Draft

October 27, 2021

Abstract

This theses follows 2 convictions that have been under explored in the type theory literature:

- It should be easier to write programs with dependent types then without
- Whenever possible, static errors should be replaced with
  - static warnings,
  - more concrete and clear runtime errors,
  - and testing strategies that uncover concrete runtime errors

Part I
Introduction

Programming is an error-filled process. While different formal methods approaches can make some error rare or impossible, they burden programers with complex additional syntax and semantics that can make them hard to work with. Dependent type systems offer a simpler approach. In a dependent type system, proofs and invariants can borrow from the syntax and semantics already familiar to functional programmers.

This promise of dependent types in a practical programming language has inspired research projects for decades. Several approaches have now been explored. The **full-spectrum** approach is a popular and parsimonious approach that allow computation to behave the same at the term and type level [Aug98, Nor07, Bra13, SCA+12]. While this approach offers tradeoffs, it seems to be the most predictable from the programmer's perspective.

For instance, dependent types can prevent an out-of-bounds error when indexing into a length indexed list. The following type checks in virtually all full-spectrum dependent type systems

$$
\begin{aligned}
&\texttt{Bool} : *,\\
&\ \texttt{Nat} : *,\\
&\ \texttt{Vec} : * \to \texttt{Nat} \to *,\\
&\ \texttt{add} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat},\\
&\ \texttt{rep} : (A : *) \to A \to (x : \texttt{Nat}) \to \texttt{Vec}\, A\, x,\\
&\texttt{head} : (A : *) \to (x : \texttt{Nat}) \to \texttt{Vec}\, A\, (\texttt{add}\, 1\, x) \to A
\end{aligned}
$$

$$
\vdash \lambda x.\texttt{head}\, \texttt{Bool}\, x\ (\texttt{rep}\, \texttt{Bool}\, \texttt{true}\ (\texttt{add}\, 1\, x)) : \texttt{Nat} \to \texttt{Bool}
$$

We are sure `head` never inspects an empty list because the `rep` function will always return a list of length $1 + x$. In a more polished implementation many arguments would be implicit and the above could be written as $\lambda x.\texttt{head}\ (\texttt{rep}\, \texttt{true}\ (1 + x)) : \texttt{Nat} \to \texttt{Bool}$.

Unfortunately, dependent types have yet to see widespread industrial use. Programmers often find dependent type systems difficult to learn and use. One of the reasons for this difficulty is that conservative assumptions about equality create subtle issues for users, and lead to some of the confusing error messages these languages are known to produce [ESH19].

The following will not type check in any conventional system with user defined addition,

$$
\not\vdash \lambda x.\texttt{head}\, \texttt{Bool}\, x\ (\texttt{rep}\, \texttt{Bool}\, \texttt{true}\ (\texttt{add}\, x\, 1)) : \texttt{Nat} \to \texttt{Bool}
$$

Obviously $1 + x = x + 1$. However in the majority of dependently typed programming languages, $\texttt{add}\, 1\, x \equiv \texttt{add}\, x\, 1$ is not a definitional equality. This means a term of type $\texttt{Vec}\, (\texttt{add}\, 1\, x)$ cannot be used where a term of type $\texttt{Vec}\, (\texttt{add}\, x\, 1)$ is expected. Usually when dependent type systems encounter situations like this, they will give a type error and prevent evaluation. If the programmer made a mistake in the definition of addition such that $\texttt{add}\, 1\, x \not\equiv \texttt{add}\, x\, 1$, no hints are given to correct the mistake. This increase of friction and lack of communication are key reasons that dependent types systems are not more widely used.

Instead why not sidestep static equality? We could assume the equalities hold and discover a concrete witness of inequality as a runtime error. Assuming there was a mistake in the implementation of `add`, we could instead provide a runtime error that gives an exact counter example. For instance, if the `add` function incorrectly computes $\texttt{add}\, 8\, 1 = 0$ the above function will "get stuck" on the input 8. If that application is encountered at runtime we can give the error $\texttt{add}\, 1\, 8 = 9 \not= 0 = \texttt{add}\, 8\, 1$.There is some evidence that specific examples like this can help clarify the type error messages in OCaml [SJW16] and there has been an effort to make refinement type error messages more concrete in other systems like Liquid Haskell [HXB+19].

