

Chapter 4 (draft): Data and Pattern Matching in the Surface Language

Mark Lemay

December 26, 2021

“not handle data types in a rigorous way.”

Part I

Introduction

User defined data is an important part of a realistic programming language. Programmers need to be able to define concrete types that are meaningful for the problems they are trying to solve.

Dependent data types allow these user defined types, while also unifying over many types that are handled as special cases in most mainstream languages. For instance, “primitive” data types like `Nat` and `Bool` are a degenerate forms of dependent data. Dependent data can represent mathematical predicates like equality or the evenness of a number. Dependent data can also be used to preserve invariants, like the length of a list in `Vec`, or the “color” of a node in a red-black-tree.

The encoding scheme for data presented in Chapter 2 could handle all of these cases. However, such an encoding is inconvenient in practice. Since “ease of use” is the overriding concern for the system developed in this thesis, encoding are an unrealistic way to use data.

In this chapter we will show 2 different, closely related ways to add data to the surface language. The first which we call the direct eliminator encoding, is metatheoretically well behaved but cumbersome. The second is based on pattern matching, and is extremely useful for programmers, though it’s meta theory is much more difficult. The specific form of pattern matching in this chapter is designed to allow syntactic sugar of function definition by cases.

Part II

Data

A dependent data type is defined by a type constructor indexed by arguments, and a set of data constructors that tag data and characterize their arguments. Several familiar data types are defined in Figure 1. For example, the data type `Nat` is defined with the type constructor `Nat` (which has no type arguments), the data constructors `Z` which takes no further information and the data constructor `S` which is formed with the prior number. The data type `Vec` has two type arguments corresponding to the type contained in the vector and its length, it has two data constructors that allow building an empty vector, or to add an element to the front of an existing vector.

Data defined in this style is simple to build and reason about, since data can only be created from its constructors. Unfortunately the details of data elimination are a little more involved.

Part III

Direct Elimination

How should a program observe data? Since the term of a given data type can only be created with one of the constructors from its definition, we can completely handle a data expression if each possible constructor is accounted

```

data Unit : * {
| tt : Unit
};

data Bool : * {
| True : Bool
| False : Bool
};

data Nat : * {
| Z : Nat
| S : Nat -> Nat
};
three : Nat;
three = S (S (S Z));
-- Syntactic sugar allows 3 = S (S (S Z))

data Vec : (A : *) -> Nat -> * {
| Nil : (A : *) -> Vec A Z
| Cons : (A : *) -> A -> (x : Nat)
        -> Vec A x -> Vec A (S x)
};

someBools : Vec Bool 2;
someBools = Cons Bool True 1 (Cons Bool False 0 (Nil Bool));

data Id : (A : *) -> A -> A -> * {
| refl : (A : *) -> (a : A) -> Id A a a
};

threeEqThree : Id Nat three three;
threeEqThree = refl Nat three;

```

Figure 1: Definitions of Common Data Types

list of O , separated with s		
$\overline{sO}, \overline{Os}$	$::=$	$.$
	$ $	\overline{sOsO}
Δ, Θ	$::=$	$(x : M) \rightarrow$
data type identifier,		
D		
data constructor identifier,		
d		
contexts,		
Γ	$::=$	\dots
	$ $	$\Gamma, \text{data } D : \Delta \rightarrow \star \left\{ \overline{ d : \Theta \rightarrow D\overline{m} } \right\}$
	$ $	$\Gamma, \text{data } D : \Delta \rightarrow \star$
m, n, M, N	$::=$	\dots
	$ $	D
	$ $	d
	$ $	$\text{case } \overline{N}, n \left\{ \overline{ x \Rightarrow (d \overline{y}) \Rightarrow m } \right\}$
	$ $	$\text{case } \overline{N}, n \langle x \Rightarrow y : D \overline{x} \Rightarrow M \rangle \left\{ \overline{ x \Rightarrow (d \overline{y}) \Rightarrow m } \right\}$
values,		
v	$::=$	\dots
	$ $	$D \overline{v}$
	$ $	$d \overline{v}$

Figure 2: Surface Language (Direct Eliminator) Data

	syntax	written	when	for example	is w
leading separator	\overline{sOsO}	\overline{OsO}	clear from context	, 1, 2, 3	1, 2,
trailing separator	\overline{sO}	\overline{sOs}	clarifies intent	$(x \rightarrow y \rightarrow z) \text{Id } x y z$	$x \rightarrow$
non dependent telescope binder	$(x : M) \rightarrow$	$M \rightarrow$	x is not intended to bind	$(x : \text{Nat}) \rightarrow (y : \text{IsEven } x) \rightarrow \text{Nat}$	$(x :$
repeated application	$m \ n_0 \ n_1 \dots$	$m \ \overline{n}$	$\overline{n} = n_0 \ n_1 \dots$		

review if there are more abbreviations around?

Figure 3: Surface Language Abbreviations

for. For instance, **Nat** has the two constructors **Z** and **S** (which holds the preceding number), so the expression $\text{case } n \{ | Z \Rightarrow Z | S x \Rightarrow x \}$ will extract the proceeding number from n (or 0 if $n = 0$). In this light, boolean case elimination corresponds to the if-then-else expression found in many mainstream languages.

We will need to extend the syntax of **cases** to support dependent type checking. Specifically, we will need to add a **motive** annotation that allows the type checker to compute the output type of the branches if they vary in terms of the input. For instance, recursively generating a vector of a given length. We may also want to use some values of the type level argument to calculate the motive, and type the branches. This will be allowed with additional bindings in the motive and in each branch. In general, motive annotations will be treated like the typing annotations in Chapter 2, in that the TAS will only allow correct motives in a well typed term, and that the motive will be definitionally irrelevant.

