Chapter 1 (draft): Introduction

Mark Lemay

December 23, 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 offer a simpler approach. In dependent type systems, proofs and properties use the same language and meaning already familiar to functional programmers.

what is a type, what is a dependent type. : In mainstream type systems, the types relate to the

For instance, dependent type systems can prevent 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"

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

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

make this a "code" example?

Where `Vec` is a length indexed by the type of element it contains and its length, it is a type that depends on an term level expression for 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 $\texttt{add}\,1\,x$, perhaps retuning the first element of that non-empty list.

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

This example only scratches the surface of what is possible with dependent types. According to the Curry-Howard correspondence[1] 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.

For instance

The power of dependent types has been recognized for decades. Dependent types form the back bone of several poof system, such as Coq, Lean, and Agda. They have have been proposed as a foundation for mathematics. They are directly used in several programming languages such as ATS and Idris, and dependent types have influenced many other programing languages.

Curry-Howard

[1] Also called...

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

better graphics

Figure 1: Standard Typed Programming Workflow

Unfortunately, programmers often find dependent type systems difficult to learn and use. This resistance has limited the ability of dependent to reach their potential. 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 "definitionally equal". "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 $\mathtt{Vec}\,(\mathtt{add}\,1\,x)$ where a term of type $\mathtt{Vec}\,(\mathtt{add}\,x\,1)$ is expected. Usually when dependent type systems encounter situations like this, they will give an error message and block evaluation until the "error" is resolved.

While the intent and properties of the add function is clear 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 type "error" is removed.

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

Usually types are justified to avoid programs that "get stuck". If it is the case that $\mathtt{add}\,1\,x = \mathtt{add}\,x\,1$ the program will never get stuck. If there is a mistake in the implementation of add, we are able to stop the program execution and provide a concrete witness for the inequality. For instance, if the add function incorrectly computes $\mathtt{add}\,8\,1 = 0$ the above function will "get stuck" on the input 8. If that application is encountered at runtime we can give an error stating $\mathtt{add}\,1\,8 = 9 \neq 0 = \mathtt{add}\,8\,1$.

Part II
A Different Workflow

This thesis advocates an alternative pragmatic view of types. Where "the programer is always right (until proven wrong)". I expect this slogan will go over better with programmers then the existing "Can't run programs until the type system is convinced [2]".

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

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

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

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.

[2]often requiring a graduate degree and uncommon patients

```
edit program
 |
 |
 v
 Elaboration                              ^
 |                |                       |  runtime error
 | no warnings    |  type warnings        |
 v                v                       |
    run program      _____________________
```

better graphics

Figure 2: Workflow for this Thesis

The standard workflow seems sufficient for type systems in many mainstream programing languages. Though there is experimental evidence that even OCaml can be easier to use with the proposed workflow [SJW16]. However in the presence of dependent types the standard workflow is challenging for beginners and experts alike.

The difficulty with dependently typed languages has been reflected in the consensus that dependent type errors are bad[ESH19]. However the problem is likely a mismatch between the expectations of the programmer and the design of the underling type theory. Better worded english sentences are unlikely to bridge this gap when the type system doubts x+1 = 1+x.

By switching to the proposed workflow, type errors become type warnings, and the programer is fee 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 would be justified in calling their program a proof. If there is value in a type error message it comes from the message itself and not the hard stop it puts to programming.

The proposed workflow is further justified, since often the type system is too conservative and the programmer is correct in implicitly asserting an equality. That the programmer may need to go outside the conservative bounds of definitional equality has been recognized since the earliest dependent type theories . Difficulties in dependently typed equality have motivated many research projects [Pro13, SW15, CTW21]. However, these impressive efforts are still only used by experts. 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 dependent typed language has a way to assume equalities, even though these assumptions will result in computationally bad behavior (the program may "get stuck").

ML

Finally this proposed workflow is justified by:

- The relation between warnings and runtime errors, in that 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 is found.
- Programs that type check against a model type system will not have warnings, and therefore cannot have errors.
- Other then warnings and error the runtime behavior is identical to the model system.

why focus on equality?

why is this hard?

1 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 by warnings that are justified with external mathematics or programatic intuition, while being theoretically thorny in dependent type theory.

```
Var : * ;
Var = Nat

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

For instance, here is an interpreter for System F[3] that encodes the type of the term. The interpreter function asserts type preservation in its function signature,

the fol-
,

It will generate warnings like the following

...

ally un-
are talk-
fort

libs

First note that the program has assumed several of the standard properties of substitution. Formalizing substitution is in a dependent type theory is a substantial task. Informally substitution and binding is usually considered obvious and uninteresting, and little explanation is usually given[4].

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, essentially formally proving all the required properties (this is possible but would take prohibitive effort)
- exercise interpreter using standard software testing techniques. If there interpreter does not preserve types, then a concrete counter example will 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

...

[3]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.

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

```
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

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 theories in this thesis is considered **full-spectrum**. 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]. This is contrasted with a leveled theory where terms embedded in types may be limited in their behavior, this is the approach taken in ATS. 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.

- The theories presented in this thesis will allow unrestricted general recursion and thus non-termination. While there is some dispute about essential general recursion is, there is no mainstream programing language without it. This removes any justification for a type universe hierarchy, so our theories will have type-in-type.

that mcbr

- Aside from nontermination, effects will not be considered. Even though effects seem essential to standard programing they are a very complicated area of active research that will not be considered here. In this sense the language will be pure like Haskell.

Part IV
The work in this thesis

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

This should probly be expanded

Chapter 2 describes a dependently typed language intended to model standard dependent type theories (called the **Surface Language**) and 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 type checking, 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.

Versions of the proof of type soundness in Chapter 2, and the cast soundness in Chapter 3 have been formally proven in Coq[5].

Those interested in exploring the metatheory of standard dependent type theory can read chapters 2 and 4.

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.

[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.

[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.

[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.

[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.

Part V
TODO

Am I still trying to make gradual correctness a thing?

quick thakeaways?

[5]The formalization is due to Robin, using libraries...

Todo list

Part VI
notes

Part VII
unused

2 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].