Dissertation Prospectus A Full-Spectrum Dependently typed language for testing with dynamic equality

Mark Lemay

January 14, 2021

Abstract

Dependent type systems offer a powerful tool to eliminate bugs from programs. Interest in dependent types is often driven by the inherent usability of such systems: Dependent types systems can re-use that methodology and syntax that functional programmers are familiar with for formal proofs. This insight has lead to several Full-Spectrum languages that try and present programmers with a consistent and unrestricted view of proofs and programs. However these languages still have substantial usability issues: missing features like general recursion, confusingly conservative equality, an inability to prototype, and no strait forward way to test specifications that have not yet been proven.

I attempt to solve these problems by building a new language that contains standard functional programming features such as general recursion, with a gradualized equality, runtime proof search and a testing system.

Contents

1 Introduction

The promise of dependent types in a practical programming language has been the goal of research projects for decades. There have been many formalization and prototypes that make different compromises in the design space. One popular direction to explore is "Full-spectrum" dependently types languages, these languages tend to have a minimalist approach: computation can appear anywhere in a term or type. Such a design purposely exposes the Curry-Howard correspondence, as opposed to trying to hide it as a technical foundation: a proof has the exact same syntax and behavior as a program. This direct approach tries to make clear to the programmer the subtleties of the proof system that are often obscured by other formal method systems. Even though this style makes writing efficient programs hard, and drastically complicates the ability to encode effects, it can be seen in some of the most popular dependently typed languages (notably Agda and Idris).

However there are several inconveniences with languages in this style:

1. A restriction on standard programming features, such as general recursion
2. A subtle and weak notion of equality
3. difficulties in prototyping proofs and programs
4. Difficulties in testing programs that make use of dependent types

While each problem will be treated as separately as possible, the nature of dependent types requires that equality is modified before testing and prototyping can be handled. The notion of equality itself is also very sensitive to which programmatic features are included. My thesis will solve these problems by

- Defining a full-spectrum dependently typed base language, with a few of the most essential programming features like general recursion and user defined data types
- A cast language that supports dynamic equality checking

- Syntax that supports runtime proof search
- A symbolic testing system that will exercise terms with uncertain equalities and runtime proof search

2 A Dependently Typed Base Language

The base language contains the features:

- Unrestricted dependent data types (no requirement of strict positivity)
- Unrestricted recursion (no required termination checking)
- Type-in-type (no predictive hierarchy of universes)

Any one of these features can result in logical unsoundness[1], but they are widely used in mainstream functional programming. In spite of the logical unsoundness, the resulting language is still has type soundness[2]. Type checking is undecidable for this language, however this has not been a problem in practice[3].

The type theory is intutionistic, definitional equality is the $\alpha\beta$ equivalence of terms. The implementation is written in a bidirectional style allowing some annotations to be inferred.

Though this language is non-terminating it supports a partial correctness property for first order data types when run with CBV, for instance:

$$\vdash M : \sum x : \mathbb{N}.\mathtt{IsEven}\, x$$

$\mathtt{fst}\, M$ may not terminate, but if it does, $\mathtt{fst}\, M$ will be an even $\mathbb{N}$. However, this property does not extend to functions

$$\vdash M : \sum x : \mathbb{N}.\, (y : \mathbb{N}) \to x \leq y$$

it is possible that $\mathtt{fst}\, M \equiv 7$ if

$$\langle 7, \lambda y.\mathtt{loopForever} \rangle$$

[1] Every type is inhabited by an infinite loop

[2] no term with a reduct that applies an argument to a non-function in the empty context will type

[3] While languages like Coq and Agda claim decidable typechecking, it is easy to construct terms who's type verification would exceed the computational resources of the universe

The hope would be that the type is sufficient to communicate intent, in the same way unproductive non-termination is trappable in the vast majority of typed languages but still considered a bug.