This version of data can be given by extending the surface language syntax in Chapter 2, as in Figure 1. This direct eliminator scheme, is roughly similar to how Coq handles data in it's core language.

motive should not need to insist on the type info of the binder? grey out? Grey out things that are surface syntax but not needed for theory?

The case eliminator first takes the explicit type arguments, followed by a **scrutinee**¹ of correct type. Then optionally a motive that characterizes the output type of each branch with all the type arguments and scrutinee abstracted and in scope. For instance, this case expression checks if a vector x is empty,

$x : \text{Vec } \mathbb{B} \ 1 \vdash \text{case } \mathbb{B}, 1, x \langle y \Rightarrow z \Rightarrow s : \text{Vec } y z \Rightarrow \mathbb{B} \rangle \{ | y \Rightarrow z \Rightarrow \text{Nil} \Rightarrow \text{True} | y \Rightarrow z \Rightarrow \text{Cons } \dots \Rightarrow \text{False} \}$

¹also called a **discriminee**

We include a little more syntax than is strictly necessary, since the $\mathbb{B}, 1$, list could be inferred and the $y \Rightarrow z \Rightarrow$ binders are not needed in the branch. This slightly verbose case eliminator syntax is designed to be forward compatible with the pattern matching system defined in the rest of this chapter.

Additionally we define telescopes, which generalize zero or more typed bindings. This allows a much cleaner definition of data than is otherwise possible. Also we define syntactic lists that allow zero or more pieces of syntax. Expressions in a list can be used to generalize dependent pairs, and can be type checked against a telescope. For instance, the list $\text{Nat}, 2, 2, \text{refl Nat}$ type checks against $(x : \star) \rightarrow (y : \text{Nat}) \rightarrow (z : \text{Nat}) \rightarrow (- : \text{Id } x y z)$. This becomes helpful in several situations, but especially when we need work with the listed arguments of the data type constructor. We will allow several syntactic puns, such as treating telescopes as prefixes for function types. For instance, if $\Delta = (x : \star) \rightarrow (y : \text{Nat}) \rightarrow (z : \text{Nat}) \rightarrow (- : \text{Id } x y z)$ then writing $f : \Delta \rightarrow \text{Nat}$ will be short hand for $f : (x : \star) \rightarrow (y : \text{Nat}) \rightarrow (z : \text{Nat}) \rightarrow \text{Id } x y z \rightarrow \text{Nat}$.

In the presence of general recursion case elimination is powerful. Well-founded recursion can be used to make structurally inductive computations that can be interpreted as proofs.

Adding data allows for two additional sources of bad behavior. Incomplete matches, and nontermination from non-strictly positive data.

1 Incomplete Eliminations

Consider the match

$$x : \mathbb{N} \vdash \text{case } x \langle s : \mathbb{N} \Rightarrow \mathbb{B} \rangle \{ | S - \Rightarrow \text{True} \}$$

This match will “get stuck” if 0 is substituted for x . Recall that the key theorem of the surface language is type soundness, “well typed terms don’t get stuck”. Since verifying every constructor has a branch is relatively easy, the surface language TAS will require every constructor to be matched in order to type check type check with direct elimination. This is in contrast to most programming languages, which do allow incomplete patterns, though usually a warning is given, and a runtime error is raised if the scrutinee cannot be matched.

This thesis already has a system for handling warnings and runtime errors through the cast language. When we get to the cast language, we will allow non-exhaustive data to be reported as a warning and that will allow “unmatched” errors to be observed at runtime.

For similar reasons we will insist that each constructor is matched at most once, so there is no ambiguity for how an case eliminated when using direct eliminations.

2 (non) Strict Positivity

A more subtle concern is posed by data definitions that are not strictly positive. Consider the following definition,

```
data Bad : * {
| C : (Bad -> Bad) -> Bad
};

selfApply : Bad -> Bad;
selfApply = \ b =>
  case b {
  | C f => f b
  };

loop : Bad;
loop = selfApply (C selfApply)
```

The C constructor in the definitions of Bad has a self reference in a negative position, $(\text{Bad} \rightarrow \underline{\text{Bad}}) \rightarrow \text{Bad}$.

non strictly positive data definitions can cause non-termination, independent of the two other sources of non-termination already considered (general recursion and type-in-type). Dependent type systems usually require a strictness check on data definitions to avoid this possibility. However, this would disallow some useful constructions like higher order abstract syntax. Since non-termination is already allowed in the surface TAS, we will not restrict the surface language to strictly positive data.

$$\begin{array}{c}
\frac{\Gamma \mathbf{ok}}{\Gamma \vdash . \mathbf{ok}} \mathbf{ok-Tel-empty} \\
\\
\frac{\Gamma \vdash M : \star \quad \Gamma, x : M \vdash \Delta \mathbf{ok}}{\Gamma \vdash (x : M) \rightarrow \Delta \mathbf{ok}} \mathbf{ok-Tel-ext} \\
\\
\frac{\Gamma \mathbf{ok}}{\Gamma \vdash, : .} \mathbf{ty-ls-empty} \\
\\
\frac{\Gamma, x : M \vdash \Delta \quad \Gamma \vdash m : M \quad \Gamma \vdash \bar{n}, [x := m] : \Delta [x := m]}{\Gamma \vdash m, \bar{n} : (x : M) \rightarrow \Delta} \mathbf{ty-ls-ext}
\end{array}$$

