Chapter 4 (draft): Data and Pattern Matching

Mark Lemay

November 16, 2021

Part I

Introduction

clean up!

The encoded data presented in Chapter 2 is unrealistically inconvenient and is especially implausible for a dependently typed programming language intended to be easy to use.

User defined data is an essential feature of a realistic programming language. Simple data types like `Nat` and `Bool` are essential for organizing readable programs. Dependent data like `Id` can represent the mathematical predicates like equality. Dependent data can also be used to preserve invariants, like the length of a list in `Vec`.

We have opted for data definitions like those found in systems like Agda and Coq. A data definition is formed by a type constructor indexed by value arguments, and a set of constructors that tag data and characterize their arguments. See Figure 1 for the definitions of several standard data types. Data is easy to build and reason about, since data can only be created from its constructors. Unfortunately data elimination is more murky.

How should data be used? One option is a direct eliminator scheme, like Coq uses in its core language. It is worth formalizing that example.

Part II

Data and elimination

A minimal accounting of data can be given by extending the surface language syntax in ...

ebnf? if r
underline

pat or p?

alternativ

differentiate identifiers with font

motive should not need to insist on the type info of the binder?

Grey out things that are surface syntax but not needed for theory

Make identifiers consistent with chapter 2, and locations in chapter 3

Make a module syntax?

The slightly awkward eliminator syntax is designed to be forward compatible with the pattern matching system defined in the rest of this section. Mutual patterns. Recursion does the work of induction in the MLTT style

abbreviate away dumb arrows, unstated separator is a space, also usual syntax (:*)? also shorthands for telescopes

relevent motives!

define a closed context

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

data Nat : * {
| Z : Nat
| S : Nat -> Nat
};
-- Syntactic sugar expands decimal numbers
-- 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)
};

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

Figure 1: Definitions of Common Data Types

telescope,			
Δ, Θ	::=	$.$	empty telescope
	|	$x : M,\ \Delta$	extend telescope
list of O, separated with s			
$\overline{sO}$,$\overline{Os}$	::=	s	empty list
	|	$sO\overline{sO}$	extend list
data type identifier,			
D			
data constructor identifier,			
d			
contexts,			
Γ	::=	...	
	|	$\Gamma, \mathsf{data}\, D\ :\ \Delta \to *\ \left\{\overline{\mid d\ :\ \Theta \to D\overline{m}}\right\}$	data def extention
	|	$\Gamma, \mathsf{data}\, D\ :\ \Delta \to *$	abstract data extention
m, n, M, N	::=	...	
	|	D	type cons.
	|	d	data cons.
	|	$\mathsf{case}\, \overline{N}$,$n\ \left\{\overline{\mid pat \Rightarrow m}\right\}$	data elim. without motive
	|	$\mathsf{case}\, \overline{N}$,$n\ \langle \overline{x \Rightarrow} y : D\,\overline{x} \Rightarrow M\rangle\ \left\{\overline{\mid pat \Rightarrow m}\right\}$	data elim. with motive
(minimal) patterns,			
pat	::=	$\overline{\Rightarrow x} \Rightarrow (d\,\overline{y})$	
pat	::=	$x \Rightarrow pat$	match a variable
	|	$(d\,\overline{x})$	match a constructor
values,			
v	::=	...	
	|	$D\,\overline{v}$	
	|	$d\,\overline{v}$	

Constructors build application, drop this with applications

Figure 2: Surface Language Data

1 Incomplete Eliminations

2 (non) Strict Positivity

3 Specification

telescopes are well formed

$$\frac{\Gamma\,\mathbf{ok}}{\Gamma \vdash .\,\mathbf{ok}}\cdots$$

$$\frac{\Gamma \vdash M : \star \quad \Gamma, x : M \vdash \Delta\,\mathbf{ok}}{\Gamma \vdash x : M, \Delta\,\mathbf{ok}}\cdots$$

suspect this also hinges on regularity

$$\frac{\Gamma\,\mathbf{ok}}{\Gamma \vdash \lozenge : .}\cdots$$

$$\frac{\Gamma, x : M \vdash \Delta \quad \Gamma \vdash m : M \quad \Gamma \vdash \overline{n}\,[x := m] : \Delta\,[x := m]}{\Gamma \vdash m, \overline{n}\ :\ x : M, \Delta}\cdots$$

$$\frac{\Gamma\,\mathbf{ok} \quad \mathsf{data}\,D\,\Delta \in \Gamma}{\Gamma \vdash D\ :\ \Delta \to *}\cdots$$

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

define these $\in$

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

don't need $\Gamma \vdash \overline{N} : \Delta$?