2.1 Prior work for the Base Language

While the base language is in line with prior research, I am unaware of any development with exactly these features. Agda supports general recursion and Typy-in-type with compiler flags, and can some non-positive data types using coinduction. Idris supports similar "unsafe" features. Meta-theoretically, this base language is similar to [23] though data and equality are formulated differently. The base language has been deeply informed by the Trellys Project[12][23][3, 2] [24] [22] and the Zombie Language[4] it produced.

[11] claims a similar "partial correctness" criterion.

3 A Language with Dynamic Dependent Equality

A key issue with full-spectrum dependent type theories is the characterization of definitional equality. Since computation can appear at the type level, and types must be checked for equality, traditional dependent type theories pick a subset of equivalences to support. For instance, the base language follows the common choice of $\alpha\beta$ equivalence of terms. However this causes many obvious programs to not type-check:

$$\begin{aligned}
\texttt{Vec} &: \mathbb{N} \to * \to * \\
\texttt{rep} &: (x : \mathbb{N}) \to \texttt{Vec}\, x\, \mathbb{B} \\
\texttt{head} &: (x : \mathbb{N}) \to \texttt{Vec}\ (1 + x)\ \mathbb{B} \to \mathbb{B} \\
&\nvdash \lambda x.\texttt{head}\, x\ (\texttt{rep}\ (x + 1))\ :\ \mathbb{N} \to \mathbb{B}
\end{aligned}$$

Since $1 + x$ does not have the same definition as $x + 1$.

Overly fine definitional equalities directly results in the poor error messages that are common for dependently typed languages [7]. For instance, the above will give the error message "x + 1 != suc x of type ℕ when checking that the expression rep (x + 1) has type Vec Bool (1 + x)" in Agda. The error is confusing since it objects to an obvious property of addition, and if addition were buggy no hints would be given to fix the problem. Ideally the error messages would give a specific instance of x where $x + 1 \neq 1 + x$ or

[4] https://github.com/sweirich/trellys

remain silent. There is some evidence that specific examples can help clarify the error messages in OCaml[20] and there has been an effort to make refinement type error messages more concrete in Liquid Haskell[10].

Strengthening the equality relation dependently typed languages is used to motivate many research projects (to name a few [4, 24, 19]). However, every formulation I am aware of intends to preserve decidable type checking or logical soundness, so equality will never be complete[5]. Since I intend to dispense with both decidable type checking or logical soundness I can propose a language with an equality that is more convenient in practice.

Building off the base language I purpose a dynamic cast language, a cast type system and a partial elaboration function that satisfies the following basic guarantees:

1. $\vdash e : M$ and $elab\,(e, M) = e'$ then $\vdash e' : M'$ for some $\vdash M' : *$.
2. $\vdash e' : M'$ and $e' \downarrow blame$ then there is no $\vdash e : M$ such that $elab\,(e, M) = e'$
3. $\vdash e' : *$ and $elab\,(e, *) = e'$ then
   (a) if $e' \downarrow *$ then $e \downarrow *$
   (b) if $e' \downarrow (x : M') \to N'$ then $e \downarrow (x : M) \to N$
   (c) if $e' \downarrow TCon\triangle'$ then $e \downarrow TCon\triangle$
4. $\vdash e' : M'$ then
   (a) $e' \downarrow v'$ and $\vdash v' : M'$
   (b) or $e' \uparrow$
   (c) or $e' \downarrow blame$

In the example above $\lambda x.\mathtt{head}\,x\ (\mathtt{rep}\ (x+1))\ :\ \mathbb{N} \to \mathbb{B}$ will not emit any errors at compile time or runtime (though a warning may be given). If the example is changed to

$$\lambda x.\mathtt{head}\,x\ (\mathtt{rep}\,x)\ :\ \mathbb{N} \to \mathbb{B}$$

no static error will be given, but if the function is called a runtime error will be throughn. The blame tracking system will blame the exact static location that uses unequal types with a direct proof of inequality, allowing an error like "failed at application $\left(\mathtt{head}\,x : \mathtt{Vec}\,\underline{(1+x)}\,\mathbb{B} \to ...\right)(rep\,x : \mathtt{Vec}\,\underline{x}\,\mathbb{B})$

[5]I am also unaware of any suitable notion of complete equality though it is considered in [22]

since when $x = 3$, $1 + x = 4 \neq 3 = x$", regardless of where in the program the the function was called.

