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

Mark Lemay

December 9, 2021

for the purposes of this draft separating the "conventional stuff" from the new stuff. this chapter is the conventional stuff. should it be recombined?

## 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 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 essential for organizing readable programs, and are a degenerate forms of dependent data. Dependent data can represent mathematical predicates like equality. 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 church encoded data form Chapter 2 could handle all of these cases. However church encoded data is inconvenient in practice. Since "ease of use" is the overriding concern for the language developed in this thesis, Church encoding are an unrealistic way yo handle 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 standard data types are defined in Figure 1. For example, the data type `Nat` is 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 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 easy 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 II
# Data and Direct Elimination

How should a program observe data? Since the term of a given type can only be created from one of the constructors from its definition, we can completely handle a data expression if each possible constructor is accounted for. For instance, `Nat` has the two constructors `Z` and `S` (which holds the preceding number), so the expression $\mathsf{case}\, n\ \{\mid Z \Rightarrow Z \mid S\, x \Rightarrow x\}$ 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 above 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[1]. In general, motive annotations will be treated like the typing

give expli
the motiv

give expli
inations t
annotatio

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

data Nat : * {
| Z : Nat
| S : Nat -> Nat
};
three : Nat;
three = S (S (S Z)));
-- Syntactic sugar expands number literals
-- into their unary representation.

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

| | | | |
|---|---|---|---|
| telescope, | | | |
| $\Delta, \Theta$ | $::=$ | $.$ | empty telescope |
| | $\|$ | $x : M,\ \Delta$ | extend telescope |
| list of $O$, separated with $s$ | | | |
| $\overline{sO}$,$\overline{Os}$ | $::=$ | $s$ | empty list |
| | $\|$ | $sOs\overline{O}$ | extend list |
| data type identifier, | | | |
| $D$ | | | |
| data constructor identifier, | | | |
| $d$ | | | |
| contexts, | | | |
| $\Gamma$ | $::=$ | $...$ | |
| | $\|$ | $\Gamma, \mathsf{data}\, D\ :\ \Delta \to \star\ \left\{\overline{\|\, d\ :\ \Theta \to D\overline{m}}\right\}$ | data def extension |
| | $\|$ | $\Gamma, \mathsf{data}\, D\ :\ \Delta \to \star$ | abstract data extension |
| $m, n, M, N$ | $::=$ | $...$ | |
| | $\|$ | $D$ | type cons. |
| | $\|$ | $d$ | data cons. |
| | $\|$ | $\mathsf{case}\, \overline{N}$,$n\ \left\{\overline{\|\overline{x \Rightarrow}(d\,\overline{y}) \Rightarrow m}\right\}$ | data elim. without motive |
| | $\|$ | $\mathsf{case}\, \overline{N}$,$n\ \langle \overline{x \Rightarrow} y : D\,\overline{x} \Rightarrow M \rangle\ \left\{\overline{\|\overline{x \Rightarrow}(d\,\overline{y}) \Rightarrow m}\right\}$ | data elim. with motive |
| values, | | | |
| $v$ | $::=$ | $...$ | |
| | $\|$ | $D\,\overline{v}$ | |
| | $\|$ | $d\,\overline{v}$ | |

shorthands!

Figure 2: Surface Language Data

annotations in Chapter 2, in that the TAS will confirm that any given motive is correct 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 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**[2] 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,

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

Additionally we define telescopes, which generalize zero or more typed bindings. This allows us to specify data definitions in a clean way. 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 $\mathtt{Nat}, 2, 2, refl\,\mathtt{Nat}\,2$ type checks against $(x : \star), (y : \mathtt{Nat}), (z : \mathtt{Nat}), (- : Id\ x\,y\,z)$. 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 $\Delta = (x : \star), (y : \mathtt{Nat}), (z : \mathtt{Nat}), (- : Id\ x\,y\,z)$ then writing $f : \Delta \to \mathtt{Nat}$ will be short hand for $f : (x : \star) \to (y : \mathtt{Nat}) \to (z : \mathtt{Nat}) \to Id\ x\,y\,z \to \mathtt{Nat}$.

move this
typing sec

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..

say with r

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

[1] This slightly awkward case eliminator syntax is designed to be forward compatible with the pattern matching system defined in the rest of this chapter, which in turn allows function definition by cases.