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

may not need scrut wf check? oh but what about the empty types!

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

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

to ensure regularity

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

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

red

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

it's actually kind of fine discriminating between non converting motives?

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

structural reductions

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

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

$$\frac{\overline{m} \Rrightarrow \overline{m'}}{D\overline{m} \Rrightarrow D\overline{m'}} \ldots$$

$$\frac{\overline{m} \Rrightarrow \overline{m'}}{d\overline{m} \Rrightarrow d\overline{m'}} \ldots$$

extend reductions over lists

cbv

$$\frac{}{\mathsf{case}\, \overline{N}, n\ \langle ... \rangle \left\{ \overline{\mid \overline{\Rrightarrow x} \Rightarrow (d'\, \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \rightsquigarrow \mathsf{case}\, \overline{N}, n \left\{ \overline{\mid \overline{\Rrightarrow x} \Rightarrow (d'\, \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}} \ldots$$

$$\frac{\exists \overline{x \Rrightarrow} (d\, \overline{y}_d) \Rightarrow m_d \in \left\{ \overline{\mid \overline{x \Rrightarrow} (d'\, \overline{y}_{d'}) \Rightarrow m_{d'}} \right\}}{\mathsf{case}\, \overline{V}, d\overline{v}\ \left\{ \overline{\mid \overline{x \Rrightarrow} (d'\, \overline{y}_{d'}) \Rightarrow m_{d'}} \right\} \rightsquigarrow m_d \left[\overline{x} \coloneqq \overline{V}, \overline{y}_d \coloneqq \overline{v} \right]} \ \Rrightarrow\text{-case-red}$$

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

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

structural reductions

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

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

$$\frac{\overline{m} \rightsquigarrow \overline{m'}}{D\overline{m} \rightsquigarrow \overline{D\overline{m'}}} \ \ldots$$

what about D? how much of a value should it be?

$$\frac{\overline{m} \rightsquigarrow \overline{m'}}{D\overline{m} \rightsquigarrow \overline{D\overline{m'}}} \ \ldots$$

extend step over lists

4 Bidirectional extension

a convenient bidirectional interpretation

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

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

bidirectional non dependent elimination

$$\frac{\begin{array}{c}\mathsf{data}\, D\, \Delta \in \Gamma \\ \Gamma \vdash n \overset{\rightarrow}{:} D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\,\overline{x} \vdash M \overset{\leftarrow}{:} \star \\ \forall\, d\ :\ \mathit{\Theta} \to D\overline{m} \in \Gamma. \quad \Gamma, \overline{x} : \Delta, \overline{y}_d : \mathit{\Theta} \vdash m_d \overset{\leftarrow}{:} M\end{array}}{\Gamma \vdash \mathsf{case}\, n\ \left\{\overline{\mid (d\,\overline{y}_d) \Rightarrow m_d}\right\} \overset{\leftarrow}{:} M} \ \ldots$$

bidirectional dependent elimination

$$\frac{\begin{array}{c}\mathsf{data}\,D\,\Delta\in\Gamma\\ \Gamma\vdash\overline{N}\overset{\leftarrow}{:}\Delta\quad\Gamma\vdash n\overset{\leftarrow}{:}D\overline{N}\\ \Gamma,\overline{x}:\Delta,z:D\,\overline{x}\vdash M\overset{\leftarrow}{:}\star\\ \forall\,d\,:\,\Theta\to D\overline{m}\in\Gamma.\quad\Gamma,\overline{x}:\Delta,\overline{y}_d:\Theta\vdash m_d\overset{\leftarrow}{:}M\end{array}}{\begin{array}{c}\Gamma\vdash\mathsf{case}\,\overline{,N},n\,\langle\overline{x\Rightarrow}z:D\,\overline{x}\Rightarrow M\rangle\left\{\overline{|\,\overline{x\Rightarrow}(d\,\overline{y}_d)\Rightarrow m_d}\right\}\\ :M\left[\overline{x}:=\overline{N},z:=n\right]\end{array}}\ldots$$

may not need scrut wf check? oh but what about the empty types!

