CS290C Dependent Types Write Up

Dependent Types and Impure Computation

by

Tegan Brennan

March 21, 2015

1 Introduction

Dependent types are a powerful tool that allow for a high degree of verification and specification to be integrated into programming. As terms may appear in types, the type of a function can provide a very detailed specification about its behavior. This results in the type checker being able to catch a whole class of run-time errors, such as attempting to access an out-of-bounds array index or dividing by zero, missed by languages with less expressive type systems. While pure dependently types languages are easy to reason about and provide a high degree of safety, their use as a general all purpose programming language is limited by their requirement that all computations be pure and terminating in order to guarantee soundness [12]. In real life many useful applications have side effects. They might have state, communicate across networks or interact with users. Ideally, there would be some way to reconcile a dependently typed functional language with imperative features such as mutable states and non-termination while maintaining strong guarantees about the correctness of the effectful program.

2 A Very Brief Review of Dependent Types

We are all familiar with types that depend on other types, i.e. a list of integers, List<Integer>. Dependent types are types that depend on terms, or elements of types. A list of integers of length three, List<Integer><3>, would be an example of a dependent type. Here the size of the list is encoded in the type, allowing a type checker to confirm that functions such as element wise subtraction of lists are only executed on lists of the appropriate length. At present, a number of dependently typed languages exist through which one may explore the potential of dependently typed programming. They include the functional programming languages Agda, Idris, Epigram, Coq, and Cayenne and well as the imperative languages Xanadu and Ynot. Each of theses languages has had to answer the challenge of how to handle impure program fragments and the rest of this write-up will devote itself to some of the more interesting answers to this question.

3 Hoare Type Theory

3.1 Hoare Logic

Over the past 40 years, Hoare logic has been used to specify and prove imperative programs [7]. Hoare logic uses first order logic assertions in order to describe memory segments or heaps. The derivable judgments of Hoare logic are Hoare triples

$$\{P\}\ \ E\ \ \{Q\}$$

which specify the behavior of an effectful computation. In the above triple, E is the computation for which the specification is given and P and Q are assertions about the heap, or state of the program, before and after the computation is run. These are respectively called the pre- and postconditions of the computation. Assuming that P was satisfied before the effectful program fragment E was run, then Q is satisfied after the computation takes place. How can we show that this is the case? We can use a set of inference rules called Hoare rules given below [15].

$$\textbf{SKIP}\ \frac{}{\{P\}\ \text{skip}\ \{P\}} \qquad \textbf{ASSIGN}\ \frac{}{\{P[a/x]\}x{:=}a\{P\}}$$

$$\textbf{IF}\ \frac{\{P{\wedge}b\}\ c1\ \{Q\} \qquad \{P{\wedge}\neg b\}\ c2\ \{Q\}}{\{P\}\ \text{if b then c1 else c2}\ \{Q\}}$$

$$\textbf{SEQ}\ \frac{\{P\}\ c1\ \{R\} \qquad \{R\}\ c2\ \{Q\}}{\{P\}\ \ c1;c2\ \ \{Q\}}\ \textbf{WHILE}\ \frac{\{P{\wedge}b\}\ c\ \{P\}}{\{P\}\ \text{while b do c}\ \{P{\wedge}\not{b}\}}$$

$$\textbf{CONSEQUENCE}\ \frac{(P{\Rightarrow}P')\ \{P'\}\ c\ \{Q'\}\ (Q'{\Rightarrow}Q)}{\{P\}\ c\ \{Q\}}$$

The **SKIP**, **ASSIGN**, **SEQ**, and **IF** rules are self explanatory; the **WHILE** rule is essentially a loop invariant, and the **CONSEQUENCE** rule allows preconditions to be strengthened and postconditions weakened. This set of rules forms an inductive definition of Hoare logic where $\{P\}c\{Q\}$ is said to be a theorem if there is a finite proof tree for it. Hoare logic is sound and relatively complete (Hoare logic is complete provided there are proofs for the assertions $(P \Rightarrow P')$ and $(Q' \Rightarrow Q)$ in the **CONSEQUENCE** rule). [4]

3.2 Hoare Type Theory

From the basic overview of Hoare logic given above, one can begin to understand how it might be use in order to provide specifications for effectful computation. Traditional methods of doing so; however, have not integrated Hoare logic into the type system itself, instead separating the specifications encapsulated by the pre- and postconditions from the actual type system. Recently the YNot group from Harvard, led by Nanevski and Morrisett, have proposed a method for integrating dependent types and Hoare-style logic into a language that supports both strong typing guarantees and imperative commands called Hoare Type Theory. [9]

The key mechanism of this theory lies in the introduction of a Hoare type to classify impure programs. The Hoare type $\Psi.X.\{P\}x : A\{Q\}$ specifies an effectful computation returning a result of type A with precondition P and postcondition Q. It thus serves as a way to specify and describe the effects of imperative commands. The contexts Ψ and X list the variables and heap variables respectively that may appear in both P and Q. These are known as **ghost** or **local variables** and they may appear only in assertions, not in the computations nor in the type A. The constructor for the Hoare type is **do** E, which encapsulates the effectful computation E and suspends its evaluation. The name was chosen in order to resemble Haskell style do notation which creates an anonymous monad in order to specify an effectful program. The elimination form for the Hoare type is given by x ← K;E which activates the suspended computation determined by K. When the computation is executed, all side effects that might have have been suspended are performed.

