Draft

August 26, 2021

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 [1, 9, 2, 11]. 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

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

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

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

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 [5].

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

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

Obviously $1 + x = x + 1$. However in the majority of dependently typed programming languages, $\mathtt{add}\,1\,x \equiv \mathtt{add}\,x\,1$ is not a definitional equality. This means a term of type $\mathtt{Vec}\,(\mathtt{add}\,1\,x)$ cannot be used 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 a type error and prevent evaluation. If the programmer made a mistake in the definition of addition such that $\mathtt{add}\,1\,x \not\equiv \mathtt{add}\,x\,1$, 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 $\mathtt{add}\,8\,1 = 0$ the above function will "get stuck" on the input 8. If that application is encountered at runtime we can give the error $\mathtt{add}\,1\,8 = 9 \not\equiv 0 = \mathtt{add}\,8\,1$.There is some evidence that specific examples like this can help clarify the type error messages in OCaml [10] and there has been an effort to make refinement type error messages more concrete in other systems like Liquid Haskell [6].

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

In an ideal world programmers would write perfect code with perfectly proven equalities. The surface language models this ideal, but difficult, system. The surface language enforces definitional equality, and is a standard well behaved core calculus. Programmers should "think" in the surface language, and the machinery of later sections should reinforce an understanding of the surface type system, while being transparent to the programmer.

The surface language presented here is a minimal intensional dependent type theory. The surface language allows some programmatic features at the expense of logical soundness. The language allows general recursion, since general

source labels,			
ℓ			
variable contexts,			
Γ	$::=$	$\lozenge \mid \Gamma, x : M$	
expressions,			
m, n, h, M, N, H	$::=$	x	variable
	$\mid$	$m ::_{\ell} M$	annotation
	$\mid$	$\star$	type universe
	$\mid$	$(x : M_{\ell}) \to N_{\ell'}$	function type
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	function
	$\mid$	$m_{\ell}\, n$	application
values,			
v	$::=$	$x \mid \star$	
	$\mid$	$(x : M_{\ell}) \to N_{\ell'}$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	

Figure 1: Surface Language Pre-Syntax

recursion is useful for general purpose functional programming. It also supports type-in-type, since it simplifies the system for programmers and makes the metatheory slightly easier.

Though similar systems have been studied over the last few decades this chapter gives a self contained presentation of important meta-theoretic results sometimes simplified and with modern notation, in addition to many examples. The surface language has been an excellent platform to conduct research into "full spectrum" dependent type theory, and hopefully this exposition will be helpful for future researchers.

1 Formal Surface Language

The pre-syntax can be seen in figure 1. Location data ℓ is marked at every position in syntax where a type error might occur. When unnecessary the location information ℓ will be left implicit.

2 Examples

The surface system is extremely expressive. Church encodings are expressible.

2.1 Church Booleans

$\mathbb{B}_c \coloneqq (A : \star) \to A \to A \to A$
$\quad true_c \coloneqq \lambda A.\lambda then.\lambda else.then$
$\quad false_c \coloneqq \lambda A.\lambda then.\lambda else.else$

$$\frac{}{x \Rrightarrow x}$$

$$\frac{m \Rrightarrow m'}{m ::_\ell M \Rrightarrow m'}$$

$$\frac{m \Rrightarrow m' \quad M \Rrightarrow M'}{m ::_\ell M \Rrightarrow m' ::_\ell M'}$$

$$\frac{M \Rrightarrow M' \quad N \Rrightarrow N'}{(x : M_\ell) \to N_{\ell'} \Rrightarrow (x : M'_\ell) \to N'_{\ell'}}$$

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{(\mathsf{fun}\, f\, x \Rightarrow m)_\ell\, n \Rrightarrow m'\, [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n']}$$

$$\frac{m \Rrightarrow m'}{\mathsf{fun}\, f\, x \Rightarrow m \;\Rrightarrow\; \mathsf{fun}\, f\, x \Rightarrow m'}$$

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{m_\ell\, n \Rrightarrow m'_\ell\, n'}$$

$$\frac{x : M \in \Gamma}{\Gamma \vdash x \,:\, M} var - ty$$

$$\frac{\Gamma \vdash m \,:\, M \quad \Gamma \vdash M \,:\, \star}{\Gamma \vdash m ::_\ell M \,:\, M} :: -ty$$

$$\frac{}{\Gamma \vdash \star \,:\, \star} \star -ty$$

$$\frac{\Gamma \vdash M \,:\, \star \quad \Gamma, x : M \vdash N \,:\, \star}{\Gamma \vdash (x : M) \to N \,:\, \star} \Pi - ty$$

$$\frac{\Gamma \vdash m \,:\, (x : N) \to M \quad \Gamma \vdash n \,:\, N}{\Gamma \vdash m\, n \,:\, M\, [x \coloneqq n]} \Pi - app - ty$$

$$\frac{\Gamma \vdash m \,:\, M \quad \Gamma \vdash M \equiv M' : \star}{\Gamma \vdash m \,:\, M'} conv$$

$$\frac{\Gamma, f : (x : N) \to M, x : N \vdash m \,:\, M}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \,:\, (x : N) \to M} \Pi - \mathsf{fun} - ty$$

