quote: C/CUDA/OpenCL/Objective-C quasiquoting library*. Retrieved January 2022 from https://hackage.haskell.org/package/language-c-quote.

Geoffrey Mainland. 2007. "Why it's nice to be quoted: quasiquoting for Haskell." In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop* (Haskell '07). ACM, Freiburg, Germany, 73–82. ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291211.

Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. 2007. "PADS/ML: a functional data description language." In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '07). ACM, Nice, France, 77–83. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190231.

Simon Marlow. 2010a. *Happy: The Parser Generator for Haskell*. Retrieved February 2022 from https://www.haskell.org/happy/.

Simon Marlow. 2010b. *Haskell 2010 Language Report*. Language Report. Chap. 8.4.2. Retrieved April 2019 from https://www.haskell.org/onlinereport/haskell2010.

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. "Linearity and Uniqueness: An Entente Cordiale." In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Springer International Publishing, Cham, 346–375. ISBN: 978-3-030-99336-8.

Per Martin-Löf. 1972. *An Intuitionistic Theory of Types*.

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis.

Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. Feb. 2015. "Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification." In: *International Conference on Software Engineering*. Firenze, Italy, 11.

Conor McBride. Mar. 2016. "I got plenty o' nuttin'." In: *A List of Successes That Can Change the World*. Lecture Notes in Computer Science. Ed. by Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella. Springer, 207–233. ISBN: 9783319309354. DOI: 10.1007/978-3-319-30936-1_12.

Conor McBride. 2010. "Outrageous but Meaningful Coincidences: Dependent Type-Safe Syntax and Evaluation." In: *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*

(WGP '10). Association for Computing Machinery, Baltimore, Maryland, USA, 1–12. ISBN: 9781450302517. DOI: 10.1145/1863495.1863497.

Peter J. McCann and Satish Chandra. 2000. "PacketTypes: abstract specification of network protocol messages." In: *Proceedings of the ACM Conference on Communications*. ACM, Stockholm, Sweden, 321–333.

Robin Milner. 1978. "A Theory of Type Polymorphism in Programming." *J. Comput. Syst. Sci.*, 17, 3, 348–375.

John C. Mitchell and Gordon D. Plotkin. 1985. "Abstract Types Have Existential Types." In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '85). Association for Computing Machinery, New Orleans, Louisiana, USA, 37–51. ISBN: 0897911474. DOI: 10.1145/318593.318606.

Neil Mitchell and Colin Runciman. 2007. "Uniform Boilerplate and List Processing." In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Haskell '07). Association for Computing Machinery, Freiburg, Germany, 49–60. ISBN: 9781595936745. DOI: 10.1145/1291201.1291208.

Carroll Morgan. 1990. *Programming from Specifications.* (2nd ed.). Prentice Hall.

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2005. "L3: A Linear Language with Locations." In: *Typed Lambda Calculi and Applications*. Ed. by Paweł Urzyczyn. Springer Berlin Heidelberg, Berlin, Heidelberg, 293–307. ISBN: 978-3-540-32014-2.

Wojciech Mostowski, Thomas Arts, and John Hughes. Apr. 2017. "Modelling of Autosar Libraries for Large Scale Testing." In: *Workshop on Models for Formal Analysis of Real Systems (MARS@ETAPS)*. Uppsala, SE, 184–199. DOI: 10.4204/EPTCS.244.7.

Emmet Murray. 2019. "Recursive Types for COGENT." Bachelor's Thesis. Retrieved April 2022 from https://github.com/emmet-m/thesis.

George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. 2002. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." In: *Compiler Construction*. Ed. by R. Nigel Horspool. Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228. ISBN: 978-3-540-45937-8.

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. "Hyperkernel: Push-Button Verification of an OS Kernel." In: *ACM Symposium on Operating Systems Principles*. ACM, 252–269. DOI: 10.1145/3132747.3132748.

Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics with Isabelle/HOL*. Springer. ISBN: 978-3-319-10541-3. DOI: 10.1007/978-3-319-10542-0.

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Vol. 2283. Springer. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9.

Ulf Norell. 2009. "Dependently typed programming in Agda." In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (TLDI '09). ACM, Savannah, GA, USA, 1–2. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481862.

Ulf Norell. 2007. "Towards a practical programming language based on dependent type theory." PhD Thesis. Department of Computer Science and Engineering, Göteborg, Sweden.

Liam O'Connor. 2019a. "Deferring the Details and Deriving Programs." In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development* (TyDe 2019). Association for Computing Machinery, Berlin, Germany, 27–39. ISBN: 9781450368155. DOI: 10.1145/3331554.3342605.