Figure 4: Meta rules

$$\begin{array}{c}
\frac{\Gamma \vdash \Delta \mathbf{ok}}{\Gamma \vdash \mathbf{data} D \Delta \mathbf{ok}} \mathbf{ok-abs-data} \\
\\
\frac{\Gamma \vdash \mathbf{data} D \Delta \mathbf{ok}}{\Gamma, \mathbf{data} D \Delta \mathbf{ok}} \mathbf{ok-data-ext} \\
\\
\frac{\Gamma \vdash \mathbf{data} D \Delta \mathbf{ok} \quad \forall d. \Gamma, \mathbf{data} D \Delta \vdash \Theta_d \mathbf{ok} \quad \forall d. \Gamma, \mathbf{data} D \Delta, \Theta_d \vdash \bar{m}_d : \Delta}{\Gamma \vdash \mathbf{data} D : \Delta \left\{ \overline{d : \Theta_d \rightarrow D \bar{m}_d} \right\} \mathbf{ok}} \mathbf{ok-data} \\
\\
\frac{\Gamma \vdash \mathbf{data} D : \Delta \left\{ \overline{d : \Theta \rightarrow D \bar{m}} \right\} \mathbf{ok}}{\Gamma, \mathbf{data} D : \Delta \left\{ \overline{d : \Theta \rightarrow D \bar{m}} \right\} \mathbf{ok}} \mathbf{ok-data-ext}
\end{array}$$

Figure 5: Surface Language Data ok

3 Typing

Before the typing rules for data can be considered, first some meta rules must be presented that will allow the simultaneous type-checking of lists and telescopes. These rules are listed in 4, and are standard. Telescopes are **ok** when they extend the context in an **ok** way. Lists of expressions can be said to have the type of the telescope if every expression in the list types successively.

Data definitions can be added to contexts if all of their constituents are well typed and **ok**. The rules are listed in Figure 5. The **ok-abs-data** rule allows data to be considered abstractly if it is formed with a plausible telescope. **ok-data** checks a full data definition with an abstract reference to a data definition in context, which allows recursive data definitions such as **Nat** which needs **Nat** to be in scope to define the **S** constructor. This thesis does not formalize a syntax that adds data to context, though a very module system has been implemented in the prototype. It is taken for granted that any well formed data in context is fine. This presentation of data definitions largely follows [SCA⁺12].

The type assignment system with direct elimination must be extended with the rules in 6. These rules make use of several convenient shorthands: $\mathbf{data} D \Delta \in \Gamma$ and $d : \Theta \rightarrow D \bar{m} \in \Gamma$ extract the type constructor definitions and data constructor definitions from the context respectively. **ty-TCon** and **ty-Con** allow type and data constructors to be used as functions of appropriate type. The **ty-case** $\langle \rangle$ rule types a case expression by ensuring that the correct data definition for D is in context, the scrutinee n has the correct type, the motive M is well formed under the type arguments and the scrutinee, finally every data constructor is verified to have a corresponding branch. **ty-case** allows for the same typing logic, but does not require the motive be annotated in syntax. In both rules we allow telescopes to rename their variables with the shorthand $\bar{x} : \Delta$.

suspect this also hinges on regularity, which should be addressed more directly

Extensions to the parallel reduction rules are listed in Figure 7. They follow the scheme of parallel reductions laid out in chapter 2. The \Rightarrow -**case-red** rule² reduces a case expression by choosing the appropriate branch. The \Rightarrow -**case** $\langle \rangle$ -red rule removes the motive annotation, much like the annotation rule in Chapter 2. The rules \Rightarrow -**case** $\langle \rangle$, \Rightarrow - D , and \Rightarrow - d keep the \Rightarrow relation reflexive. The reduction relation must be generalized to lists in the

²Also called ι , or Iota reduction

$$\begin{array}{c}
\frac{\Gamma \text{ ok} \quad \text{data } D \Delta \in \Gamma}{\Gamma \vdash D : \Delta \rightarrow \star} \text{ty-TCon} \\
\\
\frac{\Gamma \text{ ok} \quad d : \Theta \rightarrow D\bar{m} \in \Gamma}{\Gamma \vdash d : \Theta \rightarrow D\bar{m}} \text{ty-Con} \\
\\
\begin{array}{c}
\text{data } D \Delta \{ \dots \} \in \Gamma \\
\Gamma \vdash n : D\bar{N} \\
\Gamma, \bar{x} : \Delta, z : D\bar{x} \vdash M : \star \\
\forall d : \Theta \rightarrow D\bar{o} \in \Gamma. \quad \Gamma, \bar{y}_d : \Theta \vdash m_d [\bar{x} := \bar{o} [\Theta := \bar{y}_d]] : M [\bar{x} := \bar{o} [\Theta := \bar{y}_d], z := d\bar{y}_d] \\
\text{No duplicate branches}
\end{array} \\
\hline
\begin{array}{c}
\Gamma \vdash \text{case } \bar{N}, n \langle \bar{x} \Rightarrow z : D\bar{x} \Rightarrow M \rangle \left\{ \overline{[\bar{x} \Rightarrow d\bar{y}_d \Rightarrow m_d]} \right\} \\
: M [\bar{x} := \bar{N}, z := n]
\end{array} \text{ty-case } \langle \rangle \\
\\
\begin{array}{c}
\text{data } D \Delta \{ \dots \} \in \Gamma \\
\Gamma \vdash n : D\bar{N} \\
\Gamma, \bar{x} : \Delta, z : D\bar{x} \vdash M : \star \\
\forall d : \Theta \rightarrow D\bar{o} \in \Gamma. \quad \Gamma, \bar{y}_d : \Theta \vdash m_d [\bar{x} := \bar{o} [\Theta := \bar{y}_d]] : M [\bar{x} := \bar{o} [\Theta := \bar{y}_d], z := d\bar{y}_d]
\end{array} \\
\hline
\begin{array}{c}
\Gamma \vdash \text{case } \bar{N}, n \left\{ \overline{[\bar{x} \Rightarrow d\bar{y}_d \Rightarrow m_d]} \right\} \\
: M [\bar{x} := \bar{N}, z := n]
\end{array} \text{ty-case}
\end{array}$$

Figure 6: Surface Language Data Typing

expected way.