2.2 Church $\mathbb{N}$

$\mathbb{N}_c \coloneqq (A : \star) \to (A \to A) \to A \to A$

$0_c \coloneqq \lambda A.\lambda s.\lambda z.z$

$1_c \coloneqq \lambda A.\lambda s.\lambda z.s\, z$

$2_c \coloneqq \lambda A.\lambda s.\lambda z.s\ (s\, z)$

$n_c \coloneqq \lambda A.\lambda s.\lambda z.s^n\, z$

$suc_c\, x \coloneqq \lambda A.\lambda s.\lambda z.s\, (x\, A\, s\, z)$

$x +_c y \coloneqq \lambda A.\lambda s.\lambda z.x\, A\, s\ (y\, A\, s\, z)$

2.3 Unit

$Unit \coloneqq (A : \star) \to A \to A$

$tt \coloneqq \lambda A.\lambda a.a$

2.4 Void

$\bot \coloneqq \mathit{\Pi} x : \star.x$

Calculus of Constructions constructions encodings are expressible,

2.5 Negation

$\neg A \coloneqq A \to \bot$

2.6 Leibniz equality

$a_1 =_A a_2 \coloneqq \mathit{\Pi} C : (A \to \star)\,.C\, a_1 \to C\, a_2$

$refl_{a:A} \coloneqq \lambda C : (A \to \star)\,.\lambda x : C\, a.x \qquad : a =_A a$

$\neg A \coloneqq A \to \bot$

2.7 Large Elimination

"Large eliminations" are possible with type-in-type.

$\lambda b.b\ \star\ Unit\ \bot \quad : \mathbb{B}_c \to \star$

$\lambda n.n\ \star\ (\lambda - .Unit)\ \bot \quad : \mathbb{N}_c \to \star$

Note that such a function is not possible in the Calculus of Constructions (CC).

large eliminations can prove standard inequalities that can be hard or impossible to express in other dependent type theories

2.8 $\neg \star =_\star \bot$

$\lambda pr.pr\ (\lambda x.x)\ \bot \qquad : \neg \star =_\star \bot$

2.9 $\neg Unit =_\star \bot$

$\lambda pr.pr\ (\lambda x.x)\, tt \qquad : \neg Unit =_\star \bot$

2.10 $\neg true_c =_{\mathbb{B}_c} false_c$

$\lambda pr.pr\ (\lambda b.b \star Unit\ \bot)\ tt \qquad : \neg true_c =_{\mathbb{B}_c} false_c$

2.11 $\neg 1_c =_{\mathbb{N}_c} 0_c$

$\lambda pr.pr\ (\lambda n.n \star (\lambda - .Unit)\ \bot)\ tt \qquad : \neg 1_c =_{\mathbb{N}_c} 0_c$

Such a proof is impossible in CC

2.12 $(x : \mathbb{N}_c) \to 0_c +_c x =_{\mathbb{N}_c} x +_c 0_c$ (by recursion)

$\mathsf{fun}\, f\, x \Rightarrow x\ (0_c +_c x =_{\mathbb{N}_c} x +_c 0_c)\ f\ (refl_{0_c:\mathbb{N}_c}) \qquad : (x : \mathbb{N}_c) \to 0_c +_c x =_{\mathbb{N}_c} x +_c 0_c$

TODO: check and discuss

2.13 Every type is inhabited (by recursion)

$\mathsf{fun}\, f\, x \Rightarrow f\, x \qquad : \bot$

This shows that the surface language is "logically unsound", every type is inhabited. while the surface language supports proofs, not every term typed in the surface language is a proof.

2.14 Every type is inhabited (by Type-in-type)

It is possible to encode Gerard's paradox, producing another source of logical unsoundness. Though a subtle form of recursive behavior can be built out of Gerard's paradox, direct inclusion of recursion is much easier to work with.

...

There are more examples in [3] where Cardelli has studied a similar system.

3 Meta-theory

The type assignment system can be shown sound using a progress and preservation style proof. The key is to show that computation is confluent and use that computation to generate the definitional equality relation. This allows definitional equality to distinguish constructors while still being easy to prove an equivalence. Computation can be shown confluent using parallel-reductions [12].

