Chapter 1 (draft): Introduction

Mark Lemay

December 31, 2021

Part I

Introduction

Writing correct programs is difficult. While many formal methods approaches make some errors rare or impossible, they often require programmers learn additional syntax and semantics. Dependent type systems can offer a simpler approach. In dependent type systems, proofs and properties use the same language and meaning already familiar to functional programmers.

While the type systems of mainstream programing languages allows tracking simple properties, like `7 : int`. Dependent types allow complicated properties to be assumed and verified, such as a provably correct sorting function

$$sort\ :\ (input : List\,\mathbb{N}) \to \Sigma ls : List\,\mathbb{N}.IsSorted\,input\,ls$$

by providing an appropriate term of that type. From the programmer's perspective, the function arrow and the implication arrow are the same. The proof $IsSorted$ is no different then any other term of a datatype like $List$ or $\mathbb{N}$.

The power of dependent types has been recognized for decades. Dependent types form the back bone of several poof system, such as Coq[CDT12], Lean[MU21], and Agda[Nor07] . They have have been proposed as a foundation for mathematics[ML72, Pro13]. Dependent types are directly used in several programming languages such as ATS[Xi07] and Idris[Bra13], while influencing many other programing languages such as Haskell and Scala.

not clear best citat seems to sensus onl to cite sof

cite the la dependent neither?

Unfortunately, dependent types have not yet become mainstream in the software industry. Many of the usability issues with dependent types can trace their root to the the conservative nature of dependently typed equality. This thesis illustrates a new way to deal with equality constraints by delaying them until runtime.

A fragment of the system is proven correct according to a modified view of type soundness, and several of the proofs have been validated in Coq [1]. The system has been prototyped[2].

revise last paragraph

1 Example

For example, dependent type systems can prevent an index-out-of-bounds error when trying to read the first element a list. A version of the following type checks in virtually all dependent type systems:

expand this example? perhaps at the head of a constant vector first? "and this reasoning can be abstracted under functions"

[1]available at https://github.com/marklemay/dtest-coq, most work is due to Qiancheng Fu
[2]available at https://github.com/marklemay/dDynamic

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

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

make this a "code" example?

Where $\to$ is a function and $*$ means that the function results in a type. `Vec` is a list indexed by the type of element it contains and its length, it is a type that depends on its length. `rep` is a dependent function that produces a list containing a type with a given length, by repeating its input that number of times. `head` is a dependent function that expects a list of length $\mathtt{add}\,1\,x$, retuning the first element of that non-empty list.

There is no risk that `head` inspects an empty list. Luckily in the example the $\mathtt{rep}\,\mathtt{Bool}\,\mathtt{true}\,(\mathtt{add}\,1\,\mathtt{x})$ function will return a list of length $\mathtt{add}\,1\,x$, exactly the type that is required.

Unfortunately, programmers often find dependent type systems difficult to learn and use. This resistance has limited the ability of dependent types reach their full potential to help eliminate the bugs that pervade software systems. One of the deepest underling reasons for this frustration is the way dependent type systems handle equality.

For example, the following will not type check in any conventional dependent type system with user defined addition,

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

While "obviously" $1 + x = x + 1$, in the majority of dependently typed languages, $\mathtt{add}\,1\,x \equiv \mathtt{add}\,x\,1$ is not a "definitional" equality. "Definitional equality" is the name for the conservative approximation of equality used by dependent type systems for when two types are "obviously" the same. This prevents the use of a term of type $\mathtt{Vec}\,\mathtt{Bool}\,(\underline{\mathtt{add}\,1\,x})$ where a term of type $\mathtt{Vec}\,\mathtt{Bool}\,(\underline{\mathtt{add}\,x\,1})$ is expected. Usually when dependent type systems encounter situations like this, they will give an error message and block evaluation until the "mistake" is resolved.

In programming, types are used to avoid bad behavior, for instance they are often used to avoid "stuck" terms. If it is the case that $\mathtt{add}\,1\,x = \mathtt{add}\,x\,1$ the program will never get stuck. However, if there is a mistake in the implementation of `add`, the program might get stuck. For instance, if the `add` function incorrectly computes $\mathtt{add}\,8\,1 = 0$ the above function will "get stuck" on the input 8.

While the intent and properties of the `add` function are clear to programmers from its name and type, this information is unavailable to the type system. If the programmer made a mistake in the definition of addition, such that for some x, $\mathtt{add}\,1\,x \not\equiv \mathtt{add}\,x\,1$, the system will not provide hints on which x witnesses this inequality. Worse, the type system may even disallow experimenting with the `add` function until the "error" is removed.

nakes

Why block programmers when there is a type "error"?