$$\frac{\Gamma\vdash\Delta\,\overset{\leftarrow}{\mathbf{ok}}}{\Gamma\vdash\mathsf{data}\,D\,\Delta\,\overset{\leftarrow}{\mathbf{ok}}}\ldots$$

abuse of notation...

ok?$\Gamma\vdash\Delta\overset{\leftarrow}{:}\star$, perhaps $\Gamma\vdash\Delta\,wf$ and $\Gamma\vdash\Delta\,\overset{\leftarrow}{wf}$. or $\Gamma\vdash\Delta\,\mathbf{ok}$ ...or $\Gamma\vdash\Delta\vdash$ ... or Γ **context** , Δ **telescope** as in [CD18]

abuse of notation...

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

Part III
Pattern Matching

Unfortunately, eliminators are cumbersome for programmers to deal with directly. For instance, in figure 3 we show 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 variable that cannot be identified automatically. Pattern matching is much more ergonomic than a direct eliminator, where 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 Agda and the user facing language of Coq.

Figure 4 shows the extensions to the surface language for data and pattern matching. The syntax of data constructors and data type constructors is standard. Our case eliminators match a tuple of expressions, allowing us to be very precise about the typing of branches.

s are al-

While pattern matching is an extremely practical feature, theoretical accounts tend to be messy. To implement standard pattern matching, a unification procedure is needed to resolve the equational constraints that arise. There are many different strategies to handle these equational constraints. Several options are explored in [CD18]. Worse, the constraints are undecidable in general, since arbitrary computation can be embedded in the type of a constructor.

there is a lot of jenkyness about unification in general, but I think the additional points lose focus?

Parentheses are used to distinguish between matching a single variable and a constructor that takes no arguments TAS pattern matching

$$\frac{\begin{array}{cl}\Gamma\vdash\overline{n}:\Delta' & (\mathit{scrutinees\ type\ check})\\ \Gamma,\overline{x}:\Delta'\vdash M:\star & (\mathit{motive\ exists\ and\ is\ well\ formed})\\ \forall i.\ \left(\forall\,\Gamma',\sigma,\overline{n'}.\,\Gamma,\overline{x}:\Delta'\sim\sigma\Gamma'\ \wedge\ \Gamma'\vdash\overline{pat}_i\sim_{\sigma'}\overline{n'}:\sigma\Delta'\ \supset\ \sigma'\Gamma'\vdash\sigma'm_i:\sigma'\sigma M\right) & (\mathit{every\ branch\ is\ well\ typed\ over\ all\ possible}\\ \forall\,\Gamma',\sigma,n'.\,\Gamma\sim\sigma\Gamma'\ \wedge\ \Gamma'\vdash n':\sigma\Delta'\ \supset\exists i.\,\sigma'\Gamma'\vdash\overline{pat}_i\sim_{\sigma'}\overline{n'}:\sigma\Delta' & (\mathit{every\ input\ is\ handled})\end{array}}{\Gamma\vdash\mathsf{case}\,\overline{n,}\ \left\{\overline{|\,\overline{pat\Rightarrow}_i m_i}\right\}:M\left[\overline{x}:=\overline{n}\right]}$$