Liam O'Connor. Aug. 2019b. "Type Systems for Systems Types." Ph.D. Dissertation. UNSW, Sydney, Australia. http://handle.unsw.edu.au/1959.4/64238.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Sept. 2016. "Refinement Through Restraint: Bringing Down the Cost of Verification." In: *International Conference on Functional Programming*. Nara, Japan.

Liam O'Connor, Zilin Chen, Partha Susarla Ajay, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Nov. 2018. "Bringing Effortless Refinement of Data Layouts to Cogent." In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, Limassol, Cyprus, 134–149. DOI: https://doi.org/10.1007/978-3-030-03418-4_9.

Peter W. O'Hearn. 2019. "Incorrectness Logic." 4, 10.1–32. DOI: 10.1145/3371078.

Bryan O'Sullivan. 2022. *aeson: Fast JSON parsing and encoding*. Retrieved August 2022 from https://hackage.haskell.org/package/aeson.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. "Cogent: uniqueness types and certifying compilation." *Journal of Functional Programming*, 31. DOI: 10.1017/S095679682100023X.

Martin Odersky. Feb. 1992. "Observers for Linear Types." In: *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*. Ed. by B. Krieg-Brückner. Lecture Notes in Computer Science 582. Springer-Verlag, 390–407.

Cyrus Omar and Jonathan Aldrich. July 2018. "Reasonably Programmable Literal Notation." *Proc. ACM Program. Lang.*, 2, International Conference on Functional Programming, Article 106, 32 pages. DOI: 10.1145/3236801.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. "Dynamic Typing with Dependent Types." In: *Exploring New Frontiers of Theoretical Informatics*. Ed. by Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell. Springer US, Boston, MA, 437–450. ISBN: 978-1-4020-8141-5.

Martin Ouimet and Kristina Lundqvist. Mar. 2007. *Formal Software Verification: Model Checking and Theorem Proving*. Tech. rep. http://www.es.mdh.se/publications/1215-.

Blaise Paradeza. 2020. "Refinement Types for COGENT." Bachelor's Thesis. Retrieved February 2022 from https://people.eng.unimelb.edu.au/rizkallahc/theses/blaise-paradeza-honours-thesis.pdf.

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. "Squid: Type-Safe, Hygienic, and Reusable Quasiquotes." In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (SCALA 2017). Association for Computing Machinery, Vancouver, BC, Canada, 56–66. ISBN: 9781450355292. DOI: 10.1145/3136000.3136005.

Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. Jan. 2021. "Data Flow Refinement Type Inference." *Proc. ACM Program. Lang.*, 5, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Article 19, 31 pages. DOI: 10.1145/3434300.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. "Translation Validation." In: *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems* (TACAS '98). Springer-Verlag, London, UK, UK, 151–166. ISBN: 3-540-64356-7. http://dl.acm.org/citation.cfm?id=646482.691453.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. "Program Synthesis from Polymorphic Refinement Types." In: *Proceedings of the 37th ACM SIGPLAN Conference on Program-*

ming Language Design and Implementation (PLDI '16). Association for Computing Machinery, Santa Barbara, CA, USA, 522–538. ISBN: 9781450342612. DOI: 10.1145/2908080.2908093.

Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 2020. "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider." In: *2020 IEEE Symposium on Security and Privacy (SP)*, 983–1002. DOI: 10.1109/SP40000.2020.00114.

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Sept. 2017. "Verified Low-Level Programming Embedded in F*." *PACMPL*, 1, ICFP, 17:1–17:29. DOI: 10.1145/3110261.

Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Aug. 2019. "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats." In: *USENIX Security*. USENIX. https://www.microsoft.com/en-us/research/publication/everparse/.

Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. "Unikernels: The Next Stage of Linux's Dominance." In: *Proceedings of the Workshop on Hot Topics in Operating Systems* (HotOS '19). Association for Computing Machinery, Bertinoro, Italy, 7–13. ISBN: 9781450367271. DOI: 10.1145/3317550.3321445.

Yann Régis-Gianas and François Pottier. 2008. "A Hoare Logic for Call-by-Value Functional Programs." In: *Mathematics of Program Construction*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Springer Berlin Heidelberg, Berlin, Heidelberg, 305–335. ISBN: 978-3-540-70594-9.

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. "SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems." In: *Proceedings of the 25th Symposium on Operating Systems Principles* (SOSP '15). Association for Computing Machinery, Monterey, California, 38–53. ISBN: 9781450338349. DOI: 10.1145/2815400.2815411.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. "QED at Large: A Survey of Engineering of Formally Verified Software." *Foundations and Trends in Programming Languages*, 5, 2-3, 102–281. DOI: 10.1561/2500000045.

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. Aug. 2016. "A Framework for the Automatic Formal Verification of Refinement from Cogent to C." In: *International Conference on Interactive Theorem Proving*. Nancy, France.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. "Comparing Libraries for Generic Programming in Haskell." In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Haskell '08). Association for Computing Machinery, Victoria, BC, Canada, 111–122. ISBN: 9781605580647. DOI: 10.1145/1411286.1411301.