[2] also **discriminee**

# 1 Incomplete Eliminations

Consider the match

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

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 cases to be exhaustive to type check. This is in contrast to most programming languages, which do allow incomplete patterns, though usually a warning is given, and a runtime error if scrutinee cannot be matched.

This thesis already has a system for handling warnings and errors in 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.

# 2 (non) Strict Positivity

A more subtle concern is posed by data definitions that are not strictly positive.

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

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

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

This data 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 data where this is a possibility. However this would block useful constructions like higher order abstract syntax. Additionally since non-termination is already allowed in the surface TAS, we will not restrict the surface language to strictly positive date.

h this
f?

# 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 3, and are standard. Telescopes are well formed when they extend the context in a well formed way. Lists of expressions can be said to have the type of the telescope if every expression has the appropriate type.

Cite De Bruijn for telescopes

Data definitions can be added to contexts if all of their constituents are well typed and **ok**. The rules are listed in 4. 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 the module system that adds data to context, it is taken for granted that any well formed data in context is fine. This presentation of data largely follows [SCA$^+$12].

The type assignment system must be extended with the rules in 5. These rules make use of several convenient shorthands: $\mathsf{data}\, D\, \Delta \in \Gamma$ and $d : \Theta \rightarrow D\overline{m} \in \Gamma$ 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-$\mathsf{case}$ <> 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-$\mathsf{case}$

$$\frac{\Gamma\,\mathbf{ok}}{\Gamma \vdash .\,\mathbf{ok}}\ \text{ok-Tel-empty}$$

$$\frac{\Gamma \vdash M : \star \quad \Gamma, x : M \vdash \Delta\,\mathbf{ok}}{\Gamma \vdash x : M, \Delta\,\mathbf{ok}}\ \text{ok-Tel-ext}$$

$$\frac{\Gamma\,\mathbf{ok}}{\Gamma \vdash \Diamond : .}\ \text{ty-ls-empty}$$

$$\frac{\Gamma, x : M \vdash \Delta \quad \Gamma \vdash m : M \quad \Gamma \vdash \overline{n,} [x \coloneqq m] : \Delta\,[x \coloneqq m]}{\Gamma \vdash m\overline{,n}\ :\ (x : M)\,, \Delta}\ \text{ty-ls-ext}$$

Figure 3: Meta rules

$$\frac{\Gamma \vdash \Delta\,\mathbf{ok}}{\Gamma \vdash \mathsf{data}\,D\,\Delta\,\mathbf{ok}}\ \mathbf{ok}\text{-abs-data}$$

$$\frac{\Gamma \vdash \mathsf{data}\,D\,\Delta\,\mathbf{ok}}{\Gamma, \mathsf{data}\,D\,\Delta\,\mathbf{ok}}\ \mathbf{ok}\text{-data-ext}$$

$$\frac{\Gamma \vdash \mathsf{data}\,D\,\Delta\,\mathbf{ok} \quad \forall d.\Gamma, \mathsf{data}\,D\,\Delta \vdash \mathit{\Theta}_d\,\mathbf{ok} \quad \forall d.\,\Gamma, \mathsf{data}\,D\,\Delta, \mathit{\Theta}_d \vdash \overline{m}_d : \Delta}{\Gamma \vdash \mathsf{data}\,D\ :\ \Delta \left\{ \overline{\mid d\ :\ \mathit{\Theta}_d \to D\overline{m}_d} \right\}\ \mathbf{ok}}\ \mathbf{ok}\text{-data}$$

$$\frac{\Gamma \vdash \mathsf{data}\,D\ :\ \Delta \left\{ \overline{\mid d\ :\ \mathit{\Theta} \to D\overline{m}} \right\}\ \mathbf{ok}}{\Gamma, \mathsf{data}\,D\ :\ \Delta \left\{ \overline{\mid d\ :\ \mathit{\Theta} \to D\overline{m}} \right\}\ \mathbf{ok}}\ \mathbf{ok}\text{-data-ext}$$

Figure 4: Surface Language Data ok

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 $\overline{x} : \Delta$.