There appears to be no reason! Alternatively, we can track unclear equalities and if the program "gets stuck", we are able to stop the program execution and provide a concrete witness for the inequality. If that application is encountered at runtime we can give a runtime error stating $\mathtt{add}\,1\,8 = 9 \neq 0 = \mathtt{add}\,8\,1$. Which is exactly the kind of specific feedback programers want when correcting code.

Part II
A Different Workflow

This thesis advocates an alternative usage of types. In most types systems a programmer can't run programs until the type system is convinced of their correctness [3]. Where this thesis argues "the programer is always right (until proven wrong)". This philosophy will likely go over better with programmers.

More concretely, whenever possible, static errors should be replaced with

[3] often requiring a graduate degree and uncommon patients

```
edit program
 |      ^
 |      | type errors
 v      |
 Type checks
 |
 | no type errors
 v
run program
```

better graphics

Figure 1: Standard Typed Programming Workflow

```
edit program
 |
 |
 v
 Elaborates                          ^
 |                |                  | runtime error
 | no warnings    | type warnings    |
 v                v                  |
    run program      ________________
```

better graphics

Figure 2: Workflow for this Thesis

- static warnings containing the same information,
- and more concrete and clear runtime errors that correspond to one of the warnings

Figure 1 illustrates the standard workflow from the perspective of programers in most typed languages. Figure 2 shows the workflow that is explored in this thesis.

These diagrams make it clear why there is so much pressure for type errors to be better in dependently typed programming[ESH19]. Type errors block programmers from running programs! However complaints about the type errors are probably better addressed by resolving mismatch between the expectations of the programmer and the design of the underling type theory. Better worded error messages are unlikely to bridge this gap when the type system doubts $x + 1 = 1 + x$.

The standard workflow seems sufficient for type systems in many mainstream typed programing languages. Though there is experimental evidence that even OCaml can be easier to learn and use with the proposed workflow [SJW16]. In the presence of dependent types the standard workflow is challenging for both beginners and experts, making a new approach much more critical.

What is new here? talk about some of the prior dependent type attempts, always a subset of syntax, or changes the syntax

By switching to the proposed workflow, type errors become type warnings, and the programer is free to run their program and experiment, while still presented with the all the information they would have gotten from a type error in the form of warnings. If there are no warnings, the programmer could call their program a proof along the lines of the Curry-Howard correspondence [4]. If there is value in a type error it comes from the message itself and not the hard stop it puts to programming. cite?

The proposed workflow is necessary, since often the type system is too conservative and the programmer is correct in implicitly asserting an equality that is not provable within the system. That the programmer may need to go outside the conservative bounds of definitional equality has been recognized since the earliest dependent type theories [ML72] and difficulties in dependently typed equality have motivated many research projects [Pro13,

[4]In the system presented here the programmer will need to manually verify termination

```
Ctx : * ;
Ctx = Var -> Ty;

data Ty : * {
| tv : tVar -> Ty
| arr : Ty -> Ty -> Ty
| forall : Ty -> Ty
};

data Term : Ctx -> Ty -> * {
| V : (ctx : (Var -> Ty)) -> (x : Var) ->
 Term ctx (ctx x)
| lam : (ctx : Ctx) ->
 (targ : Ty) -> (tbod : Ty) ->
 Term (ext ctx targ) tbod ->
 Term ctx (arr targ tbod)
| app : (ctx : Ctx) ->
 (arg : Ty) -> (bod : Ty) ->
 Term ctx (arr arg bod) ->
 Term ctx arg ->
 Term ctx bod
| tlam : (ctx : Ctx) ->
 (bod : Ty) ->
 Term ctx bod ->
 Term ctx (forall bod)
| tapp : (ctx : Ctx) ->
 (targ : Ty) -> (tbod : Ty) ->
 Term ctx (forall tbod) ->
 Term (tSubCtx targ ctx) (tSubt targ tbod)
};

step : (ctx : Ctx) -> (ty : Ty) -> Term ctx ty ->
 Term ctx ty ;
step ctx ty trm =
case trm <_ => Term ctx ty > {
| (app _ targ tbod (lam _ _ _ bod) a) =>
  sub ctx targ a tbod bod
| x => x
};
```

clean up, write out sub?

Figure 3: System F

SW15, CTW21]. However, these impressive efforts are still only usable by experts, since they frequently require the programer prove their equalities explicitly[Pro13, SW15], or add custom rules into the type system [CTW21]. Further, since program equivalence is undecidable in general, no system will be able to statically verify every “obvious” equality for arbitrary user defined data types and functions. In practice, every dependently typed language has a way to assume equalities, even though these assumptions will result in computationally bad behavior (the program may “get stuck”).