Other types of HTT include booleans and natural numbers, the unit type, dependent functions and polymorphic types of the form $\forall\alpha.A$. It's important to note that the type $\forall\alpha.A$ polymorphically quantifies over the monotype variable α where a monotype is defined as any type that does not contain polymorphic quantification, except in the assertions. This constraint leaves a type system expressive enough for modelling languages such as Standard ML but not languages such as Haskell. Potentially extending the type system to include impredicative polymorphism is discussed later.

3.2.1 Heaps

For an implementation of Hoare logic to make sense, there must be some way to describe and manipulate the heaps on which the pre and postconditions are defined. Most implementations of HTT, including that of the YNot group, model memory locations as finite numbers and heaps as finite functions mapping a location N to a pair (τ, M) where τ is the monotype of M. Heaps are limited to only values of monotypes for the same reason that polymorphic quantification is.

We now return to our definition of Hoare types. As the precondition, P, specifies the state of the program before the computation is run, the precondition needs to be given the initial heap as an argument. It is hence of type **heap** → **prop**. The postcondition relates the state before the computation to the state immediately after and hence must be given both the initial and final heaps. It is thus of the type **heap** → **heap** → **prop**. A computation of the Hoare type $\{P\}x : A\{Q\}$ can thus run in a heap i such that P i holds and will either diverge or reach a heap m and return a value of type T such that Q[v/x] i m holds.

As a simple demonstration, consider the following function for allocation:

$$\text{alloc: } \forall\alpha.\Pi x : \alpha.\{emp\}y : nat\{y \to x_\alpha\}$$

In the above function **emp** is used to denote that the current heap is empty and $y \to x_\alpha$ to denote that the current heap consists of a single location y which points to the term x of type α. The above function thus simply returns the address y which is initialized with the value x of type α and acts on a program whose current heap satisfies the precondition P. If we assume that the pre- and postconditions of a computation must be satisfied by the global heap, then our function alloc may only act when the global heap is empty. What if we wanted to run the program in a heap that already contained entries? By our assumption above, a different function would need to be constructed. This is not only inconvenient but also unneeded. An appeal to separation logic solves this problem.

The introduction of separation logic adds the frame rule to HTT.

$$\frac{\{P\}\ c\ \{Q\}}{\{P{*}R\}\ c\{Q{*}R\}}$$

where $*$ is the separation conjunction used to indicate a disjoint heap union. [7] By the above rule, if the effectful computation c converts a heap satisfying P into one that satisfies Q, then this computation may be applied to a larger heap and will act without modifying the excess part. This means that the global heap need only contain a subheap satisfying the precondition. This subheap is then changed by the computation so that the postcondition is satisfied while the remaining part of the global heap is guaranteed to remain invariant. This is called the **small footprint** specification.

3.2.2 Equality in HTT

As with all dependently typed systems, checking for equality of types in HTT involves comparing terms that appear in types. This is an undecidable problem in any Turing complete language, requiring that HTT face the problem of all dependently type systems: finding a balance between preciseness and decidability. The YNot group at Harvard tackles this problem by introducing two different notions of equality — definitional equality and propositional equality. Definitional equality is decidable. Testing for this type of equality involves reducing both terms to their canonical forms through normalization and basing their equality around their syntactical equivalence. The YNot team constrains the definition of equality to definitional in the typechecking stage, making typechecking itself decidable in HTT. Propositional equality on the other hand is both undecidable and a finer measure of equality. The Ynot team restrains the use of this measure of equality to proving. Typechecking is thus separated from the proving of program specifications. [9]

4 Effects

Another avenue to handle effectful programs in dependent type theory is through the use of algebraic effects. An effect system labels each function with its possible effects. A function with effects delimited by σ, for example, would be written as $(\tau \xrightarrow{\sigma} \tau')$. The dependently typed functional programming language Idris already has an **effects** library for managing computational effects atop of its support for monads. It is through this library that I will explore effect systems. Example programs are available on my github page (https://github.com/tsbrennan1/

dptexamples).