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

define these $\in$ more?

Extensions to the parallel reduction rules are listed in 6. They follow the scheme of parallel reductions laid out in chapter 2. The $\Rrightarrow$-case-red rule reduces a case expression by choosing the appropriate branch[3] . The $\Rrightarrow$-case<>-red rule removes the motive annotation, much like the annotation rule in Chapter 2. The rules $\Rrightarrow$-case<>, $\Rrightarrow$-$D$, and $\Rrightarrow$-$d$ keep the $\Rrightarrow$ relation reflexive.

extend reductions over lists

We are now in a position to select a sub relation of $\Rrightarrow$ reductions that will be used to characterize call-by-value evaluation. This relation could be used used to prove type safety, and is close to the reduction used in the implementation. The rules are listed in 7.

extend step over lists

Finally we characterize what data is well formed, and what it means for a context to be empty in the presence of data in 8.

While a similar system is fully explored in [SCA+12], we will not prove the type soundness of the system here. For clarity we will list it as a conjecture.

**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 $\Gamma\,\mathbf{Empty}$, $\Gamma \vdash m : M$, then $m$ is a value, or $m \rightsquigarrow m'$ .

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

[3] Also called Iota reduction

$$\frac{\Gamma\,\mathbf{ok} \quad \mathsf{data}\, D\,\Delta \in \Gamma}{\Gamma \vdash D \,:\, \Delta \to \star}\ \text{ty-TCon}$$

$$\frac{\Gamma\,\mathbf{ok} \quad d \,:\, \mathit{\Theta} \to D\overline{m} \in \Gamma}{\Gamma \vdash d \,:\, \mathit{\Theta} \to D\overline{m}}\ \text{ty-Con}$$

$$\frac{\begin{array}{c} \mathsf{data}\, D\,\Delta\,\{...\} \in \Gamma \\ \Gamma \vdash n : D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\,\overline{x} \vdash M : \star \\ \forall\, d \,:\, \mathit{\Theta} \to D\overline{o} \in \Gamma. \quad \Gamma, \overline{y}_d : \mathit{\Theta} \vdash m_d\,[\overline{x} := \overline{o}] : M\,[\overline{x} := \overline{o}, z := d\,\overline{y}_d] \end{array}}{\begin{array}{c} \Gamma \vdash \mathsf{case}\,\overline{N}{,}n\ \langle \overline{x \Rrightarrow} z : D\,\overline{x} \Rightarrow M\rangle \left\{\overline{|\,\overline{x \Rrightarrow} d\overline{y}_d\ \Rightarrow m_d}\right\} \\ : M\left[\overline{x} := \overline{N}, z := n\right] \end{array}}\ \text{ty-}\,\mathsf{case} <>$$

no duplicate branches

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

$$\frac{\begin{array}{c} \mathsf{data}\, D\,\Delta\,\{...\} \in \Gamma \\ \Gamma \vdash n : D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\,\overline{x} \vdash M : \star \\ \forall\, d \,:\, \mathit{\Theta} \to D\overline{o} \in \Gamma. \quad \Gamma, \overline{y}_d : \mathit{\Theta} \vdash m_d\,[\overline{x} := \overline{o}] : M\,[\overline{x} := \overline{o}, z := d\,\overline{y}_d] \end{array}}{\begin{array}{c} \Gamma \vdash \mathsf{case}\,\overline{N}{,}n\ \left\{\overline{|\,\overline{x \Rrightarrow} d\overline{y}_d\ \Rightarrow m_d}\right\} \\ : M\left[\overline{x} := \overline{N}, z := n\right] \end{array}}\ \text{ty-}\,\mathsf{case}$$

