Semantic Analysis of Advanced Programming Languages

Harley Eades III
Computer Science
The University of Iowa

1 Thesis: Introduction

There are two major problems growing in two areas. The first is in Computer Science, in particular software engineering. Software is becoming more and more complex, and hence more susceptible to software defects. Software bugs have two critical repercussions: they cost companies lots of money and time to fix, and they have the potential to cause harm.

The National Institute of Standards and Technology estimated that software errors cost the United State's economy approximately sixty billion dollars annually, while the Federal Bureau of Investigations estimated in a 2005 report that software bugs cost U.S. companies approximately sixty-seven billion a year [10, 15].

Software bugs have the potential to cause harm. In 2010 there were a approximately a hundred reports made to the National Highway Traffic Safety Administration of potential problems with the braking system of the 2010 Toyota Prius [1]. The problem was that the anti-lock braking system would experience a "short delay" when the brakes where pressed by the driver of the vehicle [14]. This actually caused some crashes. Toyota found that this short delay was the result of a software bug, and was able to repair the the vehicles using a software update [11]. Another incident where substantial harm was caused was in 2002 where two planes collided over Überlingen in Germany. A cargo plane operated by DHL collided with a passenger flight holding fifty-one passengers. Air-traffic control did not notice the intersecting traffic until less than a minute before the collision occurred. Furthermore, the on-board collision detection system did not alert the pilots until seconds before the collision. It was officially ruled by the German Federal Bureau of Aircraft Accidents Investigation that the on-board collision detection was indeed faulty [8].

The second major problem affects all of science. Scientific publications are riddled with errors. A portion of these errors are mathematical. In 2012 Casey Klein et al. used specialized computer software to verify the correctness of nine papers published in the proceedings of the International Conference on Functional Programming (ICFP). Two of the papers where used as a control which where known to have been formally verified before. In their paper [6] they show that all nine papers contained mathematical errors. This is disconcerting especially since most researchers trust published work and base their own work off of these papers. Kline's work shows that trusting published work might result in wasted time for the researchers basing their work off of these error prone publications. Faulty research hinders scientific progress.

Both problems outlined above have been the focus of a large body of research over the course of the last forty years. These challenges have yet to be completed successfully. The work I present here makes up the foundations of one side of the programs leading the initiative to build theory and tools which can be used to verify the correctness of software and mathematics. This program is called program verification using dependent type theories. The second program is automated theorem proving. In this program researchers build tools called model checkers and satisfiability modulo-theories solvers. These tools can be used to model and prove properties of large complex systems carrying out proofs of the satisfiability of certain constraints on the system nearly automatically, and in some cases fully automatically. As an example André Platzer and Edmund Clarke in 2009 used automated theorem proving to verify the correctness of the in flight collision detection systems used in airplanes. They actually found that there were cases where two plans could collide, and gave a way to fix the problem resulting in a fully verified algorithm for collision detection. That is he mathematically proved that there is no possible way for two plans to collide if the systems are

operational [9]. Automated theorem provers, however, are tools used to verify the correctness of software externally to the programming language and compiler one uses to write the software. In contrast with verification using dependent types we wish to include the ability to verify software within the programming language being used to write the software. Both programs have their merits and are very fruitful and interesting.

This report summarizes my dissertation by part, chapter, section, and subsection. Each section will be given the name of a part, and then the contents of the section will consist of a summery of that part. Similarly, sections will summarize chapters, and so on. I make sure to include my already published work as well as on going work that needs to be done before my defense. My thesis will be broken into five main parts. The first, gives a history of type theory, and the necessary background to facilitate understanding of the main results. The second part is on the design of new advanced functional programming languages. The third covers the basic syntactic-analysis of various type theories and functional programming languages. The forth covers normalization by hereditary substitution. Finally, the fifth covers categorical semantics of type theories.

2 Part 0: History and Background

This part provides a brief history of type theory as a foundation of mathematics and typed-functional programming languages. It begins with Bertrand Russell – the founder of type theory – and introduces key results up to the present. This part also serves as an introduction of all the necessary concepts to understand the remainder of the thesis. I make sure and present each type theory in its entirety and rigorously. In fact every language defined in the thesis will be formally defined in Ott [12]. Ott is a tool for writing definitions of programming languages, type theories, and λ-calculi. Ott generates a parser and a type checker which is used to check the accuracy of all objects definable with in the language given to Ott as input. Ott's strongest application is to check for syntax errors within research articles. Ott is a great example of a tool using the very theory I will present in my thesis. It clearly stands as a successful step towards the solution of the second major problem outlined in the introduction. Lastly, this history and background has all been written and was presented as my comprehensive exam.

