Dependent Type Theories for the Verification of Unstructured Programs

Germán Andrés Delbianco

Abstract

An ever present challenge in formal verification, specially as it comes to stateful reasoning, is modularity. But, not so much modularity of the programs themselves as proof-modularity, that is, the posibility to reuse the proofs of correctness of modular programs. Since proving properties of complex programs is hard, it is preferable to have to do them only once. Unfortunately, this is not always possible.

Although the state of the art has advanced significantly in this regard, most logics for modular stateful reasoning address *structured* programming idioms. One can even argue that structured programming has driven the design of most logics for stateful reasoning, making most of the latter developments unsuitable for modular reasoning with *unstructured* programming constructs.

The main goal of this thesis is the development of dependent type theories aimed at the modular verification of unstructured stateful programs. As particular cases of unstructured programming idioms that have not, historically, got along well with logics for reasoning modularly about state, the thesis will focus on two: continuations and shared-memory concurrency. I believe that by developing such features in a dependent type theory, one can device the appropiate setting for the verification of complex unstructured programs in a proof-modular manner.

Overview

The issue of software correctness is a long-standing problem in the computer science community, and it has always been relevant. Nowadays, even more so with the software industry becoming increasingly aware of the importance and benefits of formal verification. This comes as a consequence of realizing that having mathematical proof of the correctness of software systems (or at least particular, critical, smaller components) is more efficient, even from an economical standpoint, than relying on *a posteriori* cycles of testing, debugging and re-implementing. The problem with formal verification is that it is closely connected to the semantic models on which programming languages are developed, usually involving complex mathematics. As technology paces forward, developing new logics to keep up becomes even harder.

The state of the art in formal verification has become so mature a discipline that its underlying theories are becoming not only a standard for reasoning with computer programs but also, they are considered rigorous enough to undertake the formalization of mathematics itself [3, 15]. However, there are fundamental question still unanswered and, in this thesis, I will address the issue of modularity in the context of stateful reasoning with unstructured programming constructs.

The problem with modular formal verification of stateful programs is not only the modularity of the programs themselves, but most importantly proof-modularity i.e. the ability to reuse the proofs of correctness of modular programs at different times. For instance, in the foundational development of Hoare Logic [18] the proofs for two modularly developed stateful programs e.g. a client and a server for a complicated data structure could not be separated from each other. If at some point the internal implementation of the data structure changed, it meant that the proof for the whole system had to be redone including the one corresponding to parts that remained invariant. Since then, the state of the art has made significant advances with regards to modular verfication, with O' Hearn, Reynolds et. al. s Separation Logic [28, 33] becoming the standard logic for reasoning with stateful computations, but with a steeper development in reasoning with well-structured programs. Understandably, because the latter are easier to reason with and have more accesible mathematical models.

As particular cases of unstructured programming constructs that have not, historically, got along well with logics for reasoning modularly about state, this thesis will focus on two: unstructured control flow and unstructured shared-memory concurrency.

Unstructured control flow is sometimes disregarded as a long-forsaken bad engineering practise mostly beacuse of the ill-famous `goto` command [10]. There exists, however, further sensible programming constructs for unstructured control flow, continuations being one of them. *Continuations* are powerful abstractions that model the "future of a computation" [32]. They have a ubiquitous presence in programming languages: they allow for a family of program transformation techniques in the style of many CPS transformations [8], they underlie the denotational semantics of programs with jumps [12, 35], they give computational content to classical proofs [16], they have been used to structure computational effects [13, 19] and also to design compilation techniques [2]. In spite of this, there has been little effort towards developing verification logics for language with first class continuations.

As to shared-memory concurrency, we live in a world of massively concurrent software systems running over increasing multi-processing power. In such a context, it is natural to expect programmers– and thus programming languages– to desire to exploit the available parallelism in order to produce more efficient software. Reasoning about concurrent programs is difficult as it often entails considering all possible interactions between different threads. Unfortunately, this forces verification experts to abandon the safe waters of the sequential world where most of the advances for modular stateful reasoning have been made. Furthermore, most of the effort has been towards more modular-friendly well-structured concurrency verification [34, 27], whereas real-world programming languages trend to go towards the unstructured end of *fork/join* concurrency [17], where existing programming logics have unresolved issues as regard scalability and modularity.