last condition is incorrect, since a loop will type but not match

should delta be well formed?

```
-- 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

syntax highlighting would be bomb

Figure 3: Eliminators vs. Pattern Matching

$m...$	$::=$	...	
	$\mid$	$\mathsf{case}\,\overline{n},\ \left\{\overline{\mid pat \Rightarrow m}\right\}$	data elim. without motive
	$\mid$	$\mathsf{case}\,\overline{n},\ \langle \overline{x \Rightarrow} M \rangle \left\{\overline{\mid pat \Rightarrow m}\right\}$	data elim. with motive

(minimal) patterns,

pat	$::=$	x	match a variable
	$\mid$	$(d\,\overline{pat})$	match a constructor

Figure 4: Surface Language Data

uniquness of match. or better yet, any determinitic way to resolve paths

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

matching

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

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

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

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

bidirectional

aproxomations

$\Gamma \vdash \overline{pat} :_{E} ?\Delta$

$$\frac{}{\Gamma \vdash x :_{\{x:M\}} M}\ \cdots$$

$$\frac{d\ :\ \Theta \to D\overline{n} \in \Gamma \quad \Gamma \vdash \overline{m} :_{c} ?\Theta}{\Gamma \vdash d\overline{m} :_{\{M \sim D\overline{n}[\Theta := \overline{m}]\} \cup c} ?M}\ \cdots$$

$$\frac{}{\Gamma \vdash\ . :_{\emptyset} ?.}\ \cdots$$

$$\frac{d\ :\ \Theta \to D\overline{n} \in \Gamma \quad \Gamma \vdash \overline{m} :_{c} ?\Theta}{\Gamma \vdash d\overline{m} :_{\{M \sim D\overline{n}[\Theta := \overline{m}]\} \cup c} ?M}\ \cdots$$

abuse of notation

unification

$U(pat = M, \sigma +)$

not quite a sub propely a ctx thing

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

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

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

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

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

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

there is no inherent reason to limit vars to those that appear in the pattern

non termination of unification (cyclic)

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

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

This complicates the simple story from chapter 2, where the bidirectional system made the TAS system tractable by only adding annotations (and having annotatability). It seems plausible that the bidirectional rules here can be considered an extension over the TAS rules outlines above, but the exact formulation would probably be cumbersome involving non-trivial transformations of case expressions, along with an appropriate equivalence associating those equivalences.

Part IV
Cast Language Data

Surprisingly, the cast system can be extended with a pattern matching construct without unification.

Consider the normal forms of data terms. While in standard dependent type theory a normal form of data, in a closed context, must have the data constructor in head position (justifying the pattern syntax), in the cast language the normal form of data will have a stack of 0 or more casts applied to it. Casts will be wrapped around constructors during the elaboration procedure, and will accumulate during evaluation. If the cast language is extended with a path variable that can represent the stack of equalities then that stack can be matched and used in the body of the branch. Since the type constructor is known, it is possible to check the coverage of the patterns against it's constructors. If every constructor is accounted for, only blameable data remains. Quantifying over casts allows blame to be redirected, so if the program gets stuck in a pattern branch it can blame the malformed input.

to stylize consistently, should use math font, or like a nice image

It makes sense to further extend the cast syntax so that blame can be invoked directly by a path. For example, in 5. Branches set up in this way prove their "unreachability" by using their input to generate blame. Once direct blame is added it makes sense to allow operations on paths, such as concatenation, and reverse so that more proofs of inequality are possible. Proofs of inequality can be further helped by inspecting the arguments of type constructors and data constructors. See 6 for the extended path syntax.

We conjecture that this extension preserves cast soundness.

Extending the syntax in this way complicates the type system and the reduction rules. We include a sample of our reduction rules in

fig

. We conjecture that these extensions support all of our expected properties.

technically speaking, telescopes should generalize to the different syntactic classes

~ is a bit of a hack

typing rules (pathing rules)

not sure it's good to have paths be quantifiable?it will never come up in elabvoration. try it now for simplicity

$$\dfrac{x_{p} : A \approx B \in H}{H \vdash x_{p} : A \approx B}$$

$$\dfrac{H \vdash A : \star}{H \vdash refl : A \approx A}$$

```
-- standard data in normal form, 3
S (S (S 0))

-- cast data in normal form
S (S (S 0) :: Nat ) :: Nat :: Nat :: Nat
S (S (S 0) :: Nat ) :: Bool :: Nat
True :: Nat

-- cast pattern matching
case x <_ => Bool> {
| (Z :: _) => True
| (S (Z :: _) :: _) => True
| (S (S :: _) :: _) => False
}