3.1 Preservation

3.2 Progress

3.3 type soundness

The language has type soundness, well typed terms will never "get stuck" in the surface language.

$$\frac{x : M \in \Gamma}{\Gamma \vdash x \overset{\rightarrow}{:} M}\ \text{var-ty}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \star \overset{\rightarrow}{:} \star}\ \star\text{-ty}$$

$$\frac{\Gamma \vdash m \overset{\leftarrow}{:} M \quad \Gamma \vdash M \overset{\leftarrow}{:} \star}{\Gamma \vdash m ::_{\ell} M \overset{\rightarrow}{:} M}\ ::\text{-ty}$$

$$\frac{\Gamma \vdash M \overset{\leftarrow}{:} \star \quad \Gamma, x : M \vdash N \overset{\leftarrow}{:} \star}{\Gamma \vdash (x : M) \to N \overset{\rightarrow}{:} \star}\ \Pi\text{-ty}$$

$$\frac{\Gamma \vdash m \overset{\rightarrow}{:} (x : N) \to M \quad \Gamma \vdash n \overset{\leftarrow}{:} N}{\Gamma \vdash m\, n \overset{\rightarrow}{:} M\,[x := n]}\ \Pi\text{-app-ty}$$

$$\frac{\Gamma \vdash m \overset{\rightarrow}{:} M \quad \Gamma \vdash M \equiv M' : \star}{\Gamma \vdash m \overset{\leftarrow}{:} M'}\ \text{conv}$$

$$\frac{\Gamma, f : (x : N) \to M, x : N \vdash m \overset{\leftarrow}{:} M}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \overset{\leftarrow}{:} (x : N) \to M}\ \Pi\text{-fun-ty}$$

Figure 2: Surface Language Bidirectional Typing Rules

3.4 Type checking is undecidable

Given a thunk $f : Unit$ defined in pcf, it can be encoded into the surface system as a thunk $f' : Unit$, such that if f reduces to the canonical unit then $f' \Rrightarrow^{*} \lambda A.\lambda a.a$

$\vdash \star : f'\ \star\ \star$ type-checks by conversion exactly when f halts

If there is a procedure to decide type checking we can decide exactly when any pcf function halts

4 Bi-directional surface language

4.1 the basic surface language is impractical as a programming language

typing is not unique up to conversion

$\lambda x.x$ has many types

TODO

4.2 ...

The surface language supports bidirectional type-checking over the pre-syntax with the rules in figure 2. Bidirectional type-checking is a form of lightweight type inference, and strikes a good compromise between the needed type annotations and the simplicity of the theory. This is accomplished by breaking typing judgments into 2 forms:

- Inference judgments where type information propagates out of a term, $\overset{\rightarrow}{:}$ in our notation.
- And Checking judgments where a type is checked against a term, $\overset{\leftarrow}{:}$ in our notation.

Unfortunately, the system is logically unsound (every type is trivially inhabited with recursion), since our language attempts to be more oriented to programs than proofs. We expect this is acceptable.

It might seem restrictive that the surface language only supports dependent recursive functions. However, this is extremely expressive: church style data can be encoded, as can calculus of construction style predicates, recursion can simulate induction, and type-in-type allows large elimination (see [3] for examples). This is still inconvenient, so we have implemented dependent data in our prototype. We suggest ways dependent data could be added to the theory in Section 4.

4.3 Bi-directional metatheory

4.3.1 If it types in the bidirectional system then it types in the TAS system

4.3.2 If it types in the TAS system annotations can be added such that an equivalent term types in the bidirectional system

4.3.3 Type-checking in the Bi-directional system is still undecidable

Type checking remains undecidable because of our addition of general recursion and type-in-type. However, since the user is not expected to type-check their program directly this should not cause any issues in practice.

5 Implementation

We have mechanized the type soundness of the type assignment system (without location data) in Coq.

6 Related work

It should be noted that similar systems have been studied going back to [8] before type-in-type was known to be unsound. The semantics was further explored in [3] and an early bidirectional type-checking algorithm for a similar language is specified in [4]. Cayenne [1], a Haskell-like language, combined dependent types with type-in-type and non-termination. It was more recently explored in the context of call by value evaluation in [7] and [11]. Though not novel, we believe our Coq proof to be the clearest formal exposition to date.

A similar proof of type soundness appears in [11].

References

[1] 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.

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

[3] Luca Cardelli. A polymorphic [lambda]-calculus with type: Type. Technical report, DEC System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 1986.

[4] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996.

[5] 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.

[6] 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.

[7] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. *SIGPLAN Not.*, 45(1):275–286, January 2010.

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

[9] 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.

[10] 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.

[11] 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.

[12] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.