We are now in a position to select a sub relation of \Rightarrow reductions that will be used to characterize call-by-value evaluation. This relation could be used to prove type safety, and is close to the reduction used in the implementation. The rules are listed in Figure 8.

extend cbv over lists

extend step over lists

Finally we characterize what it means for a context to be empty in the presence of data in Figure 9.

While a system with a similar presentation has proven type soundness in [SCA⁺12], we will not prove the type soundness of the system here. For clarity we will list the important properties as conjectures.

Conjecture the surface language extended with data and elimination preserves types over reduction.

Conjecture the surface language extended with data and elimination has progress if $\Gamma \text{ Empty}$, $\Gamma \vdash m : M$, then m is a value, or $m \rightsquigarrow m'$.

Conjecture the surface language extended with data and elimination is type sound.

4 Bidirectional Type Checking

A bidirectional type checking procedure exists for the type assignment rules listed above. An outline of these rules is in Figure 9. As noted in [DK21], the bidirectional rules around data are open to some interpretation. The dependent case simplifies these questions since only a few schemes are feasible.

The type of data constructors and type constructors can always be inferred. If the motive does not depend on the scrutinee or type arguments, it can be used to check against the type of the branches. An unmotivated **case** will be type checked. A **case** with a motive will have its type inferred.

We can confidently conjecture that the desired bidirectional properties hold.

Conjecture the data extension to the bidirectional surface language is type sound.

Conjecture the data extension to the bidirectional surface language is weakly annotatable from the data extension of the surface language.

This is a minimal (and somewhat crude) accounting of bidirectional data. It is possible to imagine syntactic sugar that doesn't require the \bar{N} , and $\bar{x} \Rightarrow$ the in case expression of the $\text{ty-case } \langle \rangle$ rule. In the dependent rule $\text{ty-case } \langle \rangle$ it is also possible to imagine some type constructor arguments being inferred. These features and

$$\begin{array}{c}
\frac{\overline{N} \Rightarrow \overline{N'} \quad \overline{m} \Rightarrow \overline{m'}}{\exists \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m_d \in \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}} \\
\frac{m_d \Rightarrow m'_d}{\text{case } \overline{N}, d\overline{m} \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \Rightarrow m_d [\overline{x} := \overline{N'}, \overline{y}_d := \overline{m'}]} \Rightarrow\text{-case-red} \\
\\
\frac{\overline{N} \Rightarrow \overline{N'} \quad m \Rightarrow m'}{\forall \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m_d \in \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} . m_d \Rightarrow m'_d} \\
\frac{\text{case } \overline{N}, m \langle \dots \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \Rightarrow \text{case } \overline{N'}, m' \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m'_{d'}} \right\}}{\text{case } \overline{N}, m \langle \dots \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \Rightarrow \text{case } \overline{N'}, m' \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m'_{d'}} \right\}} \Rightarrow\text{-case}<>\text{-red} \\
\\
\frac{\overline{N} \Rightarrow \overline{N'} \quad m \Rightarrow m' \quad M \Rightarrow M'}{\forall \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'} \in \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m_d} \right\} . m_{d'} \Rightarrow m'_{d'}} \\
\frac{\text{case } \overline{N}, m \langle \overline{x} \Rightarrow z : D \overline{x} \Rightarrow M \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m_d} \right\} \Rightarrow \text{case } \overline{N}, m' \langle \overline{x} \Rightarrow z : D \overline{x} \Rightarrow M' \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m'_d} \right\}}{\text{case } \overline{N}, m \langle \overline{x} \Rightarrow z : D \overline{x} \Rightarrow M \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m_d} \right\} \Rightarrow \text{case } \overline{N}, m' \langle \overline{x} \Rightarrow z : D \overline{x} \Rightarrow M' \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m'_d} \right\}} \Rightarrow\text{-case}<> \\
\\
\frac{}{\overline{D} \Rightarrow \overline{D}} \Rightarrow\text{-}D \\
\frac{}{\overline{d} \Rightarrow \overline{d}} \Rightarrow\text{-}d
\end{array}$$

Figure 7: Surface Language Data Reduction

$$\begin{array}{c}
\frac{}{\text{case } \overline{N}, n \langle \dots \rangle \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \rightsquigarrow \text{case } \overline{N}, n \left\{ \overline{\mid \Rightarrow \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}} \rightsquigarrow\text{-case}<> \\
\\
\frac{\exists \overline{x} \Rightarrow (d \overline{y}_d) \Rightarrow m_d \in \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}}{\text{case } \overline{V}, d\overline{v} \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \rightsquigarrow m_d [\overline{x} := \overline{V}, \overline{y}_d := \overline{v}]} \rightsquigarrow\text{-case-red} \\
\\
\frac{\overline{N} \rightsquigarrow \overline{N'}}{\text{case } \overline{N}, n \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \rightsquigarrow \text{case } \overline{N'}, n \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}} \\
\\
\frac{n \rightsquigarrow n'}{\text{case } \overline{V}, n \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \rightsquigarrow \text{case } \overline{V}, n' \left\{ \overline{\mid \overline{x} \Rightarrow (d' \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}}
\end{array}$$

Figure 8: Surface Language Data CBV

$$\begin{array}{c}
\frac{}{\Diamond \mathbf{Empty}} \text{Empty-ctx} \\
\\
\frac{\Gamma \mathbf{Empty} \quad \Gamma \vdash \text{data } D : \Delta \left\{ \overline{\mid d : \Theta_d \rightarrow D\overline{m}_d} \right\} \mathbf{ok}}{\Gamma, \text{data } D : \Delta \left\{ \overline{\mid d : \Theta_d \rightarrow D\overline{m}_d} \right\} \mathbf{Empty}} \text{Empty-ctx}
\end{array}$$