Willem-Paul de Roever and Kai Engelhardt. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science 47. Cambridge University Press, United Kingdom.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. “Liquid Types.” In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI ’08). ACM, Tucson, AZ, USA, 159–169. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375602.

Mendel Rosenblum and John Ousterhout. 1992. “The Design and Implementation of a Log-Structured File System.” *ACM Transactions on Computer Systems*, 10, 26–52.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. “Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values.” In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Haskell ’08). ACM, Victoria, BC, Canada, 37–48. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411292.

John Rushby. Dec. 1981. “Design and Verification of Secure Systems.” In: *ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, USA, 12–21.

Amr Sabry and Matthias Felleisen. Jan. 1992. “Reasoning About Programs in Continuation-passing Style.” *SIGPLAN Lisp Pointers*, V, 1, 288–298.

Norbert Schirmer. 2005. “A Verification Environment for Sequential Imperative Programs in Isabelle/HOL.” In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Franz Baader and Andrei Voronkov. Springer Berlin Heidelberg, Berlin, Heidelberg, 398–414. ISBN: 978-3-540-32275-7.

Norbert Schirmer. 2006. “Verification of Sequential Imperative Programs in Isabelle/HOL.” Ph.D. Dissertation. Technische Universität München.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. “Nested Hoare Triples and Frame Rules for Higher-Order Store.” In: *Proceedings of the 23rd CSL International Conference and 18th EACSL Annual Conference on Computer Science Logic* (CSL’09/EACSL’09). Springer-Verlag, Coimbra, Portugal, 440–454. ISBN: 3642040268.

Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. “Type Targeted Testing.” In: *Programming Languages and Systems*. Ed. by Jan Vitek. Springer Berlin Heidelberg, Berlin, Heidelberg, 812–836. ISBN: 978-3-662-46669-8.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. June 2013. “Translation Validation for a Verified OS Kernel.” In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Seattle, Washington, USA, 471–481.

Denys Shabalin, Eugene Burmako, and Martin Odersky. 2013. “Quasiquotes for Scala,” 15. http://infoscience.epfl.ch/record/185242.

Natarajan Shankar. 2018. “Formalizing Hoare Logic in PVS.” In: *Engineering Trustworthy Software Systems*. Ed. by Jonathan P. Bowen, Zhiming Liu, and Zili Zhang. Springer International Publishing, Cham, 89–114. ISBN: 978-3-030-02928-9.

Tim Sheard and Simon Peyton Jones. 2002. “Template meta-programming for Haskell.” In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (Haskell ’02). ACM, Pittsburgh, Pennsylvania, 1–16. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581691.

Konrad Slind. 2021. “Specifying Message Formats with Contiguity Types.” In: *12th International Conference on Interactive Theorem Proving (ITP 2021)* (Leibniz International Proceedings in Informatics (LIPIcs)). Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:17. ISBN: 978-3-95977-188-7. DOI: 10.4230/LIPIcs.ITP.2021.30.

SPARK Pro. Retrieved April 2019 from https://www.adacore.com/sparkpro/.

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Jan. 2018. “Total Haskell is reasonable Coq.” In: *International Conference on Certified Programs and Proofs*. Los Angeles, CA, USA, 14–27.

Jacob Stanley. 2019. *Hedgehog will eat all your bugs*. Open Source Project. Retrieved April 2019 from https://github.com/hedgehogqa/haskell-hedgehog.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Jan. 2016. “Dependent Types and Multi-Monadic Effects in F*.” In: *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 256–270. ISBN: 978-1-4503-3549-2. https://www.fstar-lang.org/papers/mumon/.

Don Syme. Jan. 2006. “Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution.” In: *Proceedings of the 2006 workshop on ML*. ACM.

Akira Tanimura and Hideya Iwasaki. 2016. “Integrating Lua into C for Embedding Lua Interpreters in a C Application.” In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (SAC ’16). Association for Computing Machinery, Pisa, Italy, 1936–1943. ISBN: 9781450337397. DOI: 10.1145/2851613.2851747.

Gunnar Teege. 2019. *Dargent Proposal*. Retrieved February 2022 from https://github.com/au-ts/cogent/issues/316.

The Coq Proof Assistant. Retrieved September 2022 from https://coq.inria.fr/.

The FUSE Project. Retrieved April 2019 from https://github.com/libfuse/libfuse.

The Haskell Lightweight Virtual Machine (HaLVM) source archive. Retrieved April 2019 from https://github.com/GaloisInc/HaLVM.

The Rust Programming Language. Retrieved January 2022 from https://www.rust-lang.org.