3 Part I: Design

This part presents the design of a two general-purpose dependently-typed functional programming languages called Freedom of Speech, and Separation of Proof from Program, and a new constructive type theory with constructive control called Dualized Type Theory. This part will have a chapter per language. All the work with respect to the first two languages is complete although mostly unpublished. The language of Dualized Type Theory is stabilizing, but its analysis is on going work [13].

The TRELLYS project is a collaboration between the University of Iowa, University of Pennsylvania, and Portland State University to design a new general-purpose dependently-typed functional programming language that supports type-based verification. What sets TRELLYS apart from other similar projects is that it contains a number of advanced features within the same language. For example, mixing type-based verification with general recursion is not well understood. This mixture is one of the primary aims of this project. TRELLYS was designed from the bottom up. We started with a small trusted language called the core, and then, after the core was finished, we would build a surface language on top of it. Then the latter would elaborate into the former. My primary contribution to this project was to the design and analysis of the core. I helped with the design and analysis of two core languages. The first is Freedom of Speech.

3.1 Chapter I: Freedom of Speech

The main objective of the TRELLYS project is to design a functional programming language with two discernible fragments: a logical fragment, and a programmatic fragment. The programmatic fragment is a general purpose dependently-type functional programming language. This is a Turing-complete language with general recursion. In addition it contains the type in type axiom which

leads to paradoxes [2, 3]. This axiom allows for the definition of generic programs. Now the logical fragment is the fragment that can be used to prove properties about the programs defined in the programmatic fragment. In order for this fragment to be considered a logic it must terminating. Meaning every program written in the this fragment must terminate in general. This property guarantees that the logic is consistent.

Recall that the three perspectives of computation[1] tell us that programs are proofs and their types are propositions. The most significant feature of Freedom of Speech is that logical types can contain programs from the programmatic fragment, but they are never allowed to be applied to any arguments. Thus, the logical fragment is allowed to "talk" about the programmatic fragment. This property we call free speech.

This chapter introduces the Freedom of Speech language and discusses its design. The analysis of this language greatly influenced its design. I discuss the analysis in Section 4.1. Interesting properties of this design include: implicit arguments, the free speech property, a collapsed syntax, and judgmental fragmentation into the logical and programmatic fragment. This design is complete and formalized in Ott, but was never published.

3.2 Chapter II: Separation of Proof from Program (SepPP)

One of the key features of the Freedom of Speech language is that it has a collapsed language where terms and types are apart of the same syntactic category. This makes for a very elegant design, but due to having to fragment the language into a logical fragment and a programmatic fragment using the typing judgment the language became very hard to reason about. This is especially true when we want to extend the Freedom of Speech language with additional features. For example, algebraic data types. To overcome this problem we (the University of Iowa group) proposed to separate the logical fragment from the programmatic fragment into two distinct languages. Then provide the necessary linkage to allow for the logical fragment to contain the free-speech property.

In this section the full design of SepPP is detailed and discussed. We did not carry out any analysis of this language, but only explored its design, and its use in writing real-world programs [5, 4]. The full design of SepPP has never been published, however, its design is completely written up, and an implementation[2] exists.

3.3 Chapter III: Dualized Type Theory (DTT)

The Freedom of Speech language did not contain algebraic data types. SepPP added data types, but separated the two worlds all together. In addition neither language contained coinductive data types. These are data types representing infinite objects. For example, streams. So how could we add coinductive data types? To make matters a bit more interesting, how could we add coinductive data types in such a way that inductive-coinductive data types could be defined? That is, can we allow for the mixture of inductive and coinductive data? This is not a well understood feature. In fact, a general framework for inductive and coinductive data is an open problem.