The **effects** library supports a large variety of impure computations, including stateful operations, non-determinism, I/O operations, exceptions and random numbers. These are all example of the type EFFECT. Every effect is associated with a resource. For example, a state effect storing a counter would contain the integer value of the counter intself as its resource. Effects are algebraic data types in which the constructor describes the operations provided by the effect. For example, the state effect might have two operations — get, to return the resource currently stored in the state effect, and put, which takes an argument and updates the resource of the state effect to be of the appropriate type. Effectful programs themselves are given the following type

$$\text{Eff: } (\text{x:Type}) \to \text{List EFFECT} \to (\text{x} \to \text{List EFFECT}) \to \text{Type}$$

From the above, we see that effectful programs are parametrized over the result type x, a list of input effects, and a list of input effects which is computed using the result of the program. This last parameter means that running an effectful program can change the list of effects available. For example, a state effect may include a dependent type, such as a vector, as its resource and a computation may increase of decrease that vector's size.

Wadler and Thiemann have shown that the a specific effect system, that of Talpin and Jouevelot, carries over directly to an analogous system for monads, and provide a technique that should allow one to transpose any effect system into a corresponding monad system. [14] Nevertheless, Edwin Brady provides some reasons why Idris' **effects** library might be preferred to the use of monads when both options present themselves. [1] In particular, he highlights the ease of writing programs which compose multiple effects compared to the unwieldiness of using monad transformers. Effects can be implemented individually and combined in a list with no further effort. In the case of Idris' **effects** library though, it is important to note that algebraic effects cannot capture all monads. It is thus important for both to remain options.

5 Conclusion and Related Work

Hoare type theory and effect systems provide separate avenues for handling impure operations in pure dependently typed systems. Both methods involve attempts to exploit the benefits of impure computation while delimiting their scope and maintaining the strong guarantees of the dependent type system. Research in this area remains far from complete. Recently, there have been attempts to extend Hoare Type Theory to support impredicative polymorphism [11] and Edwin Brady has recently attempted to use the effects system to reason about programs in the presence of run-time state transitions [3]. Another potential solution might lie in linear dependent types, a type system that integrates dependent and linear types. [6] provides a detailed explanation of this integration and how it allows us to decompose the specification information found in Hoare triples into simpler type theoretic connectives. A major advantage of this approach is that it results in a much simplified equational theory than HTT does.

References

[1] Brady, Edwin. ”Programming and reasoning with algebraic effects and dependent types.” ACM SIGPLAN Notices. Vol. 48. No. 9. ACM, 2013.

[2] Brady, Edwin. ”Programming and Reasoning with Side-Effects in IDRIS.” (2014).

[3] Brady, Edwin. ”Resource-Dependent Algebraic Effects.” Trends in Functional Programming. Springer International Publishing, 2015. 18-33.

[4] Cook, Stephen A. ”Soundness and completeness of an axiom system for program verification.” SIAM Journal on Computing 7.1 (1978): 70-90.

[5] Dal Lago, Ugo, and Marco Gaboardi. ”Linear dependent types and relative completeness.” Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on. IEEE, 2011.

[6] Krishnaswami, Neelakantan R., Pierre Pradic, and Nick Benton. ”Integrating Dependent and Linear Types.” Retrievable from: https://www. mpi-sws. org/neelk/dlnl-paper. pdf.

[7] Makarov, Evgeny, and Christian Skalka. ”Formalized Proof of Type Safety of Hoare Type Theory.” (2010).

[8] Nanevski, Aleksandar, et al. ”Abstract predicates and mutable adts in hoare type theory.” Programming Languages and Systems. Springer Berlin Heidelberg, 2007. 189-204.

[9] Nanevski, Aleksandar, Greg Morrisett, and Lars Birkedal. ”Hoare type theory, polymorphism and separation.” Journal of Functional Programming 18.5-6 (2008): 865-911.

[10] Nanevski, Aleksandar, Greg Morrisett, and Lars Birkedal. ”Polymorphism and separation in hoare type theory.” ACM SIGPLAN Notices. Vol. 41. No. 9. ACM, 2006.

[11] Petersen, Rasmus Lerchedahl, et al. ”A realizability model for impredicative Hoare type theory.” Programming Languages and Systems. Springer Berlin Heidelberg, 2008. 337-352.

[12] Svendsen, Kasper, Lars Birkedal, and Aleksandar Nanevski. ”Partiality, state and dependent types.” Typed Lambda Calculi and Applications. Springer Berlin Heidelberg, 2011. 198-212.

[13] Vákár, Matthijs. "A Categorical Semantics for Linear Logical Frameworks." arXiv preprint arXiv:1501.05016 (2015).

[14] Wadler, Philip. "The marriage of effects and monads." ACM SIGPLAN Notices. Vol. 34. No. 1. ACM, 1998.

[15] https://www.cs.cornell.edu/Courses/cs4110/2012fa/lectures/lecture11.pdf