The seL4 Microkernel. Retrieved Mar. 4, 2018 from https://sel4.systems/.

The Cogent team. 2023. *Cogent Project*. Retrieved February 2023 from https://github.com/au-ts/cogent.

Ed. by Marco Bernardo and Valérie Issarny. “Model-Based Testing and Some Steps towards Test-Based Modelling.” *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 297–326. ISBN: 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_9.

UBI - Unsorted Block Images. Retrieved December 2021 from http://linux-mtd.infradead.org/doc/ubi.html.

Using the GNU Compiler Collection (GCC). Retrieved December 2021 from https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html.

Mark Utting, Alexander Pretschner, and Bruno Legeard. Aug. 2012. “A Taxonomy of Model-Based Testing Approaches.” *Softw. Test. Verif. Reliab.*, 22, 5, 297–312. DOI: 10.1002/stvr.456.

Dale Vaillancourt, Rex Page, and Matthias Felleisen. 2006. “ACL2 in DrScheme.” In: *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications* (ACL2 ’06). ACM, Seattle, Washington, USA, 107–116. ISBN: 0-9788493-0-2. DOI: 10.1145/1217975.1217999.

Niki Vazou. 2016. “Liquid Haskell: Haskell as a Theorem Prover.” Ph.D. Dissertation. University of California, San Diego, USA.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. "LiquidHaskell: Experience with Refinement Types in the Real World." In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. ACM, Gothenburg, Sweden, 39–51. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633366.

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. "LoCal: A Language for Programs Operating on Serialized Data." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019). Association for Computing Machinery, Phoenix, AZ, USA, 48–62. ISBN: 9781450367127. DOI: 10.1145/3314221.3314631.

Philip Wadler. 1990. "Linear types can change the world!" In: *Programming Concepts and Methods*.

David Walker. 2005. "Substructural Type Systems." In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. MIT Press. Chap. 1, 3–43.

David Walker and Kevin Watkins. 2001. "On Regions and Linear Types (Extended Abstract)." In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (ICFP '01). Association for Computing Machinery, Florence, Italy, 181–192. ISBN: 1581134150. DOI: 10.1145/507635.507658.

Yan Wang and Verónica Gaspes. Jan. 2011. "An embedded language for programming protocol stacks in embedded systems." In: *ACM PEPM*. Austin, TX, US, 63–72. http://doi.acm.org/10.1145/1929501.1929511.

Declan Warn. 2021. "Extending the Debugging Capabilities of the COGENT Compiler." Bachelor's Thesis. Retrieved June 2022 from https://people.eng.unimelb.edu.au/rizkallahc/.

Makarius Wenzel. Dec. 2021. *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/doc/isar-ref.pdf.

David Woodhouse. 2001. "JFFS: the journalling flash file system." In: *Ottawa Linux Symposium*.

A.K. Wright and M. Felleisen. Nov. 1994. "A Syntactic Approach to Type Soundness." *Inf. Comput.*, 115, 1, 38–94. DOI: 10.1006/inco.1994.1093.

Hongwei Xi. 2017. "Applied Type System: An Approach to Practical Programming with Theorem-Proving." abs/1703.08683. http://arxiv.org/abs/1703.08683 arXiv: 1703.08683.

Li-yao Xia. 2022. *quickcheck-higherorder: QuickCheck extension for higher-order properties*. Retrieved September 2022 from https://hackage.haskell.org/package/quickcheck-higherorder.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. "Static Contract Checking for Haskell." In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '09). ACM, Savannah, GA, USA, 41–52. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480889.

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. Aug. 2021. "GhostCell: Separating Permissions from Data in Rust." *Proc. ACM Program. Lang.*, 5, International Conference on Functional Programming, Article 92, 30 pages. DOI: 10.1145/3473597.

Qianchuan Ye and Benjamin Delaware. 2019. "A Verified Protocol Buffer Compiler." In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2019). Association for Computing Machinery, Cascais, Portugal, 222–233. ISBN: 9781450362221. DOI: 10.1145/3293880.3294105.

Nobuko Yoshida, Kohei Honda, and Martin Berger. 2007. "Local State in Hoare Logic for Imperative Higher-Order Functions." In: *Proc. Fossacs* (Lecture Notes in Computer Science). Vol. 4423. Springer, 361–377.

Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. "A Formally Verified NAT." In: *Proceedings of the Conference of the ACM Special Interest Group* *on Data Communication* (SIGCOMM '17). Association for Computing Machinery, Los Angeles, CA, USA, 141–154. ISBN: 9781450346535. DOI: 10.1145/3098822.3098833.

He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. "Learning Refinement Types." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2015). Association for Computing Machinery, Vancouver, BC, Canada, 400–411. ISBN: 9781450336697. DOI: 10.1145/2784731.2784766.