In a more general perspective, I believe that the challenges of software verification should be addressed in systems where the simultaneous development of programs and proofs can be carried out in an integrated way, allowing that the same mechanisms for structuring and reuse be applied to both software development and their proofs of correctness. By judiciously structuring programs and proofs together, one can oversee that the proof burden introduced by verification does not blow up, while at the same time create new abstractions to reason with our programs.

I believe that Type Theory, in particular dependently-typed theories such as the one underlying Coq [5, 23] is a strong, mature framework in which undertaking this task. In dependently-typed theories, programs and their proofs are indivisible, dual manifestations of the same phenomenon, namely the inhabitants of dataypes. Dependent Type theory is modular and supports the higher-order features that unstructured programming requires, but is usually *purely functional* i.e., there are no computational effects. In such type theories, one could develop programs and their correctness proofs, but the programs have to be purely functional and terminating. There is however, a suitable precedent towards my goal.

Hoare Type Theory [25, 26] addresses the difficulty of reconciling proving with impure programming features by encapsulating sequential, stateful reasoning through computational types. This encapsulation, however, is quite different from the usual approach where one implements the abstract syntax of the programming language of choice as a datatype in the meta logic, and then reasons about abstract syntax trees. Instead, HTT uses Coq directly as a programming language, thus removing a level of indirection. HTT has carried out the initial steps to incorporate stateful reasoning through effectful dependent type theories, but it has only considered one effect: dynamic state in a sequential setting. I propose to use this experience as a stepping-stone to tame the verification of the unstructured programming features described above.

Research Statement

The foremost objective of this thesis is the development of dependent type theories aimed at the modular verification of unstructured stateful programs. As particular cases of computational effects that have not, historically, got along well with logics for reasoning modularly about state, the thesis will focus on two: continuations and shared-memory concurrency.

To this end, I will extend the type underlying dependently-typed theory of Coq with computational types which encapsulate the imperative features under study. This has been done before, in HTT, but for a language with dynamic state in a sequential setting with no support for reasoning with *unstructurered* effects. However, since both *continuations* and *concurrency* have been studied from a computational effects perspective [19, 36, 1], I am confident that those features can be embedded into a dependent type theory, following this methodology, and thus enabling modular stateful verification of unstructured programming.

As I have discussed before, *continuations* are ubiquitous in the programming languages community. Although the state of the art concerning formal reasoning about them is vast, it has focused predominantly on the semantic modelling of higher-order control operators and CPS transformations, and verification of programs directly in the semantic model. In contrast, one of the goals of this thesis is to develop a Hoare-style logic in which one can systematically specify and verify full functional correctness of programs with higher-order jumps, in the presence of dynamic mutable state and first class continuations i.e., where continuations can be returned as the result of computations, stored in the heap or passed as argument of high-order programs.

The first step towards that goal was the development of $\mathsf{HTT}_{\mathsf{cc}}$ [9], a higher-order type theory for verification of programs with `callcc` control operators. $\mathsf{HTT}_{\mathsf{cc}}$ supports mutable state in the style of Separation logic, and, to the best of our knowledge, is the first Hoare logic or type theory to support the combination of higher-order functions, mutable state and control operators. In the future I plan to provide further examples of the scalibility of this logic, providing more complex real - world examples from the concurrency domain, which are discussed below. Additionally, logics for more complex models of continuations, such as *delimited continuations* [11] can be implemented following the same methodology.

As for shared-memory concurrency there are two well established styles of program logics, customarily divided according to the supported kind of granularity of program interference. Logics for coarse-grained concurrency such as O'Hearn et. al.s Concurrent Separation Logic (CSL) [27] restrict the interference to critical sections only, but generally lead to more modular specifications and simpler proofs of program correctness. Logics for *fine-grained* concurrency such as Jones' Rely-Guarantee (RG) [21] admit arbitrary interference, but their specifications have traditionally been more monolithic.

The goal of this thesis in this regard is to identify the essential ingredients required for compositional specification of concurrent programs, and combine them in a novel way to reconcile the two approaches. A first step towards that goal are *fine-grained resources* [24, 22]: a novel model of concurrent computations with shared memory which allow for a simple, yet powerful, logical framework for uniform Hoare-style reasoning about partial correctness of coarse- and fine-grained concurrent programs in the *well-behaved manner* of CSL. I propose to build upoon this result, developing new logics that extend this reasoning to the less-structured approach of fork/join primitives present in real world libraries such as Pthreads.

