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

Mark Lemay

November 29, 2021

for the perposes of this draft seperating the "conventional stuff" from the new stuff? should it be recombined?

Part I

Introduction

clean up!

User defined data is a core feature of a realistic programming language. Simple data types like `Nat` and `Bool` are essential for organizing readable programs. In dependently typed languages, 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 is unrealistically inconvenient and is especially implausible for an "easy to use" dependently typed programming language.

A data definition is formed by a type constructor indexed by 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 defined in this style is easy to build and reason about, since data can only be created from its constructors. Unfortunately data elimination is more murky.

Part II

Data and direct elimination

How should data be used? Since the term of a given type can only be created from one of a few constructors, we can completely handle data of a type if each sonstructor is acconted for. For instance, Nat has the two constructors Z and S (which holds the preceding number), so teh expression $\texttt{case}\, n\ \{ |\, Z \Rightarrow Z \,|\, S\, x \Rightarrow x \}$ will extract teh preceeding number from n(or 0 if $n = 0$). In this light, boolean case eleimination is corresponds to an if-then-else expression. This style of case elimination is pervasive in ML style langugaes and has become popular in more mainstrame languages such as Python and Java.

double ch
not restric

We will need to extend the syntax above to support realistic dependent type checking. Specifically, we will need to add a motive that allows the type checker to compute the output type of the branches if the very in terms of the input. For insance, recursively generating a vector of a given length. We may also want to use some values of teh 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].

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

ebnf? if r
underline

differentiate identifiers with font

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

[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.

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

Figure 2: Surface Language Data

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

The case eliminator first takes the explicit type arguments, followed by a scrutinee of correct type. Then optionally a motive that characterizes the output type of each branch with all the type arguments and scrutiny 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 0 or more typed bindings. This allows us to set up data definitnions in a clean way. Also we define syntactic lists, when syntax is listed it can be used to generalize dependent pairs, this becomes helpfule wehn we need to extract the applide arguments of a constructor.

In the presence of general recursion this form of case eliminator is very expressive. Well-founded recursion can be used to make structurally inductive computations that can be interperted as proofs.

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

define a closed context

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

1 Incomplete Eliminations

Consider the match

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

this match will get stuck if 0 is substitutted for x. Because it is easy to ensure all constructors are matched, the surface language type assignment system will require cases to be exhuastive.

When we get to the cast language, we will allow in-exhaustive data to ber reported as an elaboration 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.

Example!

This data can cause unexpected non-termination. Dependent types usually requires a strictness check to eliminate this possiblity. However this would block usfule constructions like higher order abstract synax, and co-inductive uses of data. Additionally since non-termination is already allowed in the surface TAS, we will not restrict ourselves to strictly positive date.

3 Specification

The type assignment system must be extended with the rules in . TODO

telescopes are well formed

$$\frac{\Gamma\ \mathbf{ok}}{\Gamma \vdash .\ \mathbf{ok}}\ \ldots$$

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

suspect this also hinges on regularity

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

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

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

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

define these $\in$

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

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

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

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

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

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

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

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

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

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

structural reductions

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

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

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

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

extend reductions over lists

cbv

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

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

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

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

structural reductions

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

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

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

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

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

extend step over lists

While similar systems are explored in , we will not prove the Type soundness of the system here. For clarity we will list it as a conjecture.

Conjecture the data aextention to the surface langugae is type sound.

4 Bidirectional extension

A bidirectional interpretation exists over the type assignment rules listed above

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

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

bidirectional non-dependent elimination

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

tying information flows outside in.