Figure 9: Surface Language Empty

$$\begin{array}{c}
\frac{\text{data } D \Delta \in \Gamma}{\Gamma \vdash D \overset{\rightarrow}{\vdash} \Delta \rightarrow *} \overrightarrow{ty}\text{-TCon} \\
\\
\frac{d : \Theta \rightarrow D\overline{m} \in \Gamma}{\Gamma \vdash d \overset{\rightarrow}{\vdash} \Theta \rightarrow D\overline{m}} \overrightarrow{ty}\text{-Con} \\
\\
\frac{\begin{array}{c} \text{data } D \Delta \{...\} \in \Gamma \\ \Gamma \vdash \overline{N} \overset{\leftarrow}{\vdash} \Delta \quad \Gamma \vdash n \overset{\leftarrow}{\vdash} D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\overline{x} \vdash M \overset{\leftarrow}{\vdash} \star \\ \forall d : \Theta \rightarrow D\overline{o} \in \Gamma. \quad \Gamma, \overline{y}_d : \Theta \vdash m_d [\overline{x} := \overline{o}] \overset{\leftarrow}{\vdash} M [\overline{x} := \overline{o}, z := d\overline{y}_d] \end{array}}{\Gamma \vdash \text{case } \overline{N}, n \langle \overline{x} \Rightarrow z : D\overline{x} \Rightarrow M \rangle \left\{ \overline{x} \Rightarrow (d\overline{y}_d) \Rightarrow m_d \right\} \overset{\rightarrow}{\vdash} M [\overline{x} := \overline{N}, z := n]} \overrightarrow{ty}\text{-case } <> \\
\\
\frac{\begin{array}{c} \text{data } D \Delta \{...\} \in \Gamma \\ \Gamma \vdash \overline{N} \overset{\leftarrow}{\vdash} \Delta \quad \Gamma \vdash n \overset{\rightarrow}{\vdash} D\overline{N} \\ \Gamma \vdash M \overset{\leftarrow}{\vdash} \star \\ \forall d : \Theta \rightarrow D\overline{o} \in \Gamma. \quad \Gamma, \overline{y}_d : \Theta \vdash m_d [\overline{x} := \overline{o}] \overset{\leftarrow}{\vdash} M \end{array}}{\Gamma \vdash \text{case } \overline{N}, n \left\{ \overline{x} \Rightarrow (d\overline{y}_d) \Rightarrow m_d \right\} \overset{\leftarrow}{\vdash} M} \overleftarrow{ty}\text{-case } <>
\end{array}$$

reparam o in $[\overline{x} := \overline{o}]$ also

Figure 10: Surface Language Bidirectional type checking

more will be subsumed by the dependent pattern matching of the next section, though this will complicate the meta-theory.

Part IV

Pattern Matching

Unfortunately, the direct eliminator style is cumbersome for programmers to deal with. For instance, Figure 11 shows how `Vec` data can be directly eliminated to extract the first element of a non-empty list in the definition of `head'`. The `head'` function needs to redirect unreachable vector inputs to a dummy type (`Unit`) and requires several copies of the same `A` variable that are not identified automatically by the eliminator described in the last section. The usual solution is to extend case elimination with **Pattern matching**.

Pattern matching is much more ergonomic than a direct eliminator `case`. In Figure 11, the `head` defined though pattern matching is simpler and clearer. Nested constructor matching is now possible. When pattern matching is extended to dependent types variables will be assigned their definitions as needed, and unreachable branches can be omitted from code. For this reason, pattern matching has been considered an “essential” feature for dependently typed languages since [Coq92] and is implemented in most popular systems, such as Agda and the user facing language of Coq.

Figure 12 shows the extensions to the surface language for data and pattern matching. Our case expression match a list of scrutinees, allowing us to be very precise about the typing of branches. Additionally this style allows for syntactic sugar for easy definitions of functions by cases. The syntax of the direct eliminator style `cases` of the last section was designed to be a special case of pattern matching.

Patterns correspond to a specific form of expression syntax. When an expression matches a pattern it will capture the relevant subexpressions as variables. For instance, the expression.

`Cons B true (S (S (S (Z)))) (Cons B false (S (S (Z)))) y'`
will match the patterns

- `Cons w x y z` with bindings $w = \mathbb{B}, x = \text{true}, y = 3, z = \text{Cons } \mathbb{B} \text{ false } (S (S (Z))) y'$
- `x` with bindings $x = \text{Cons } \mathbb{B} \text{ true } (S (S (S (Z)))) (Cons \mathbb{B} \text{ false } (S (S (Z)))) y'$
- `Cons - x - (Cons - y - -)` with bindings $x = \text{true}, y = \text{false}$


```

-- eliminator style
head' : (A : *) -> (n : Nat) ->
  Vec A (S n) ->
  A ;
head' A n v =
  case A, (S n), v <
    A' => n' => _ : Vec A' n' =>
      case n' < _ => * > {
        | (Z _) => Unit
        | (S _) => A'
      }
  >{
  | _ => (Z _) => (Nil _ _) => tt
  | _ => (S _) => (Cons _ a _ _) => a
  } ;

-- pattern match style
head : (A : *) -> (n : Nat) ->
  Vec A (S n) ->
  A ;
head A n v =
  case v < _ => A > {
  | (Cons _ a _ _) => a
  } ;

```

match the single eliminator syntax

clean when I get motive inference working

example def by cases

Figure 11: Eliminators vs. Pattern Matching

$m...$	$::=$	$...$	
		$\text{case } \overline{n}, \left\{ \overline{\text{pat} \Rightarrow m} \right\}$	data elim. without motive
		$\text{case } \overline{n}, \langle \overline{x} \Rightarrow M \rangle \left\{ \overline{\text{pat} \Rightarrow m} \right\}$	data elim. with motive
patterns,			
pat	$::=$	x	match a variable
		$(d \text{ pat})$	match a constructor

Figure 12: Surface Language Data

$$\begin{array}{c}
\frac{}{x \mathbf{Match}_{\{x:=m\}} m} \\
\frac{\frac{}{\overline{pat} \mathbf{Match}_{\sigma} \overline{m}}}{\overline{dpat} \mathbf{Match}_{\sigma} d\overline{m}} \\
\frac{pat' \mathbf{Match}_{\sigma'} n \quad \overline{pat} \mathbf{Match}_{\sigma} \overline{m}}{pat', \overline{pat} \mathbf{Match}_{\sigma' \cup \sigma} n, \overline{m}} \\
\frac{}{. \mathbf{Match}_{\emptyset} .}
\end{array}$$

Figure 13: Surface Language Match

When patterns are used in the case construct, the appropriate branch will reduce with the correct bindings in scope. Therefore the expression

`case Cons B true (S (S (S (Z)))) (Cons B false (S (S (Z)))) y' {Cons - x - (Cons - y - -) ⇒ x&y}` reduces to *false*.