Figure 5: Surface Language Data Typing

$$\frac{\begin{array}{c} \overline{N} \Rrightarrow \overline{N'} \quad \overline{m} \Rrightarrow \overline{m'} \\ \exists \overline{x \Rrightarrow}(d\,\overline{y}_d) \Rightarrow m_d \in \left\{\overline{|\,\overline{x \Rrightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \\ m_d \Rrightarrow m'_d \end{array}}{\mathsf{case}\,\overline{N}, d\overline{m}\ \left\{\overline{|\,\overline{x \Rrightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \Rrightarrow m_d\left[\overline{x} := \overline{N'}, \overline{y}_d := \overline{m'}\right]}\ \Rrightarrow\text{-}\mathsf{case}\text{-red}$$

$$\frac{\begin{array}{c} \overline{N} \Rrightarrow \overline{N'} \quad m \Rrightarrow m' \\ \forall \overline{x \Rrightarrow}(d\,\overline{y}_d) \Rightarrow m_d \in \left\{\overline{|\,\overline{\Rrightarrow x} \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}. \, m_d \Rrightarrow m'_d \end{array}}{\mathsf{case}\,\overline{N}, m\ \langle ...\rangle \left\{\overline{|\,\overline{\Rrightarrow x} \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \Rrightarrow \mathsf{case}\,\overline{N'}, m' \left\{\overline{|\,\overline{\Rrightarrow x} \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m'_{d'}}\right\}}\ \Rrightarrow\text{-}\mathsf{case}{<>}\text{-red}$$

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

$$\frac{}{D \Rrightarrow D}\ \Rrightarrow\text{-}D$$

$$\frac{}{d \Rrightarrow d}\ \Rrightarrow\text{-}d$$

Figure 6: Surface Language Data Reduction

$$\dfrac{}{\mathsf{case}\,\overline{N}, n\ \langle ...\rangle \left\{\overline{|\,\overline{\Rightarrow x} \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \rightsquigarrow \mathsf{case}\,\overline{N}, n \left\{\overline{|\,\overline{\Rightarrow x} \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}}\ \rightsquigarrow\text{-}\mathsf{case}<>$$

$$\dfrac{\exists \overline{x \Rightarrow}(d\,\overline{y}_d) \Rightarrow m_d \in \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}}{\mathsf{case}\,\overline{V}, d\overline{v}\ \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \rightsquigarrow m_d\left[\overline{x} := \overline{V}, \overline{y}_d := \overline{v}\right]}\ \rightsquigarrow\text{-}\mathsf{case}\text{-red}$$

$$\dfrac{\overline{N} \rightsquigarrow \overline{N'}}{\mathsf{case}\,\overline{N}, n\ \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \rightsquigarrow \mathsf{case}\,\overline{N'}, n\ \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}}$$

$$\dfrac{n \rightsquigarrow n'}{\mathsf{case}\,\overline{V}, n\ \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \rightsquigarrow \mathsf{case}\,\overline{V}, n'\ \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}}$$

Figure 7: Surface Language Data CBV

$$\dfrac{}{\lozenge\,\mathbf{Empty}}\ \text{Empty-ctx}$$

$$\dfrac{\Gamma\,\mathbf{Empty} \quad \Gamma \vdash \mathsf{data}\,D\ :\ \Delta\left\{\overline{|\,d\ :\ \Theta_d \to D\overline{m}_d}\right\}\ \mathbf{ok}}{\Gamma, \mathsf{data}\,D\ :\ \Delta\left\{\overline{|\,d\ :\ \Theta_d \to D\overline{m}_d}\right\}\ \mathbf{Empty}}\ \text{Empty-ctx}$$

Figure 8: Surface Language Empty

## 4 Bidirectional extension

A bidirectional interpretation exists over the type assignment rules listed above. An outline of these rules is in 8. 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.

As noted in [DK21], in general the bidirectional rules are open to some interpretation. The dependent case simplifies the analysis slightly by having an unmotivated case be a check, and having a motivated case be an infer.