The proposed workflow presented in this thesis is justified by:

- The strict relation between warnings and runtime errors. A runtime error will always correspond exactly to a reported warning, always adding specificity to the the warning that was presented.
- A form of type soundness holds, programs will never “get stuck” unless a concrete counter example to a type assertion is found.
- Programs that type check against a model type system will not have warnings, and therefore cannot have errors.
- Other then warnings and errors the runtime behavior is identical to a conventional type theory.

2 Example

While the primary benefit of this system is the ability to experiment more freely with dependent types, while still getting the full feedback of a dependent type system, it is also possible to encode examples that would be unfeasible in existing systems. This comes from accepting warnings that are justified with external mathematical or programatic intuition, even while being theoretically thorny in dependent type theory.

For instance, here is part of an interpreter for interpreter for System F[5] that encodes the type of the term at the type level. The step function asserts type preservation in its function signature,

the fol-
’ need to
ugs here

It will generate warnings like the following

[5]System F is one of the foundational systems used to study programming languages. It is possible to fully encode evaluation and proofs into Agda, but it is difficult if substitution computation happens in a type. In our system, it is possible to start with the ideal type indexed encoding and build an interpreter, without proving any properties of substitution.

- `tbod` in `Term (tSubCtx targ ctx) (tSubt targ tbod)` may have the wrong type

First note that the program has assumed several of the standard properties of substitution. Formalizing substitution in a dependent type theory is a substantial task[SSK19]. Informally substitution and binding is usually considered obvious and uninteresting, and little explanation is usually given[6].

this might
derselling
ing month

cite some

Second, the type contexts have been encoded as functions. This would be a reasonable encoding in a mainstream functional language since it hides the uninteresting lookup information. This encoding would be unthinkable in other dependently typed languages since equality over functions is so fraught. Here we can rest on our intuition that functions that act the same are the same.

Finally it is perfectly possible that is a bug in the code invalidating one of the assumptions. There are two options for the programmer:

- reformulate the above code so that there are no warnings, formally proving all the required properties in the language (this is possible but would take prohibitive effort with substantial changes)

- exercise the *step* function using standard software testing techniques. If there interpreter does not preserve types, then a concrete counter example can be found

The programmer is free to choose how much effort should go into removing warnings. But even if the programmer wanted a fully formally correct interpreter, it would still be wise to test the functions first before attempting such a proof.

For instance, if the following error is introduced,

Then it will be possible to get the runtime error

Part III
Design Decisions

There are many flavors of dependent types that can be explored, this thesis attempts to always us the simplest and most programmer friendly formulations. Specifically,

- The type system in this thesis is a **full spectrum** dependent type system. The full-spectrum approach is the most uniform approach to dependent type theory, computation behaves the same at the term and type level [Aug98, Nor07, Bra13, SCA+12]. This is contrasted with a leveled theory where terms embedded in types may have different or limited behavior[7]. While the full spectrum approach offers tradeoffs (it is harder to deal with effects), it seems to be the most predictable from the programmer's perspective.

- Data types and pattern matching are essential to practical programming. While it is theoretically possible to simulate data types via church encodings, they are too awkward for programmers to work with, and would complicate the runtime errors this system hopes to deliver. To provide a better programing experience data types are built into the system and pattern matching is supported.

- The theories presented in this thesis will allow unrestricted general recursion and thus allow non-termination. While there is some dispute about how essential general recursion is, there is no mainstream general prepose programing language without it. Allowing nontermination weakens the system when considered as a logic, (any proposition can be given a nonterminating inhabitant). This removes any justification for a type universe hierarchy, so our theories will have type-in-type. Similarly non-strict data definitions will be allowed.

  that mcbr
  "Church's
  Functiona
  ming" (20

- Aside from the non-termination mentioned above, effects will not be considered. Even though effects seem essential to mainstream programing they are a very complicated area of active research that will not be considered here. In this sense the language will be a "pure" functional language like Haskell. As in Haskell, effects can be treated though a functional interface.

It is possible to imagine a system where a wide range of properties are held optimistically and tested at runtime. However the bulk of this thesis will only deal with equality, since that relation is uniquely fundamental to dependent

[6] A convention that will be followed in this thesis

[7] this is the approach taken in ATS, and other refinement type systems

```
Surface language (ch. 2,4)
 {syntax {typed syntax {bidirectionally typed syntax}       }            }
                       |                             |                |
elaboration (ch. 3)    |without warnings             | with warnings  |
                       |                             |                |
                       vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
    {                   cast system (ch. 3)                          }
```

better graphics

Figure 4: Systems in This Thesis

type systems. Since computation can appear at the type level, and types must be checked for equality, dependent type theories must define what computations they intend to equate for type checking. It would be premature to deal with any other properties until equality is dealt with.

Part IV
Issues

Inserting runtime equality checks into a dependent types system is easier said then done.