The explicit rules for pattern matching are listed in Figure 13, where σ will hold a possibly empty set of assignments.

It is now easier for case branches to overlap, which could allow nondeterministic reduction. There are several plausible ways to handle this, such as requiring each branch to have independent patterns, or requiring patterns have the same behavior when they overlap [CPD14]. For the purposes of this thesis, we will use the programatic convention that the first matching pattern takes precedence. For example, we will be able to type check

`case 4 ⟨s : N ⇒ B⟩ {|S (S -) ⇒ True | - ⇒ False}`

and it will reduce to *True*.

While pattern matching is an extremely practical feature, typing these expressions tends to be messy. To implement dependently typed pattern matching, a procedure is needed to resolve the equational constraints that arise within each pattern, and to confirm the impossibility of unwritten branches.

Since arbitrary computation can be embedded in the arguments of a type constructor³, the equational constraints are undecidable in general. Any approach to constraint solving will have to be an approximation that performs well enough in practice. Usually this procedure usually takes the form of a first order unification.

talk about or formalize the more subtle inference in the actual system

5 First Order Unification

When type checking the branches of the a case expression, the patterns are interpreted as expressions under bindings for each variable used in the pattern. If these equations can be unified, then the brach will type-check under the variable assignments, with the additional typing information. For instance, the pattern

`Cons x (S y) 2 z`

could be checked against the type `Vec Nat w`

this implies the typings $x : *, y : Nat, (S y) : x, 2 : Nat, z : Vec x 2, (Cons x (S y) 2 z) : Vec Nat w$

which in turn imply the equalities

$x = Nat, w = 3$

This is a very simple example, in the worst case we may have equations in the form $m n = m' n'$ which are hard to solve directly (but may become easy to solve if assignment of $m = \lambda x.x$, and $m' = \lambda - .0$ are discovered).

One advantage of the first order unification approach is that if the algorithm succeeds, it will succeed with a unique, most general solution. Since assignments are maximal, we are sure that a unified pattern will still be able to match any well typed syntax.

A simplified version of a typical unification procedure is listed in Figure 14. Several variations are explored in [CD18]. Unification is not guaranteed to terminate since it relies on definitional equalities, which are undecidable in the surface language. The unification procedure does not exclude the possibly cyclic assignments that could occur, such as $x = S x$.

³At least in a full spectrum theory, such as the one we study here.

$$\begin{array}{c}
\overline{U(\emptyset, \emptyset)} \\
\\
\frac{U(E, a) \quad m \equiv m'}{U(\{m \sim m'\} \cup E, a)} \\
\\
\frac{U(E[x := m], a[x := m])}{U(\{x \sim m\} \cup E, \{a, x := m\})} \\
\\
\frac{U(E[x := m], a[x := m])}{U(\{m \sim x\} \cup E, a \cup \{x := m\})} \\
\\
\frac{U(\overline{m} \sim \overline{m'} \cup E, a) \quad n \equiv d\overline{m} \quad n' \equiv d\overline{m'}}{U(\{n \sim n'\} \cup E, a)} \\
\\
\frac{U(\overline{m} \sim \overline{m'} \cup E, a) \quad N \equiv D\overline{m} \quad N' \equiv D\overline{m'}}{U(\{N \sim N'\} \cup E, a)}
\end{array}$$

Figure 14: Surface Language Unification

After the branches have type checked we should make sure that they are exhaustive, such that every possible branch will be covered. There are several possible strategies. In general it is undecidable whether any given pattern is impossible or not, so a practical approximation must be chosen. At least programmers have the ability to manually include non-obviously unreachable branches and prove their unreachability, (or direct those branches to dummy outputs). Though there is a real risk that the unification procedure gets stuck in ways that are not clear to the programmer, and a clean error message may be very difficult.

Usually a branch is characterized as unreachable if a contradiction is found in the unification procedure. But that pattern or patterns must still be generated, given the explicit branches the programmer introduced. There is no clear “best way” to do this since a more fine division of patterns may allow enough additional definitional information to show unsatisfiability, while a more coarse division of patterns will be more efficient. Agda uses a tree branching approach, that is efficient, but generates coarse patterns. The implementation of the language in this thesis generates patterns by a system of complements, this system seems slightly easier to implement, more uniform, and generates a much finer system of patterns than the case trees used in Agda. However this approach is exponentially less performant than Agda in the worst case.