3.1 Prior work

It is unsurprising that dynamic quality is shares many of the same concerns as the large amount of work contracts, hybrid types, gradual types, and blame. In fact, this work could be seen as gradualizing the Reflection Rule in Extensional Type Theory.

Blame has been strongly advocated for in [26, 25]. Blame tracking can establish the reasonableness of gradual typing systems, though as many authors have noticed, proving blame correctness is tedious and error prone, many authors only conjecture it for their systems.

The basic correctness conditions are inspired by the Gradual Guarantee [21]. The implementation also takes inspiration from "Abstracting gradual typing"[9], where static evidence annotations become runtime checks. Unlike some impressive attempts to gradualize the polymorphic lambda calculus [1], dynamic equality does not attempt to preserve any parametric properties of the base language.

A direct attempt has been made to gradualize a full spectrum dependently typed language to an untyped lambda calculus using the AGT philosophy in [8]. However that system retains the definitional style of equality and user defined data types are not supported. The paper is largely concerned with establishing decidable type checking via an approximate term normalization.

A refinement type system with higher order features is gradualized in [28] though it does not appear powerful enough to be characterized a a full-spectrum dependent type theory. [28] builds on earlier refinement type system work, which described itself as "dynamic" . A notable example is [18] which describes a refinement system that limit's predicates to base types.

4 Prototyping proofs and programs

Just as "obvious" equalities are missing from the definitional relation, "obvious" proofs and programs are not always conveniently available to the programmer. For instance, in Agda it is possible to write a sorting sorting function quickly using simple types. With expertise and effort is it possible to prove that sorting procedure correct by rewriting it with the necessarily invariants. However very little is offered in between. The problem is magnified if module boundaries hide the implementation details of a function, since the details are exactly what is needed to make a proof! This is especially important for larger scale software where a library may expect proof terms that while "correct" are not constructible from the exports of the other library.

The solution proposed here is some additional syntax that will search for a term of the type when resolved at runtime. Given the sorting function

$$\texttt{sort} : \texttt{List}\,\mathbb{N} \to \texttt{List}\,\mathbb{N}$$

and given the first order predicate that

$$\texttt{IsSorted} : \texttt{List}\,\mathbb{N} \to *$$

then it is possible to assert that `sort` behaves as expected with

$$\lambda x.? : (x : \texttt{List}\,\mathbb{N}) \to \texttt{IsSorted}\,(\texttt{sort} x)$$

this term will act like any other term at runtime, given a list input it will verify that the `sort` function correctly handles that input, give an error, or non-terminate.

Additionally this would allow simple prototyping form first order specification. For instance,

$$\begin{aligned}
&\mathit{data}\,\texttt{Mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to * \,\mathit{where} \\
&\qquad \texttt{base} : (x : \mathbb{N}) \to \texttt{Mult} 0\, x\, 0 \\
&\qquad \texttt{suc} : (x\, y\, z : \mathbb{N}) \to \texttt{Mult}\, x\, y\, z \to \texttt{Mult}\ (1 + x)\ y\, (y + z)
\end{aligned}$$

can be used to prototype

$$\texttt{div} = \lambda x.\lambda y.\texttt{fst}\left(? : \sum z : \mathbb{N}.\texttt{Mult} x\, y\, z\right)$$

The term search is made subtly easier by the the dynamic equality, otherwise examples like

$$? : \sum f : \mathbb{N} \to \mathbb{N}.\texttt{Id}\,(f, \lambda x.x + 1)\,\&\texttt{Id}\,(f, \lambda x.1 + x)$$

which would require resolving definitional behavior. Using dynamic equality it is possible only consider the extensional behavior of functions.