Runtime type checking leads to a different workflow than traditional type systems. Instead of type checking first and only then executing the program,

execution and type checking can both inform the programmer. Users can still be warned about uncertain equalities, but the warning need not block the flow of programming. Since the user can gradually correct their program as errors surface, we call this workflow **gradual correctness**.

Additionally, our approach avoids fundamental issues of definitional equality. No system will be able to statically verify every "obvious" equality for arbitrary user defined data types and functions, since general program equivalence is famously undecidable. By weakening the assumption that all equalities be decided statically, we can experiment with other advanced features without arbitrarily committing to which equalities are acceptable. Finally, we expect this approach to equality is a prerequisite for other desirable features such as a foreign function interface, runtime proof search, and a lightweight ability to test dependent type specifications.

Though gradual correctness is an apparently simple idea, there are several subtle issues that must be dealt with. While it is easy to check ground natural numbers for equality, even simply typed functions have undecidable equality. This means that we cannot just check types for equality at applications of higher order functions. Dependent functions mean that equality checks may propagate into the type level. Simply removing all type annotations will mean there is not enough information to construct good error messages. We are unaware of research that directly handles all of these concerns.

We solve these problems with a system of 2 dependently typed languages connected by an elaboration procedure.

- The surface language, a conventional full-spectrum dependently typed language (section 2)
  - the untyped syntax is used directly by the programer
  - the type theory is introduced to make formal comparisons
- The cast language, a dependently typed language with embedded runtime checks (section 3)
  - will actually be run
  - intended to be invisible to the programer
- An elaboration procedure that transforms untyped surface syntax into checked cast language terms (section 4)

The programmer uses the untyped syntax of the surface language to write programs that they intend to typecheck in the conventional dependently typed surface language. Programs that fail to typecheck under the conservative type theory of the surface language, are elaborated into the cast language. These cast language terms act exactly as typed surface language terms would, unless the programmer assumed an incorrect equality. If an incorrect equality is encountered, a clear runtime error message is presented against the static location of the error, with a counter example.

Part II
Surface language

see chapter draft

Part III
Dynamic and Elab

TODO this will be symmetric with Part 2

TODO discussion observaiontal equality, more correct or more explicit

TODO unusual notion of errors, as far as CBV is concerned!

Part IV
Data and Pattern matching

there are several simpler systems that can be worked through: eliminator style patterns, cast patterns, but to bring it all together we need congruence over functions.

adding paths and path variables means that constructs can still fail at runtime, but they can blame the actually problematic components

validating the K axiom, not that edqualities are unique, merely that we don't care which one of the unique equalities is used.

Other extensions to the Calculus of Constructions that are primarily concerned with data (UCC, CIC) will be reviewed in chapter 4.

Coq and Lean trace their core theory back to the Calculus of Constructions.

Part V
Automated testing

Part VI
Runtime proof search

Part VII
Conclusion And future work

Chapter 2 argueged that decidability was a poor metric for type systems, A more meaningful metric, perhaps based on localizing problems would be good

Part VIII
Scratch

The surface language is pure in the sense of Haskell, supporting only non-termination and unchecked errors as effects. Combining other effects with full-spectrum dependent types is substantially more difficult because effectful equality is hard to characterize for individual effects and especially hard for effects in combination. Several attempts have been made to combine dependent types with more effects [PT20] [Ahm17b, Ahm17a][PT20] but there is still a lot of work to be done. General effects, though undoubtedly useful, will not be considered in this proposal.

...

In spite of logical unsoundness, the surface language still supports a partial correctness property for first order data types when run with Call-by-Value. For instance,

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

$\texttt{fst}\, M$ may not terminate, but if it does, $\texttt{fst}\, M$ 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 $\texttt{fst}\, M \equiv 7$ if

$$M \equiv \langle 7, \lambda y.\texttt{loopForever} \rangle$$

References

[Ahm17a] Danel Ahman. Fibred computational effects. *arXiv preprint arXiv:1710.02594*, 2017.

[Ahm17b] Danel Ahman. Handling fibred algebraic effects. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[Aug98] Lennart Augustsson. Cayenne a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. Association for Computing Machinery.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[ESH19] 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.

[HXB+19] 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.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[PT20] Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle how to mix substitution, dependent elimination, and effects. 2020.

[SCA+12] 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.

[SJW16] 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.