- In the presence of Dependent types, equality checks may drift into types. What does it mean when a term is a list of length `Bool`?
- Terms can "get stuck" in new way. What happens when an equality check is used as a function by being applied an argument? What happens when a check blocks a pattern match?
- Equality is not decidable at many types even in the empty context. For instance, functions from `Nat` → `Nat` do not have decidable equality.

These problems are solved by extending dependent type theory with a cast checking operator. This cast operator will "get stuck" if their is a discrepancy, and we can show that a program will always resolve to a value or get stuck in such a way that a counterexample can be reported.

- A system is needed to localize casts, these can be generated by extending a **bidirectional** typing procedure to insert checks when it would statically check equality. Without a way to localize errors, the work in this thesis would be infeasible.
- Once the casts are inserted, evaluations are possible. Lazy evaluation means checking only has to happen up to the outermost type constructor, avoiding issues of undecidability.
- When an argument is applied to a cast operator, new casts can be computed using similar logic to that of contracts and monitors[FF02].
- Pattern matching can be extended to support checks, and redirect blame as needed.

Part V
The work in this thesis

While apparently a simple idea, the technical details required to manage checks that delay until runtime in a dependently typed language is fairly involved.

This should probably be expanded

- Chapter 2 describes a dependently typed language intended to model standard dependent type theories (called the **Surface Language**), proves **type soundness**, and presents a bidirectional type checking procedure system intended to model standard type checking.
- Chapter 3 describes a dependently typed language with embedded equality checks, called the **cast language**. The cast language has it's own version of type soundness, called **cast soundness**, which is proven correct. An Elaboration procedure takes most terms of the surface syntax into terms in the cast language. Several desirable properties for elaboration are presented and explored.
- Chapter 4 reviews how dependent data and pattern matching can be added to the surface language.
- Chapter 5 shows how to extend the cast language with dependent data and pattern matching.
- Chapter 6 discusses other ideas related to usability, such as automated testing and runtime proof search.

Versions of the proof of type soundness in Chapter 2, and the cast soundness in Chapter 3 have been formally proven in Coq. Properties in chapter 4 and 5 are only conjectured correct.

Those interested in exploring the meta-theory of a "standard" dependent type theory can read Chapters 2 and 4 which can serve as a self contained tutorial.

References

[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.

[CDT12] The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

[CTW21] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: A type theory with computational assumptions. In *ACM Symposium on Principles of Programming Languages*, 2021.

[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.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. Association for Computing Machinery.

[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.

[ML72] Per Martin-Löf. An intuitionistic theory of types. Technical report, University of Stockholm, 1972.

[MU21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[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.

[Pro13] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[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.

[SSK19] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 166–180, New York, NY, USA, 2019. Association for Computing Machinery.

[SW15] 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.

[Xi07] Hongwei Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 03 2007. Copyright - 2006 Cambridge University Press; Last updated - 2010-06-08.

Part VI
TODO

why bidirectionallity

define curry howard?

Am I still trying to make "gradual correctness" a thing?

quick thakeaways?

file:///Users/stephaniesavir/Downloads/Combining_proofs_and_programs.pdf has a good intro structure, perhaps copy that?

give the overall system a name?

make fonts and styles consistent

Todo list

not clear these are the best citations, there seems to be little consensus online about how to cite software . . 1
cite the languages/ the dependent features both? neither? 1
revise last paragraph 1
expand this example? perhaps at the head of a constant vector first? "and this reasoning can be abstracted under functions" 1
make this a "code" example? 2
Awk, in HM it makes sense to block 2
better graphics 3
better graphics 3
What is new here? talk about some of the prior dependent type attempts, always a subset of syntax, or changes the syntax 3
cite? 3
clean up, write out sub? 4
cite? 4
"It will generate the following warnings" need to clean up some bugs here 4
this might be really underselling it, we are talking months of effort 5
cite some of the libs 5
For instance, if the following error is introduced, ... 5

Then it will be possible to get the runtime error . . . 5
that mcbride paper? "Church's Thesis and Functional Programming" (2004) . . . 5
better graphics . . . 6
This should probably be expanded . . . 6
Curry HOW . . . 9

Part VII
notes

Part VIII
unused

The ultimate goal being that **it should be easier to write programs with dependent types then without**.

Curry HOW

According to the Curry-Howard correspondence[8] types correspond to proposition and proofs correspond to programs. This gives programmers an unrivaled degree of freedom and precision when specifying and verifying their code.

3 Error msgs

If programmers found dependent type systems easier to learn and use, software could become more reliable. Unfortunately, dependent type systems have yet to see widespread use in industry. One source of difficulty is the conservative equality checking required by most dependent type systems. This conservative equality is a source of some of the confusing error messages dependent type systems are known for [ESH19].

This error will help the programmer fix the bug in `add`. There is 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].

[8] Also called...