-- extract specific blame,
-- c is a path from Bool~Nat
case x <_ => Nat> {
| (S ((true::c)::_) :: _) =>
 add (false :: c) 2
}

-- can reconstitute any term,
-- not always possible with unification
-- based pattern matching
case x <_:Nat => Nat> {
| (Z :: c) => Z :: c
| (S x :: c) => S x :: c
}

-- direct blame
case x <_ => Nat> {
| (S (true::c) :: _) => Bool =/=c Nat
}

peek x =
case x <_: Id Nat 0 1 => Nat> {
  | (refl x :: _) => x
}

peek (refl 4 :: Id Nat 0 1) = 4
```

Figure 5: Cast Pattern Matching

path var,

x_p

path exp.,

$p, p' \quad ::= \quad x_p$

$\quad | \quad A_{\ell.x \Rightarrow C} B$ concrete cast

$\quad | \quad refl$

$\quad | \quad pp'$

$\quad | \quad rev\, p$

$\quad | \quad inTC_i\, p$

$\quad | \quad inC_i\, p$

cast pattern,

$patc \quad ::= \quad x \mid d\, \overline{pat} ::_{x_p}$

cast expression,

$a... \quad ::= \quad ...$

$\quad | \quad D\, \overline{a}$ type cons.

$\quad | \quad d\, \overline{a}$ data cons.

$\quad | \quad \mathsf{case}\, \overline{n,}\ \left\langle \overline{\Delta \Rightarrow} M \right\rangle \left\{ \overline{|\, \overline{patc \Rightarrow} n} \right\}$ data elim.

$\quad | \quad A \neq_p B$ force blame

$\quad | \quad a ::_{A,\ell.C} B$ concrete cast

$\quad | \quad a ::_{A,p,x.C} B$ symbolic cast

$\quad | \quad a \sim_{\ell o} b$

observations,

$o \quad ::= \quad ...$

$\quad | \quad o.App[a]$ application

$\quad | \quad o.TCon[i]$ type cons. arg.

$\quad | \quad o.DCon[i]$ data cons. arg.

steal path syntax from CTT?

extend H ctxs

concrete casts need more explanation

$\tilde{}$ should be removed from the grammar? is functions better as an op of elaboration

Figure 6: Cast Language Data

$$\frac{H \vdash B : \star \quad H, x : B \vdash C : \star \quad H \vdash b : B \quad H \vdash b' : B \quad C\left[x \coloneqq b\right] \equiv A \quad C\left[x \coloneqq b'\right] \equiv A'}{H \vdash A_{\ell.x \Rightarrow C} A' : A \approx A'}$$

ALT, would then need to resolve endpoint def equality

$$\frac{H \vdash a : A \quad H \vdash a' : A \quad H, x : A \vdash C : \star}{H \vdash assert_{\ell.(a=a':A).x \Rightarrow C} : C\left[x \coloneqq a\right] \approx C\left[x \coloneqq a'\right]}$$

$$\frac{H \vdash p : A \approx B \quad H \vdash p' : B \approx C}{H \vdash p\,p' : A \approx C}$$

$$\frac{H \vdash p : A \approx B}{H \vdash rev\,p : B \approx A}$$

$$\frac{H \vdash p : D\,\overline{a} \approx D\,\overline{b}}{H \vdash inTC_i\,p : a_i \approx b_i}$$

$$\frac{H \vdash p : d\,\overline{a} \approx d\,\overline{b}}{H \vdash inC_i\,p : a_i \approx b_i}$$

the last few rules that manually transform paths, are specifically designed to encode the implicit reasoning behind unification.

allow conversion of endpoints?

typing rules

$$\frac{H \vdash C : \star \quad H \vdash p : A \approx B \quad (AandBDisagree)}{H \vdash A \neq_p B : C}$$

$$\frac{H \vdash a : A \quad H, x : B \vdash C : \star \quad C\left[x \coloneqq b\right] \equiv A \quad C\left[x \coloneqq b'\right] \equiv B}{H \vdash a ::_{A,\ell.x \Rightarrow C} B}$$

ALT

$$\frac{H \vdash a : A \quad H \vdash a' : A \quad H, x : A \vdash C : \star \quad H \vdash a : c\left[x \coloneqq a\right]}{H \vdash c ::_{\ell(a=a':A).x \Rightarrow C} \quad : C\left[x \coloneqq a'\right]}$$

ALT remove concrete casts and merely use a symbolic cast instead?