Though the proof search is currently primitive, better search methods could be incorporated in future work.

4.1 Prior work

Proof search is often used for static term generation in dependently typed languages (for instance Coq tactics). A first order theorem prover is attached to Agda in [17].

Twelf made use of runtime proof search but the underling theory cannot be considered full spectrum.

5 Testing dependent programs

Both dynamic equalities and dynamic proof search vastly weaken the guarantees of normal dependent type systems. Programmers still would like a evidence of correctness, even while they intend to provide full proofs of properties in the future. However, there are few options available in full spectrum dependently typed languages aside from costly and sometimes unconstructable proofs.

The mainstream software industry has similar needs for evidence of correctness, and has made use of testing done in a separate execution phase. Given the rich and precise specifications that dependent types provide it is possible to improve on the hand crafted tests used by most of the industry. Instead we can use a type directed symbolic execution, to run questionable equalities over concrete values and engage and precompute the searched proof terms. Precomputed proof terms can be cached, so that exploration is not too inefficient in the common case of repeating tests at regular intervals of code that is mostly the same. Precomputed terms can be made available at runtime, covering for the inefficient search procedure.

Finally future work can add more advanced methods of testing and proof generation. This architecture should make it easier to add more advanced exploration and search without changing the underlining definitional behavior.

5.1 Prior work

5.1.1 Symbolic Execution

Most research for Symbolic Execution targets popular languages (like C) and uses SMT solvers to efficiently explore branches that depend on base types. Most work does not support higher order functions or makes simplifying assumptions about the type system. There are however some relevant papers:

- [10] presents a symbolic execution engine supporting Haskell's lazy execution and type system. Higher order functions are not handled
- The draft work[27], handles higher order functions as and inputs provides a proof of completeness
- Symbolic execution for higher order functions for a limited untyped variant of PCF is described in [16]

5.1.2 Testing dependent types

There has been a long recognized need for testing in addition to proving in dependent type systems

- In [6] a QuickCheck style framework was added to an earlier version of Agda
- QuickChick[6] [5][15, 14, 13] is a research project to add testing to Coq. However testing requires building types classes that establish the properties needed by the testing framework such as decidable equality. This is presumably out of reach of novice Coq users.

References

[1] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[2] Chris Casinghino. Combining proofs and programs. 2014.

[3] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[4] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: A type theory with computational assumptions. *Proceedings of the ACM on Programming Languages*, 2021.

[5] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[6] https://github.com/QuickChick/QuickChick

[6] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 188–203. Springer, 2003.

[7] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. A framework for improving error messages in dependently-typed languages. *Open Computer Science*, 9(1):1–32, 2019.

[8] Joseph Eremondi, Éric Tanter, and Ronald Garcia. Approximate normalization for gradual dependent types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[9] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. Association for Computing Machinery.

[10] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 411–424, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 275–286, 2010.

[12] Garrin Kimmell, Aaron Stump, Harley D Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 15–26, 2012.

[13] Leonidas Lampropoulos. Random testing for language design. 2018.

[14] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129, 2017.

[15] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[16] Phuc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming*, 27, 2017.

[17] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[18] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US.

[19] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[20] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 228–242, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] Vilhelm Sjöberg. A dependently typed language with nontermination. 2015.

[23] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. *Mathematically Structured Functional Programming*, 76:112–162, 2012.

[24] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 369–382, 2015.

[25] Philip Wadler. A Complement to Blame. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 309–320, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[26] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 1–16, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[27] Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. Dynamic symbolic execution of higher-order functions, 2020.

[28] Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. Î»db: Blame tracking at higher fidelity. 2020. First ACM SIGPLAN Workshop on Gradual Typing 2020, WGT 2020 ; Conference date: 19-01-2020 Through 25-01-2020.