What is more, further logics for other approaches to concurrency as *futures* [14] or coroutines [6] could be considered. A *future* represents the result of an asynchronous computation wich cannot be accesed until is completion. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. *Coroutines* are language primitives that enable cooperation between asynchronous, resumable threads. They can be implemented using continuations in a setting with *higher-order store*, i.e. one where computations can be stored in the heap, and thus a programming logic for such features is not only a natural extension of the work proposed above, but also a suitable bridge between the two pillars of this thesis: continuations and unstructured shared variable concurrency.

Methodology

The first stage of this research process is identifying examples that qualify as good candidates for formal verification. These are chosen either because their proofs of correctness are not proof-modular in the current state of the art, as in the case of shared variable concurrency, or they are not verifiable at all, as for continuations in combination with dynamic mutable state.

The following step is the development of new logics for stateful modularly reasoning with such programs. The key in the chosen methodology is to perform a *shallow embedding* of the proposed logic for stateful reasoning into the otherwise purely functional dependent type theory of Coq. The underlying type theory is extended with computational types that encapsulate stateful reasoning with continuations and, respectively, shared variable concurrency. It should be noticed that this encapsulation of effects preserves the soundness of the system.

After that, other features of the verification framework aimed at improving the verification experience are developed: logical rules, derived lemmas, tactics, etc. Finally, a verification survey will be performed, in order to assess the resulting logics, and also to investigate further particulars that might arise.

Positioning of the Proposed Research

Hoare-style stateful reasoning through dependent types

Hoare Type Theory [25, 26] has been a recent development towards reconciling stateful reasoning with dependent type theories. HTT formalises Separation Logic [28, 33] in a dependently-typed setting, extending the purely-functional type theory underlying Coq [5, 23] to integrate Separation Logic into it. Its novelty is that its philosophy is quite different from the usual approach where one implements the abstract syntax of the programming language of choice as a datatype in the meta logic, and then reasons about abstract syntax trees. Instead, HTT uses Coq directly as a programming language, thus removing a level of indirection.

The encapsulated effectful programs are classified by means of *Hoare types*, which also serve as specifications in the style of Separation Logic. A program has a Hoare type $\{P\}\,A\,\{Q\}$ if it is provably safe to run in a state satisfying the precondition P, and either diverges, or terminates with a value of type A, in a state satisfying the postcondition Q. Hoare types provide all the facilities usually required for working in Hoare logic *e.g.* specifying an invariant for a loop corresponds to providing a Hoare type for a recursive function. In such a framework, one can develop higher-order stateful programs, carry out proofs of their full functional correctness, and check the proofs mechanically. Programs and proofs can be organized into verified libraries, fulfilling the premise that modular programs should have modular, i.e. reusable, proofs.

All the logics and systems described above where designed to reason with *dynamic mutable state in a sequential setting* and they do not consider other computational effects beyond state. I propose to incorporate ideas for combining effects from the computational effects community [30, 19], and following HTT's philosophy develop new dependent type theories for stateful reasoning with continuations and concurrency.

Hoare-style logics for higher-order control

Crolard and Polonowski [7] have recently developed a Hoare logic for control operators, in which specifications are carried out in types. While in this respect, the approach is similar to the one proposed in $\mathsf{HTT}_{\mathsf{cc}}$ from the high-level point of view, there is a number of differences. For example, Crolard and Polonowski only consider mutable stack variables with block scope, but no pointers or aliasing. Procedures are not allowed to contain free variables, and type dependencies contain first-order data only, such as natural numbers. Berger [4] presents a first-order Hoare logic for `callcc` in an otherwise purely functional language.

In contrast, I propose to follow HTT's methodology in order to develop a framework for Hoare-style reasoning with higher-order programs, control effects and mutable, dynamic state. The first step towards that goal is $\mathsf{HTT}_{\mathsf{cc}}$, a dependent type-theory with first class continuations which uses computational types indexed by pre- and postconditions as our specifications in the style of Separation logic. Additionally, $\mathsf{HTT}_{\mathsf{cc}}$ features *algebraic* control operators, initially introduced by Jaskelioff [20].