...

$$\frac{H \vdash a : A \quad H, x : B \vdash C : \star \quad C\left[x \coloneqq b\right] \equiv A \quad C\left[x \coloneqq b'\right] \equiv B \quad p : b \approx b'}{H \vdash a ::_{A,p.x \Rightarrow C} B}$$

ALT

$$\frac{H \vdash c : C\left[x \coloneqq a\right] \quad H, x : A \vdash C : \star \quad H \vdash p : a \approx a'}{H \vdash c ::_{p.x \Rightarrow C} \quad : C\left[x \coloneqq a'\right]}$$

$$\frac{\begin{array}{c} H \vdash \overline{a} : \Delta \\ H, \Delta \vdash B : \star \\ \forall\, i \; \left(H \vdash Gen\left(\overline{patc}_i : \underline{\Delta, \Theta}\right) \quad \Gamma, \Theta \vdash m : M\left[\Delta \coloneqq \overline{patc}_i\right]\right) \\ H \vdash \overline{\overline{patc}} : \Delta\ \mathbf{complete} \end{array}}{\begin{array}{c} \mathsf{case}\,\overline{a,}\ \left\langle \overline{\Delta \Rightarrow} B \right\rangle \left\{ \overline{|\,\overline{patc \Rightarrow} b} \right\} \\ : M\left[\Delta \coloneqq \overline{n}\right] \end{array}} \cdots$$

Gen is defined as

$$\frac{}{H \vdash Gen\left(. : ., .\right)} \cdots$$

$$\frac{\sim H \vdash A : \star \sim}{H \vdash Gen\left(x : (x : A),\ x : A\right)} \cdots$$

$$\frac{\sim H \vdash A : \star \sim}{H \vdash Gen\,(x : A,\ x : A)}\;\ldots$$

$$\frac{d\ :\ \mathit{\Theta} \to D\overline{a} \in H \quad H \vdash Gen\left(\overline{pat_c} : \mathit{\Theta}, \Delta\right)}{H \vdash Gen\left(d\overline{pat_c} ::_{x_p} : D\overline{b},\ \Delta, x_p : D\overline{a} \approx D\overline{b}\right)}\;\ldots$$

$$\frac{H \vdash Gen\,(pat_c : A, \Theta) \quad H, \Theta \vdash Gen\left(\overline{pat_c} : \Delta\,[x \coloneqq pat_c]\,, \Theta'\right)}{H \vdash Gen\left(pat_c\overline{pat_c} : (x : A, \Delta)\,, \Theta\Theta'\right)}\;\ldots$$

other rules similar to the surface lang

| $a \sim_{\ell o} b$ hacky erasure

observations,

o ::= ...

| $o.App[a]$ application

| $o.TCon[i]$ type cons. arg.

| $o.DCon[i]$ data cons. arg.

old style red rules

$$\frac{}{rev\ (p\,p') \rightsquigarrow (rev\,p')\,(rev\,p)}$$

$$\frac{}{inTC_i\ (p\,p') \rightsquigarrow (inTC_i\,p')\,(inTC_i\,p)}$$

$$\frac{}{inC_i\ (p\,p') \rightsquigarrow (inC_i\,p')\,(inC_i\,p)}$$

$$\frac{}{inTC_i\,refl \rightsquigarrow refl}$$

$$\frac{}{inC_i\,refl \rightsquigarrow refl}$$

$$\frac{\overline{a}_i = a'\ \overline{c}_i = c'\ \overline{b}_i = b'}{inTC_i\ \left(D\,\overline{a}_{\ell.D\,\overline{c}}D\,\overline{b}\right) \rightsquigarrow a'_{\ell.c'}b'}$$

$$\frac{}{inC_i\ ((a :: A)_{\ell.c}\,b) \rightsquigarrow inC_i\ (a_{\ell.c}b)}$$

$$\frac{}{inC_i\ (a_{\ell.c}\,(b :: B)) \rightsquigarrow inC_i\ (a_{\ell.c}b)}$$

$$\frac{}{inC_i\ \left(a_{\ell.(c::C)}b\right) \rightsquigarrow inC_i\ (a_{\ell.c}b)}$$

$$\frac{\overline{a}_i = a'\ \overline{c}_i = c'\ \overline{b}_i = b'}{inTC_i\ \left(d\,\overline{a}_{\ell.d\,\overline{c}}d\,\overline{b}\right) \rightsquigarrow a'_{\ell.c'}b'}$$

$$\frac{}{a ::_{A,p\,refl,x.C} B \rightsquigarrow a ::_{A,p,x.C} B}$$

$$\frac{}{\begin{array}{c} a ::_{A,p\,A'_{\ell.C''}B',x.C} B \rightsquigarrow \\ a ::_{A,p,x.C} C\,[x \coloneqq A'] ::_{\ell.C[x:=C'']} C\,[x \coloneqq B'] \end{array}}\;c$$

c?

$$\frac{}{(a ::_{A,p,x.C} C) \sim_{\ell o} b \rightsquigarrow a \sim_{\ell o} b}$$

$$\frac{}{a \sim_{\ell o} (b ::_{B,p,x.C} C) \rightsquigarrow a \sim_{\ell o} b}$$

Part V