The bidirectional system can be extended with pattern matching with rules that look like

$$\begin{array}{c}
\frac{\begin{array}{c} \Gamma \vdash \overline{n} \dot{\Rightarrow} \Delta \\ \Gamma, \Delta \vdash M \dot{\Leftarrow} \star \\ \forall i (\Gamma \vdash \overline{pat}_i :_E ? \Delta \quad U(E, \sigma) \quad \sigma(\Gamma, |\overline{pat}_i|) \vdash \sigma m \dot{\Leftarrow} \sigma(M [\Delta := \overline{pat}_i])) \\ \Gamma \vdash \overline{pat} : \Delta \text{ complete} \end{array}}{\Gamma \vdash \text{case } \overline{n}, \langle \Delta ? \Rightarrow M \rangle \left\{ \overline{pat} \Rightarrow m \right\} \dot{\Rightarrow} M [\Delta ? := \overline{n}]} \\
\\
\frac{\begin{array}{c} \Gamma \vdash \overline{n} \dot{\Rightarrow} \Delta \\ \forall i (\Gamma \vdash \overline{pat}_i :_E ? \Delta \quad U(E, \sigma) \quad \sigma(\Gamma, |\overline{pat}_i|) \vdash \sigma m \dot{\Leftarrow} \sigma(M)) \\ \Gamma \vdash \overline{pat} : \Delta \text{ complete} \end{array}}{\Gamma \vdash \text{case } \overline{n}, \left\{ \overline{pat} \Rightarrow m \right\} \dot{\Leftarrow} M}
\end{array}$$

where $\Gamma \vdash \overline{pat} :_E ? \Delta$ is shorthand for a set of equations that allow a list of patterns to type check under Δ . and $\Gamma \vdash \overline{pat} : \Delta \text{ complete}$ is shorthand for the exhaustiveness check.

Conjecture There exists a suitable⁴ extension to the surface language TAS that supports pattern matching style elimination

Conjecture The bidirectional extension listed here is weakly annotatable with that extension to the surface language.

⁴supporting at least subject reduction, type soundness, and regularity

Additionally, it makes sense to allow some additional type annotations in the motive and for these annotations to switch the the type inference of the scrutinee into a type-check. The implementation includes this along with a simple syntax for modules, and even mutually defined data types. For simplicity these have been excluded from the formal presentation.

6 Discussion

Pattern matching seems simple, but is a surprisingly subtle.

Even without dependent types, pattern matching is a strange feature. How important is it that patterns correspond exactly to a subset expression syntax? What about capture annotations or side conditions? Restricting patterns to constructors and variable means that it is hard to encapsulate functionality, a problem noticed as early as [Wad87]. This has lead to making pattern behavior override-able in Scala via Extractor Objects. An extension in GHC allows some computations to happen within a pattern match via the *ViewPatterns* extension. It seems unreasonable to extend patterns to arbitrary computation (though this is allowed in the Curry language⁵ as a syntax for its logical programming features).

In the presence of full-spectrum dependent types, the perspective dramatically shifts. Any terminating typing procedure will necessarily exclude some type-able patterns and be unable to exclude some unreachable branches. Even though only data values are considered, dependent patterns are already attacking a much more difficult problem than in the non-dependent case. It may make sense to extend the notion of pattern matching to include other useful but difficult features. To some extent this is similar to the with syntax of [MM04].

, Agda and Idris make pattern matching more powerful using with syntax that allows further pattern based branching by attaching a computation to a branch. This is justified as syntactic sugar that corresponds to helper functions that can be appropriately elaborated and typed. The language described in this thesis does not use the with side condition since nested case expressions carry the same computational behavior, and the elaboration to the cast language will allow possibly questionable typing anyway.

More aggressive choices should be explored beyond the with construct. In principle it seems that dependent case expressions could be extended with relevant proof search, arbitrary computation or some amount of constraint solving, without being any theoretically worse than usual first order unification.

Discuss how stratified type systems like ATS handle things (additional equational information)

The details of pattern matching change the logical character of the system [CD18]. Since non-termination is allowed in the language described here, the logical issues that arise from pattern are less of a concern then the immediate logical unsoundness that was discussed in chapter 2. However it is worth noting that pattern matching as described here validates axiom k and thus appears unsuitable for Hott or CTT developments.

This chapter has glossed over the definitional behavior of cases, since we plan to sidestep the issue with the cast language. It is worth noting that their are several ways to set up the definitional reductions. Agda style case trees may result in unpredictable definitional equalities (in so far as definitional behavior is ever predictable) [CPD14]. [CPD14] advocates for a more conservative approach that makes function definitions by cases definitional (but shifts the difficulties to overlapping branches and does not allow the “first match” behavior programmers are used to). Another extreme would be to only allow reductions at fully computed scrutinee values, as in trellies work [SCA⁺12]. Alternatively a partial reduction is possible, such that branches are eliminated as they are found unreachable and substitutions made as they are available. This last approach is experimentally implemented for the language defined here.

This complicates the simple story from chapter 2, where the bidirectional system made the TAS system checkable by only adding annotations (and having annotatability). We have only conjectured the existence of a suitable TAS system for pattern matching. If the definitional equality that feeds the TAS is generated by a system of reductions, any of the reduction strategies will generate a different TAS with subtly different characteristics. For instance, insisting on a call-by-value case reduction will leave many equivalent computations unassociated. If the TAS system uses partial reductions it will need to inspect the constructors of the scrutinee in order to preserve typing when reduction eliminates branches. Agda style reductions need to extend syntax under reduction to account for side conditions.

Ideally the typing rule for pattern matching case expression in the TAS should not use the notion of unification at all. Instead the rule should characterize the behavior that is required directly and formally⁶. An ideal rule might look like

⁵<https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/>

⁶[Coq92] has a good informal description

$$\begin{array}{c}
\Gamma \vdash \bar{n} : \Delta' \quad (\text{scrutinees type check}) \\
\Gamma, \bar{x} : \Delta' \vdash M : \star \quad (\text{motive exists and is well formed}) \\
\forall i. ? \quad (\text{every branch is well typed over all possible instantiations}) \\
? \quad (\text{all scrutinees are handled}) \\
\hline
\Gamma \vdash \text{case } \bar{n}, \left\{ \overline{\text{pat} \Rightarrow_i m_i} \right\} : M [\bar{x} := \bar{n}] \quad \dots
\end{array}$$

last condition is optional if you're willing to modify type soundness to allow pattern match errors (again, they are no worse than the non-termination already allowed, and better behaved).