A conclusion from this experience is that algebraic operators require less manual program annotations, than the non-algebraic variants. Consequently, $\mathsf{HTT}_{\mathsf{cc}}$ eases the verification of significant examples involving complicated data structures, solving further issues than just providing a sound rule for the control operators.

I plan to extend $\mathsf{HTT}_{\mathsf{cc}}$with *high-order state*, thus enabling the implementation and verification of coroutines and other concurrency primives *à la* CML [31]. Furthermore, more complex approaches to control effects such as as *delimited or composable continuations* [11] are a natural, feasible extension of this work. To the best of my knowledge, there is no verification framework that can reason with *undelimited* control effects and dynamic mutable state.

Shared-variable Concurrent Reasoning

Before I have mentioned that logics for shared variable concurrency can be classified in two families, grouped by the granularity of the interference allowed between concurrent threads.

Logics for *coarse-grained* concurrency, such as Concurrent Separation Logic (CSL) [27], employ *shared resources* and associated *resource invariants* [29], to abstract the interference between threads. A resource r is a chunk of shared state, and a resource invariant I is a predicate over states, which holds of r whenever all threads are outside the critical section. By mutual exclusion, when a thread enters a critical section for r, it acquires ownership and hence exclusive access to r's state. The thread may mutate the shared state and violate the invariant I, but it must restore I before releasing r and leaving the critical section.

Logics for *fine-grained concurrency*, such as Rely-Guarantee (RG) [21], admit arbitrary interference i.e., threads can read or write on shared variables at any time. The interaction between threads is directly specified by rely and guarantee transitions on states. A *rely* specifies the thread's expectations of state transitions made by its environment. A *guarantee* specifies the state transitions made by the thread itself. RG is more expressive than CSL because transitions can encode arbitrary protocols on shared state, whereas CSL is specialized to a fixed mutual exclusion protocol on critical sections. But, CSL is more compositional in manipulating resources. Where a CSL resource invariant specifies the behavior of an individual chunk of shared state, the transitions in RG treat the whole state as monolithically shared.

This thesis aims to identify the essential ingredients required for compositional specification of concurrent programs, and combine them in a novel way to reconcile the two approaches. A first step towards that goal is the development of *fine-grained resources* [24, 22]. A fine-grained resource is specified by a resource invariant, as in CSL, but it also adds transitions in the form of relations between resource states. This transitions characterize the possible changes the threads can make to the shared state.

The logics described before verify concurrent programs written in languages where concurrency is introduced by means of the parallel composition operator $||$, or *par*. Given two programs $p1$ and $p2$, *par* composes them concurrently, creating a new program $p1 \mid\mid p2$ where resources are shared amongst two threads, with $p1$ and $p2$ potentially racing for a shared resource. This syntactic, *well-bracketed* approach to introducing concurrency in a language is prefered from a logic design perspective because it favours reasoning inductively and fosters modular reasoning. However, in most real world concurrent programming environments [17], threads are spawned dynamically using *fork* primitives. This practice enables more patterns for concurrent programming than $||$, but also hinders the modular verification of programs.

The objective of this thesis is to build upon the experience with *fine-grained resources* in order to develop new logics that can tackle the modular verification of the `fork/join` primitives present in real world libraries such as `Pthreads`. Furthermore, I plan to investigate into other models of shared-variable concurrent programming such as coroutines or cooperative threads.

References

[1] Abadi, M., and Plotkin, G. D. A model of cooperative threads. In *POPL* (2009), Z. Shao and B. C. Pierce, Eds., ACM, pp. 29–40.

[2] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Avigad, J., and Harrison, J. Formally verified mathematics. *Commun. ACM 57*, 4 (Apr. 2014), 66–75.

[4] Berger, M. Program logics for sequential higher-order control. In *FSEN* (2009), F. Arbab and M. Sirjani, Eds., vol. 5961 of *Lecture Notes in Computer Science*, Springer, pp. 194–211.

[5] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] Conway, M. E. Design of a separable transition-diagram compiler. *Commun. ACM 6*, 7 (July 1963), 396–408.

[7] Crolard, T., and Polonowski, E. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *J. Log. Algebr. Program. 81*, 3 (2012), 181–208.