This is a minimal (and somewhat crude) accounting of bidirectional data. It is possible to imagine syntactic sugar that doesn't require the $\overline{N}$, and $\overline{x \Rightarrow}$ the in case expression of the $\overleftarrow{ty}$-$\mathsf{case}$ <> rule. In the dependent rule $\overrightarrow{ty}$-$\mathsf{case}$ <> it is also also possible to imagine some type constructor arguments being inferred. These features and more will be subsumed by the dependent pattern matching of the next section, though this will complicate the meta-theory.

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.

# Part III
# Pattern Matching

explicitly talk about how pattern matching subsumes the previous sections

Unfortunately, the eliminator style is cumbersome for programmers to deal with. For instance, in figure 10 shows how `Vec` data can be directly eliminated in the definition of `head′`. The `head′` function needs to redirect impossible inputs to a dummy type 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 nested constructor matching is now possible. When pattern matching is extended to dependent types variables will be assigned their definitions as

$$\frac{\mathsf{data}\, D\, \Delta \in \Gamma}{\Gamma \vdash D \overset{\rightarrow}{:} \Delta \to *}$$

$$\frac{d \;:\; \mathit{\Theta} \to D\overline{m} \in \Gamma}{\Gamma \vdash d \overset{\rightarrow}{:} \mathit{\Theta} \to D\overline{m}}$$

names

$$\frac{\begin{array}{c}\mathsf{data}\, D\, \Delta\, \{...\} \in \Gamma \\ \Gamma \vdash \overline{N} \overset{\leftarrow}{:} \Delta \quad \Gamma \vdash n \overset{\rightarrow}{:} D\overline{N} \\ \Gamma \vdash M \overset{\leftarrow}{:} \star \\ \forall\, d \;:\; \mathit{\Theta} \to D\overline{o} \in \Gamma. \quad \Gamma, \overline{y}_d : \mathit{\Theta} \vdash m_d\, [\overline{x} \coloneqq \overline{o}] \overset{\leftarrow}{:} M \end{array}}{\Gamma \vdash \mathsf{case}\, \overline{N}{,}n\, \left\{ \overline{| \overline{x \Rightarrow} \, (d\, \overline{y}_d) \Rightarrow m_d} \right\} \overset{\leftarrow}{:} M}\; \overleftarrow{ty}\text{-}\,\mathsf{case} <>$$

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

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

Figure 9: Surface Language Empty

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 Agda and the user facing language of Coq.

Figure 11 shows the extensions to the surface language for data and pattern matching. Our case eliminators match a tuple of expressions, 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 eliminator style cases 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 choose the appropriate branch to reduce. For instance, the expession

$Cons\, \mathbb{B}\, true\ (S\, (S\, (S\, (Z))))\ (Cons\, \mathbb{B}\, false\ (S\, (S\, (Z)))\ y')$

will match the patterns

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

Therefore the expression

$\mathsf{case}\, Cons\, \mathbb{B}\, true\ (S\, (S\, (S\, (Z))))\ (Cons\, \mathbb{B}\, false\ (S\, (S\, (Z)))\ y')\, \{Cons\ -\ x\ -\ (Cons\ -\ y\ -\ -) \Rightarrow x \& y\}$ r

educes to $false$.

The explicit rules for pattern matching are listed in 12, where $\sigma$ will hold a possibly empty set of assignments.

It is now possible 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 has precedence. For example, we will be able to type check

$$\mathsf{case}\, 4\ \langle s : \mathbb{N} \Rightarrow \mathbb{B} \rangle\, \{ | S\, (S\, -) \Rightarrow True\, |\, - \Rightarrow False \}$$