Part V

Related work

7 Systems with Data

Many systems that target data only formalize a representative collection of data types, expecting the reader to be able to generalize the scheme. This data usually covers Nats (for recursion) dependent pairs (for type constructor arguments) and unit to end a chain of dependent pairs. For example, Martin Lof's original paper treated data this way, and is still a common approach to data (for instance in [JZSW10]).

Martin Lof generalized the notion of data to W types of well founded trees and this still serves as a theoretical justification for data.

Unified Type Theory (UTT)[Luo90, Luo94] is an extension to ECC that specifies a scheme to define strictly positive data types by way of a logical framework defined in MLTT. This scheme generates primitive recursors for schematized data, and does not inherently support pattern matching.

The Calculus of Inductive Constructions (CiC) is an extension to the calculus of constructions that includes a system of first class data. It was first presented in [Luo90], but the most complete up to date formulation is maintained as part of the Coq manual ⁷. A bidirectional account of CIC is given in [LB21], though it uses a different style of bidirectionality then discussed here to maintain compatibility with the existing Coq system.

confirm, c

confirm, c

finish read

"Inductive
the system
properties

CTT, higher inductive types, quotient types

8 Pattern matching

Clean up this section

Early work by Coq92 [Coq92]
with a lot of follow up from McBride [MM04]
reiterated in [Nor07]

A tutorial implementation of dynamic pattern unification Adam Gundry and Conor McBride (2012)
<http://adam.gundry.co.uk/pub/pattern-unify/> (this links give you the choice to read a more detailed chapter of Adam Gundry's thesis instead)

with substantial follow up in [CD18]

<https://research.chalmers.se/en/publication/519011> ?

https://sozeau.gitlabpages.inria.fr/www/research/publications/Equations:_A_Dependent_Pattern-Matching_Compiler.pdf ?

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.1405&rep=rep1&type=pdf> ?

talk about normalization

<https://popl19.sigplan.org/details/POPL-2019-Research-Papers/33/Higher-Inductive-Types-in-Cubical-Computational-Type-Theory>

⁷<https://coq.github.io/doc/v8.9/refman/language/cic.html>

References

- [CD18] Jesper Cockx and Dominique Devriese. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming*, 28:e12, 2018.
- [Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83. Citeseer, 1992.
- [CPD14] Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns. In Zhong Shao, editor, *Programming Languages and Systems*, pages 87–106, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.
- [JZSW10] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. *SIGPLAN Not.*, 45(1):275–286, January 2010.
- [LB21] Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
- [Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. 1994.
- [MM04] Conor McBride and James Mckinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [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.
- [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.
- [Wad87] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. Association for Computing Machinery.

Part VI

TODO

- differentiate identifiers with font, font stuff with code
 - Make identifiers consistent with chapter 2, and locations in chapter 3
- definition package?
- syntax highlighting
- additional equations al la ATS +trellis
- allowing equality matching (based on typeclass?)

Todo list

“not handle data types in a rigorous way.”	1
review if there are more abbreviations around?	3
give explicit example, note the motive	3
give explicit of data eliminations that uses these annotations	3
motive should not need to insist on the type info of the binder? grey out? Grey out things that are surface syntax but not needed for theory?	3
Cite De Bruijn for telescopes	4
move this down to the typing section?	4
say with rep example	4
who cam up with this first? Martin Lof?	4
suspect this also hinges on regularity, which should be addressed more directly	5
extend cbv over lists	6
extend step over lists	6
symbolically	6
reparam o in $[\bar{x} := \bar{o}]$ also	8
Example of why it was needed	8
match the single eliminator syntax	9
clean when I get motive inference working	9
example def by cases	9
talk about or formalize the more subtle inference in the actual system	10
as a threat to soundness this should be corrected?	10
more detail	11
ECHO VIEW from the left to	12
Inductive families	12
review	12
lots of prior work https://gitlab.haskell.org/ghc/ghc/-/wikis/view-patterns	12
discuss stuck state of unification explicitly, $x = f a$ vs. $f a = g b$, and how it for instance makes it so you can't prove transitivity of $ID fa gb, ID gb hc \rightarrow ID fa hc$. also calls into quesiton preservation entirely.	12
Epigram	12
Discus how stratified type systems like ATS handle things (additional equational information)	12
review	12
confirm	12
confirm, cite	13
confirm, cite	13
finish reading this	13
“Inductive definitions in the system Coq rules and properties” confirm	13
CTT, higher inductive types, quotient types	13
Clean up this section	13
A tutorial implementation of dynamic pattern unification Adam Gundry and Conor McBride (2012) http://adam.gundry.co.uk/publications/unify/ (this links give you the choice to read a more detailed chapter of Adam Gundry's thesis instead)	13
https://research.chalmers.se/en/publication/519011 ?	13
https://sozeau.gitlabpages.inria.fr/www/research/publications/Equations:_A_Dependent_Pattern-Matching_Compiler.pdf ?	13
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.1405&rep=rep1&type=pdf ?	13
talk about normalization	13

9 notes

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.

10 unused

$$\begin{array}{c}
 \frac{\Gamma \vdash \Delta \overleftarrow{\mathbf{ok}}}{\Gamma \vdash \mathbf{data } D \Delta \overleftarrow{\mathbf{ok}}} \\
 \hline
 \Gamma \vdash \mathbf{data } D \Delta \quad \forall d. \Gamma, \mathbf{data } D \Delta \vdash \Theta_d \quad \forall d. \Gamma, \mathbf{data } D \Delta, \Theta_d \vdash \overline{m}_d : \Delta \\
 \hline
 \Gamma \vdash \mathbf{data } D : \Delta \left\{ \overline{d} : \Theta_d \rightarrow D \overline{m}_d \right\}
 \end{array}$$

...