bidirectional dependent elimination

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

information flows from the inside out

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

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

abuse of notation...

```
-- 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

ok?$\Gamma \vdash \Delta \overleftarrow{:}\,\overline{\star}$, perhaps $\Gamma \vdash \Delta\, wf$ and $\Gamma \vdash \Delta\, \overleftarrow{wf}$. or $\Gamma \vdash \Delta\,$**ok** ...or $\Gamma \vdash \Delta \vdash$ ... or $\Gamma\,$**context** , $\Delta\,$**telescope** as in [CD18]

abuse of notation...

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

Conjecture the data extention to the bidirectional surface langugae is type sound.

Conjecture the data extention to the bidirectional surface langugae is weakly annotatatable from the data extention of the surface langugae.

Part III
Pattern Matching

Unfortunately, the eliminator style is 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 are not identified automatically by eliminators. 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. Additionally this style allows for syntactic sugar for easy definitions of functions by cases.

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

Figure 4: Surface Language Data

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

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

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

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

Figure 5: Surface Language Match

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,

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

will match the patterns

x where x is equal to the full expression

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

$Cons\ -\ x\ -\ (Cons\ -\ y\ -\ -)$ where $x = true$, $y = false$

so the expression

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

The explicit rules for pattern matching are listed in 5.

It is now possible for branches to overlap, which could allow nondeterministic reduction. There are several pluasable ways to handle this, such as reuiring each branch to have independent patterns, or requireing patterns have the same behaviour when tehy oeverlap . 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 on each branch, and to confirm the impossibility of unwritten branches. There is no "optimal" strategy to handle these equational constraints, since the constraints are undecidable in general, since arbitrary computation can be embedded in the arguments of a type constructor. Any approach will have to be an aproxomation that performs well in practice. Several options are explored in [CD18]. In practice this procedure usually takes the form of a first order unification.

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

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

5 First Order Unification

When type checing the branches of the a case expression, teh patterns are interperted as epressions under bindings for each variable used in teh pattern. If thses equations can be unified, then the brach will typecheck under the

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

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

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

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

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

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

Figure 6: Surface Language Unificaiton

variable assignments, with teh additional typing information. For insatnce, 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 ver simple example, in the worst case we may have equations in the form $m\,n=m'\,n'$ which are hard to solve (until an assignment of $m=\lambda x.x$, and $m'=\lambda-.0$ are solved).

uniquness of unificaiton solution

A simplified version of a typical unifiaction procedure is listed in 6. The unifiacation does not exclude teh possibly cyclic assingnemtns that could occur $x=S\,x$

as a threat to soundness this should be corrected?

. Unification is not guarenteed to terminate since it relies on definitional equaliteis.

After the branches have typechecked we should makes sure that they are exhastive.again tehre are several possible strategies. In general it is undecidabel wether any given pattern is impossible or not, so a practical approxomation must be chosen. At least programmers have the ability to manually include non bovous branches and prove thier imposiblity, or direct those branches to dummy outputs. Thoudh there is a real risk that the unification procedure gets stuck in ways that are not clear to the programmer, and a clean error messafge may be very dificult.

Ususally this priomative impossiblility is tied to a contrediction of the unificaiton procedure. but there is still a matter of generating patterns that cover all the unmatched cases. Again there is no clear best way to do this since a more fine devision of patterns may allow enough aditional dffinitional information to show unsatisfiablity, while a more coarse devision of patterns may be more efficient. Agda uses a tree branching approach, that is efficincent but generates course patterns. The current experemental implementation of the language in this thesis generates patterns by a system of complements, this system seams slightly eaiser to implement, more uniform, and generates a much finer system of pattterns then the case trees used in agda. However this aproach is exponentially less performant then Agda in the worse case.

All told we can extend the bidriectional system with rules that look like

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

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

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

Conjecture Their exists a suitable[2] extention to the surface language TAS that supports patten matching style elimination

Conjecture The bidirecitonal extention listed here is weakly anotatable with that extention to teh surface language.

Additionally, it makessense to allow some additional type annotations in the motive and for these annotations to swich the the type inference of teh scrutinee into a type-check. Along with a full syntax of modules, and even mutually defined data types. For simplicity these have been excluded from teh 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 . This has lead to making pattern behavior override-able in Scala. An extension in GHC allows some computations to happen within a pattern match. It seems unreasonable to extend patterns to arbitrary computation (thought this is exactly what teh Curry language does as a way to make use of it's logical programming features).

In the presence of full-spectrum dependent types, the perspective shifts. Any terminating typing procedure will necessarily exclude some typable patterns and be unable to exclude some reachable branches. Since dependent patterns are already attacking a much more difficult problem then in the non-dependent case but also only considering data values (no functions), 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 . In principle it seems that dependent case expressions could be extended with uniqueness side conditions, arbitrary computation or some amount of constraint solving, without being any theoretically worse than usual unification.

, Agda and Idris attempt to deal with these issues using **with** syntaxx that allows further branging based on teh computation of each branch. This is justifed as syntactic sugar that corresponds to several halper functions that can be apropriately typed. The language described in this thesis does not use the with side condition since nested case expressions carry the same compuatational behavour, and the elaboration to the cast langugae will allow possibly questionable typing.

ATS

How should overlapping branches be handled? partial evaluation of case expressions can change the definitional nature of the theory.

the details of pattern matching chang e the logical character of the system. Since non-ermination is allowed in the langugae described here the logical issues are less of a concern. However it is worth noting that pattern matching as described here validates axiom k and thus apears unsuatable for Hott or CTT developments.

We have glossed over the definitional behavior of case branches in this chapter since we plan sidestep the issue with the cast langauge. Though it is still worth noting that their are several ways to set up the definitional reductions. Agda style case trees may result in unpredicatble definitional equalities (in so far as definitional behavour is ever predicatable). advocates for a more conservative aproach that makes function definitionas be cases defintiional (wich is nice in that it respectes the surface langauge euqality, but shifts teh dificulties to overlapping branches and does not allow shadowinf behavior programmers are used to). another extream would be to only allowredcutions at fully computed scrutinee valuess, as in trellies worke). Alternatively a partial reduction is possible, such that branches are eliminated as they are found unreachable and substituations made as that are availible. This last aproach is experimantally implemented in the implementation.

This complicates the simple story from chapter 2, where the bidirectional system made the TAS system tractable by only adding annotations (and having annotatability). We have only conjectured the existnace of a saitable TAS

[2] supporting at least subject reduction, type soundnes, and regularity

system. If the definitinoal equality that feeds the TAS is generated by a system of reductions, any of the reduction strategies from the last section will generate a different TAS with subtly different characteristics. For instance, insisting on a call-by-value case reduction will leave many equivelent computiaitons unassociated. If the TAS system uses partial reductions it will need to inspect the constructors of the scrutinee in order to preserve typeing over reduction. Agda style reductions need to extend syntax under reduction to account for side conditions. For this reason it is rare to see a fully formailized account of pattern matching.

Ideally the typeing rule for pattern matching case expression in teh TAS should not use the notion of pattern matching at all. Instead the rule should characterize the behavour that is required directly and formally[3]. An ideal rule might look like

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

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

Minimal data with Sigma and Unit

ML W types

UTT[Luo90, Luo94] is an extention to ECC that specifies a scheme to define strictly positive data types by way of a logical framework defined in MLTT. This scheme generates primative recursors, and does not iunherently support pattern matching.

finish reading this

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 language.

CTT, higher inductive types, qoatent types

8 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 ?

[3]Cite has a good imformal description

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.

[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 V
TODO

Todo list

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

...