[8] Danvy, O., and Filinski, A. Representing control: A study of the CPS transformation. *MSCS 2*, 4 (1992), 361–391.

[9] Delbianco, G. A., and Nanevski, A. Hoare-style reasoning with (algebraic) continuations. In *ICFP* (2013), G. Morrisett and T. Uustalu, Eds., ACM, pp. 363–376.

[10] Dijkstra, E. W. Letters to the editor: Go to statement considered harmful. *Commun. ACM 11*, 3 (Mar. 1968), 147–148.

[11] Felleisen, M., Friedman, D. P., Duba, B., and Merrill, J. Beyond Continuations. Tech. Rep. 216, Indiana University, 1987.

[12] Felleisen, M., Wand, M., Friedman, D., and Duba, B. Abstract continuations: a mathematical semantics for handling full jumps. In *LISP and functional programming* (1988).

[13] Filinski, A. Representing monads. In *POPL* (1994).

[14] Flanagan, C., and Felleisen, M. The semantics of future and an application. *J. Funct. Program. 9*, 1 (1999), 1–31.

[15] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S. L., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. A machine-checked proof of the odd order theorem. In *ITP* (2013), S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., vol. 7998 of *Lecture Notes in Computer Science*, Springer, pp. 163–179.

[16] Griffin, T. A formulae-as-types notion of control. In *POPL* (1990), pp. 47–58.

[17] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[18] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580.

[19] Hyland, M., Levy, P. B., Plotkin, G. D., and Power, J. Combining algebraic effects with continuations. *Theor. Comput. Sci. 375*, 1-3 (2007), 20–40.

[20] Jaskelioff, M. Modular monad transformers. In *ESOP* (2009), G. Castagna, Ed., vol. 5502 of *Lecture Notes in Computer Science*, Springer, pp. 64–79.

[21] Jones, C. B. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 5*, 4 (Oct. 1983), 596–619.

[22] Ley-Wild, R., and Nanevski, A. Subjective auxiliary state for coarse-grained concurrency. In *POPL* (2013), R. Giacobazzi and R. Cousot, Eds., ACM, pp. 561–574.

[23] The Coq development team. *The Coq proof assistant reference manual.* TypiCal Project, 2012. Version 8.4.

[24] Nanevski, A., Ley-Wild, R., Sergey, I., and Delbianco, G. A. Communicating state transition systems for fine-grained concurrent resources. In *ESOP* (2014), Z. Shao, Ed., vol. 8410 of *Lecture Notes in Computer Science*, Springer, pp. 290–310.

[25] Nanevski, A., Morrisett, J. G., and Birkedal, L. Hoare type theory, polymorphism and separation. *J. Funct. Program. 18*, 5-6 (2008), 865–911.

[26] Nanevski, A., Vafeiadis, V., and Berdine, J. Structuring the verification of heap-manipulating programs. In *POPL* (2010), M. V. Hermenegildo and J. Palsberg, Eds., ACM, pp. 261–274.

[27] O'Hearn, P. W. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375*, 1-3 (2007), 271–307.

[28] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local reasoning about programs that alter data structures. In *CSL* (2001), L. Fribourg, Ed., vol. 2142 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[29] Owicki, S. S., and Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19*, 5 (1976), 279–285.

[30] Plotkin, G. D., and Power, J. Notions of computation determine monads. In *FoSSaCS* (2002), M. Nielsen and U. Engberg, Eds., vol. 2303 of *Lecture Notes in Computer Science*, Springer, pp. 342–356.

[31] Reppy, J. H. *Concurrent Programming in ML.* Cambridge University Press, Cambridge, England, 1999.

[32] Reynolds, J. C. The discoveries of continuations. *LISP and Symbolic Computation 6*, 3-4 (1993).

[33] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *LICS* (2002), IEEE Computer Society, pp. 55–74.

[34] Reynolds, J. C. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS* (2004), K. Lodaya and M. Mahajan, Eds., vol. 3328 of *Lecture Notes in Computer Science*, Springer, pp. 35–48.

[35] Strachey, C., and Wadsworth, C. P. Continuations: A mathematical semantics for handling full jumps. *HOSC 13*, 1/2 (2000).

[36] Thielecke, H. Control effects as a modality. *JFP 19*, 1 (2009).