Elaborating Eliminations

To make the overall system behave as expected we do not want to expose users to equality patterns, or force them to manually do the blame bookkeeping. To work around this we extend a standard unification algorithm to cast patterns with instrumentation to remember paths that were required for the solution. Then if pattern matching is satisfiable, compile additional casts into the branch based on its assignments. Unlisted patterns can be checked to confirm they are unsatisfiable. If the pattern is unsatisfiable then elaboration can use the proof of unsatisfiability to construct explicit blame. If an unlisted pattern cannot be proven “unreachable” then we could warn the user, and like most functional programming languages, blame the incomplete match if that pattern ever occurs.

"can", "could", weasel word until implemented

add rules of unification? the rules as implemented are standard, needing extended normalization is weird

We found that our normalization procedure (Figure **??**) unexpectedly blocked unification. This is because the normalization is conservative about throwing away casts that could later lead to errors. For instance, with normalization, unification will get stuck on terms like $x ::_{\mathbb{N}} \mathbb{N} = 5$. Thus we employ a more optimistic normalization procedure that will collapse casts that are equivalent Thus $x ::_{\mathbb{N}} \mathbb{N} \overset{::}{\rightsquigarrow} x$ allows us to solve the unification problem and assign $x := 5$. This approach has worked well so far.

lence is
l hacky.
dent pat-
unprin-
. I can’t
a situa-
could be
l like it
tioned.

In order to make the additional casts generated from unification we further extend the cast construct so it supports congruence. For instance, if x has type $Vec\, y\, z$ and the unification procedure established $y := Bool$ and $z := 5$ then we should cast $x :: Vec\, Bool\, 5$. Unlike in Section 3 this requires that we use evidence to inject a cast within a type. This can be done inherently by adding an "untyped" component to the cast annotation. The syntax that achieves this can be seen in 6.

Part VI

Related work

5 Systems with Data

Minimal data with Sigma and Unit

ML W types

UTT[Luo90, Luo94]

I am unaware of any clear, complete account of CIC in English. A bidirectional account of CIC is given in [LB21], though it uses a different style of biderectionality then discussed here to maintain compatibility with the existing Galina grammar.

6 Pattern matching

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

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.

[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.

Part VII
TODO

- review chapter 9 of file:///Users/stephaniesavir/Downloads/Computation-and-Reasoning.dependently

Todo list

7 notes

there are several simpler systems that can be worked through: eliminator style patterns, cast patterns, but to bring it all together we need congruence over functions.

adding paths and path variables means that constructs can still fail at runtime, but they can blame the actually problematic components

validating the K axiom, not that equalities are unique, merely that we don't care which one of the unique equalities is used.

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.

8 unused

...