A candidate for a general framework for inductive and coinductive data types begins with a logic called Bi-Intuitionistic Logic (BINT). This is a constructive logic where for every logical connective its dual connective is a logical connective of the logic. For example, BINT contains disjunction and conjunction, and implication, and its dual a connective called subtraction or exclusion. In this chapter I detail the logic of BINT called Dualized Logic (DIL), and a term assignment to DIL called Dualized Type Theory (DTT). I will also discuss its potential as a general framework for inductive and coinductive data. I also think DTT provides a nice setting for the study of constructive control operators. Constructive control operators are control operators – operators that have the ability to discard evaluation contexts – that have been restricted to discarding only certain context so as to remain constructive. The design of DIL and DTT is stabilizing, but its analysis is on going work [13].

[1] Also known as the Curry-Howard correspondence, and the propositions-as-types proofs-as-programs correspondence.

[2] SepPP's implementation was authored by Dr. Garrin Kimmel.

4 Part II: Basic-Syntactic Analysis

There is a cardinal rule one must follow when designing programming languages. This rules states that one must carry out at least a basic-syntactic analysis of the programming language one is designing. A basic-syntactic analysis includes proving that the programming language is type safe. If the language contains a logical fragment then this fragment must be proven consistent. This part presents this analysis for the Free Speech language and DTT.

4.1 Chapter I: Free Speech

This chapter presents a preservation proof of the Free Speech language. Then using this proof I prove weak normalization for the logical fragment. These results do not use any new mathematical machinery, but adapts well-known techniques to the language I am investigating. There are however a few non-trivial problems which had to be overcome which this chapter will take care in explaining. This analysis is complete and ready to be included in my thesis, however it is unpublished.

4.2 Chapter II: Dualized Type Theory

The analysis of DTT begins with the analysis of DIL. We prove that DIL is consistent, and complete with respect to a semantics for BINT using Kripke models. The consistency proof is complete and has been formalized in the interactive proof assistant Agda [7]. I am currently finishing up the completeness proof, but on paper.

Now that we know we have a sound and complete bi-intuitionistic logic we add a term assignment to DIL. This takes the logic and produces a type theory where proofs are programs and propositions are types. In addition to this term assignment we must define a reduction relation which gives the necessary rules for computaiton. That is, the reduction relation relates programs by taking a single step of computaiton. Then we must prove this reduction relation is type safe, and that it is terminating, because DTT is a logic. We want to use it for logical reasoning, and thus, it cannot contain infinite proofs. I conjecture these properties hold, but they still need to be carried out. This analysis should be strightforward using known techniques. I plan to have this work completed by Febuary 2014 for publication.

5 Part III: Normalization by Hereditary Substitution

5.1 Chapter I: Stratified System F (SSF) and its Extensions

5.2 Chapter II: The $\lambda\Delta$-Calculus

6 Part IV: Categorical Semantic

6.0.1 Chapter I: Semi-Bilinear Logic

6.0.2 Chapter II: Split Bi-Intuitionistic Logic

6.0.3 Chapter III: Dualized Type Theory

6.0.4 Chapter IV: Nested Bi-Intuitionistic Logic

7 Conclusion

References

[1] blogs.consumerreports.org. Consumer reports cars blog: Japan investigates reports of prius brake problem, 2010.

[2] T. Coquand. An analysis of girard's paradox. In *In Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[3] T. Coquand. A new paradox in type theory. In *Logic, Methodology and Philosophy of Science IX : Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.

[4] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjoberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. *Programming Languages Meets Program Verifcation (PLPV)*, 2012.

[5] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjoberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. *Special issue of Progress in Informatics*, March 2013.

[6] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

[7] U. Norell. Towards a practical programming language based on dependent type theory. PhD Thesis, 2007.

[8] The German Federal Bureau of Aircraft Accidents. Investigation report, 2004.

[9] A. Platzer and C. Edmund. Formal verification of curved flight collision avoidance maneuvers: A case study. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[10] Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002. Sponsored by the Department of Commerce's National Institute of Standards and Technology.

[11] Reuters. Toyota to recall 436,00 hybrids globally-document, February 2010.

[12] P. Sewell, F. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. StrniŠa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122.

[13] Aaron Stump, Harley Eades, and Ryan McCleary. Extended abstract: Reconsidering intuitionistic duality. Control Operators and their Semantics 2013, June 2013.

[14] thedetroitbureau.com. Nhtsa memo on regenerative braking, April 2011.

[15] United States Federal Bureau of Investigation. 2005 FBI Computer Crime Survey.