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.

```
-- 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
  } ;
```

clean when I get motive inference working

example def by cases

Figure 10: Eliminators vs. Pattern Matching

$$\begin{array}{llll} m\ldots & ::= & \ldots & \\ & | & \mathsf{case}\,\overline{n},\ \left\{\overline{|\,pat \Rightarrow m}\right\} & \text{data elim. without motive} \\ & | & \mathsf{case}\,\overline{n},\ \langle \overline{x \Rightarrow} M\rangle \left\{\overline{|\,pat \Rightarrow m}\right\} & \text{data elim. with motive} \end{array}$$

patterns,

$$\begin{array}{llll} pat & ::= & x & \text{match a variable} \\ & | & (d\,\overline{pat}) & \text{match a constructor} \end{array}$$

Figure 11: Surface Language Data

$$\frac{}{x \sim_{\{x:=m\}} m}\ldots$$

$$\frac{\overline{pat} \sim_{\sigma} \overline{m}}{d\overline{pat} \sim_{\sigma} d\overline{m}}\ldots$$

$$\frac{pat' \sim_{\sigma} n \quad \overline{pat} \sim_{\sigma'} \overline{m}}{pat', \overline{pat} \sim_{\sigma \cup \sigma'} n, \overline{m}}\ldots$$

$$\frac{}{. \sim_{\emptyset} .}\ldots$$

Make consistent with other section

Figure 12: Surface Language Match

$$\frac{}{U\left(\emptyset,\emptyset\right)}$$

$$\frac{U\left(E,a\right)\quad m\equiv m'}{U\left(\left\{m\sim m'\right\}\cup E,a\right)}$$

$$\frac{U\left(E\left[x\coloneqq m\right],a\left[x\coloneqq m\right]\right)}{U\left(\left\{x\sim m\right\}\cup E,\left\{a,x\coloneqq m\right\}\right)}$$

$$\frac{U\left(E\left[x\coloneqq m\right],a\left[x\coloneqq m\right]\right)}{U\left(\left\{m\sim x\right\}\cup E,a\cup\left\{x\coloneqq m\right\}\right)}$$

$$\frac{U\left(\overline{m}\sim\overline{m'}\cup E,a\right)\quad n\equiv d\overline{m}\quad n'\equiv d\overline{m'}}{U\left(\left\{n\sim n'\right\}\cup E,a\right)}$$

$$\frac{U\left(\overline{m}\sim\overline{m'}\cup E,a\right)\quad N\equiv D\overline{m}\quad N'\equiv D\overline{m'}}{U\left(\left\{N\sim N'\right\}\cup E,a\right)}$$

Figure 13: Surface Language Unification

Since arbitrary computation can be embedded in the arguments of a type constructor (in a full spectrum theory), the equational constraints are undecidable in general. Any approach to constraint solving will have to be an approximation that performs well enough in practice. In practice this procedure usually takes the form of a first order unification. Several variations are explored in [CD18].

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

$x=Nat,w=3$

Note that 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 13. 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 $x=S\,x$.

undness
rrected?

After the branches have type checked we should makes sure that they are exhaustive, such that every possible branch will be covered. There are several possible strategies. In general it is undecidable wether 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 devision of patterns may allow enough additional definitional information to show unsatisfiability, while a more coarse devision of patterns will be more efficient. Agda uses a tree branching approach, that is efficient, but generates course patterns. The implementation of the language in this thesis generates patterns by a system of complements, this system seams slightly easier to implement, more uniform, and generates a much finer system of patterns then the case trees used in Agda. However this approach is exponentially less performant then Agda in the worse case.

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

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

$$\frac{\begin{array}{c}\Gamma \vdash \overline{n} \overset{\rightarrow}{:} \Delta \\ \forall\, i\ \left(\Gamma \vdash \overline{pat}_i :_E ?\Delta \quad U\left(\underline{E}, \sigma\right) \quad \sigma\left(\Gamma, |\overline{pat}_i|\right) \vdash \sigma m \overset{\leftarrow}{:} \sigma\left(M\right)\right) \\ \Gamma \vdash \overline{\overline{pat}} : \Delta\ \mathbf{complete}\end{array}}{\Gamma \vdash \mathsf{case}\, \overline{n,}\ \left\{\overline{|\,\overline{pat \Rightarrow} m}\right\} \overset{\leftarrow}{:} M} \cdots$$

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

**Conjecture** Their exists a suitable[4] extension to the surface language TAS that supports patten matching style elimination

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

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 fact noticed by Wadler 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 language[5] as a syntax for its logical programming features).

review

lots of pri
https://gi
/wikis/vie

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 then 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].

Epigram

, 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 several helper functions that can be appropriately typed and elaborated. 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.

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

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

[5] https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/

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 formally[6]. An ideal rule might look like

$$\frac{\begin{array}{cl} \Gamma \vdash \overline{n} : \Delta' & (\textit{scrutinees type check}) \\ \Gamma, \overline{x} : \Delta' \vdash M : \star & (\textit{motive exists and is well formed}) \\ \forall i.\, ? & (\textit{every branch is well typed over all possible instantiations}) \\ ? & (\textit{all scrutinees are handled}) \end{array}}{\Gamma \vdash \mathsf{case}\, \overline{n}, \left\{ \overline{|\, pat \Rightarrow_i m_i} \right\} : M\, [\overline{x} := \overline{n}]} \ldots$$

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

# Part IV
# Related work

## 7 Systems with Data

Many systems that target data formalize only a representative collection of data types, expecting the scheme to be noted and inferred. 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 tp 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.

is

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 , 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 bidirectionally then discussed here to maintain compatibility with the existing Coq system.

tions in
rules and
rm

[6] [Coq92] has a good informal description

[7] https://coq.github.io/doc/v8.9/refman/language/cic.html

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

## 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 V
# TODO

- differentiate identifiers with font, font stuff with code
  - Make identifiers consistent with chapter 2, and locations in chapter 3
- definition package?
- syntax highlighting

## Todo list

for the purposes of this draft separating the "conventional stuff" from the new stuff. this chapter is the conventional stuff. should it be recombined? 1

give explicit example, not the motive 1

give explicit of data eliminations that uses these annotations 1

shorthands! 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

move this down to the typing section? 3

say with rep example 3

who cam up with this first? Martin Lof? 4

Cite De Bruijn for telescopes 4

suspect this also hinges on regularity, which should be addressed more directly 5

define these $\in$ more? 5

extend reductions over lists 5

extend step over lists 5

no duplicate branches 6

reparam o in $[\overline{x} \coloneqq \overline{o}]$ also 6

explicitly talk about how pattern matching subsumes the previous sections 7

names 8

reparam o in $[\overline{x} \coloneqq \overline{o}]$ also 8

clean when I get motive inference working 9

example def by cases 9

Make consistent with other section 9

talk about or formalize the more subtle inference in the actual system 10

as a threat to soundness this should be corrected? 10

review 11

lots of prior work https://gitlab.haskell.org/ghc/ghc/-/wikis/view-patterns 11

Epigram 11

Discus how stratified type systems like ATS handle things (additional equational information) 11

review 12

confirm 12

confirm, cite 12

confirm, cite 12

finish reading this . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
"Inductive definitions in the system Coq rules and properties" confirm . . . . . . . . . . . . . . . . . . . . . . . . 12
CTT, higher inductive types, quotient types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
Clean up this section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
A tutorial implementation of dynamic pattern unification Adam Gundry and Conor McBride (2012) http://adam.gundry.co.uk/pu
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

$$\frac{\Gamma \vdash \Delta\, \overleftarrow{\mathbf{ok}}}{\Gamma \vdash \mathsf{data}\, D\, \Delta\, \overleftarrow{\mathbf{ok}}}$$

$$\frac{\Gamma \vdash \mathsf{data}\, D\, \Delta \quad \forall d.\Gamma, \mathsf{data}\, D\, \Delta \vdash \Theta_d \quad \forall d.\, \Gamma, \mathsf{data}\, D\, \Delta, \Theta_d \vdash \overline{m}_d : \Delta}{\Gamma \vdash \mathsf{data}\, D \,:\, \Delta \left\{\overline{\mid d \,:\, \Theta_d \to D\overline{m}_d}\right\}}$$

...