

# Chapter 4b (draft): Data and Pattern Matching in the Cast Language

Mark Lemay

December 10, 2021

## Part I

## 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 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 a value. 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 its 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 original faulty assumption.

If we hope to target the properties of Chapter 3 we need the cast language to be **cast sound**. Elaboration should satisfy the additional correctness properties relative to a type assignment system and bidirectional system. In this case we will target the previously described TAS with a first order unification style pattern matching.

“gradual c

To account for unreachable patterns, we can record the proof of inequality that makes the unification unsatisfiable with an explicit blame syntax. It is perfectly possible for a case expression to reduce into such an “unreachable” branch if a bad case is made. If this happens blame will be reflected back onto a specific problematic cast of the input.

When we internalize the notion of path we need to support path operations that correspond to each of operations used by the unification of the bidirectional type system. Specifically we need to index into the arguments of data constructors, and data type constructors. We will also need to be able to reverse paths and concatenate paths since the unification algorithm implicitly uses these properties of equality. Finally we need to be able to remove and append casts from and to the endpoints of paths.

During elaboration, after a pattern is unified, we will inject the proofs of equality into the branch terms so that they cast check. This will require that our notion of paths support congruence.

## 1 Examples

### 1.1 Head

For instance if the user case matches the head of  $x$  where  $x$  has type  $\text{Vec } A (S n)$  in the surface language,

```
case x <_:Vec Bool (S n) => Bool> {  
  | Cons _ a _ _ => a  
}
```

What can go wrong in the presence of casts?

- a faulty cast assumption may have made  $x$  appear to be a Vector even when it is not. For instance, `True :: Vec Bool 3`
- the vector may be empty but cast to look like it is inhabited. For instance `Nil Bool :: Vec Bool 5`
- the vector may have a type that is not Bool. For instance `Cons Nat 3 ... :: Vec Bool 5`

To handle these issues elaboration will perform unification resulting in the term

```
case x <_:Vec Bool (S n) => Bool> {
| (Cons A a y _) :: p => a::InTCon0(p)
| (Nil A) :: p => !InTc1(p)
}
```

In the first branch  $A : \star$ ,  $a : A$ ,  $y : \text{Nat}$ , while  $p$  is a path variable  $p : \text{Vec } A \ (S y) \approx \text{Vec Bool } (S n)$  that contains evidence that the 2 types have been asserted to be the same.  $\text{InT}_0 p$  extracts the 0th argument from the type constructor  $p : \text{Vec } \underline{A} \ (S y) \approx \text{Vec } \underline{\text{Bool}} \ (S n)$  resulting in the type  $\text{InT}_0 p : A \approx \text{Bool}$ . The body of the branch casts  $a$  along  $\text{InT}_0 p$  to  $\text{Bool}$ .

In the second branch,  $A : \star$ ,  $p : \text{Vec } A \ Z \approx \text{Vec Bool } (S y)$ . The body of the branch encodes the contradiction observed by noting  $Z \neq S y$ .

Consider the ways  $x$  could go wrong,

- If the case tries to eliminate  $\text{True} :: \text{Vec Bool } 3$ , the type constructor is not matched so the faulty assumption can be blamed
- If the scrutinee is empty, we will fall into the  $\text{Nil}$  branch, which will reflect the underlying faulty assumption, via the explicit blame syntax
- If the vector is inhabited by an incorrect type, it will return  $3 :: \text{Bool}$  with the  $a$  cast that rests on the faulty assumption of  $\text{Vec Nat } 5 = \text{Vec Bool } 5$ . If checking is done aggressively this will produce blame immediately. If minimal checking is done, the blame will be discovered whenever the result is eliminated.

## 1.2 Sum

Consider the more involved example that sums the 2 numbers in a vector

```
case x <_:Vec Nat 2 => Nat> {
| Cons _ i _ (Cons _ j _ _) => i+j
}
```

the elaboration procedure will produce

```
case x <_:Vec Nat 2 => Nat> {
| (Cons Nat' i n' (Cons Nat'' j n'' rest):: p1):: p2 => i::InTc0(p2) + j::(InTc0(p1).InTc0(p2))
| (Nil Nat') :: p => !InTc1(p)
| (Cons Nat' i n' (Nil Nat'')):: p1):: p2 => !InTc1(p1).InTc1(p2)
}
```

- in the first branch we have,  $p1 : \text{Vec Nat'' } (S n'') = \text{Vec Nat' } n'$ ,  $p2 : \text{Vec Nat' } (S n') = \text{Vec Nat } 2$ ,  $i:\text{Nat'}$ ,  $j:\text{Nat''}$ 
  - this means we can deduce  $\text{InTc0}(p2) : \text{Nat'} = \text{Nat}$ , and  $\text{InTc0}(p1) \text{ InTc0}(p2) : \text{Nat''} = \text{Nat}$
- in the 2nd branch,  $p : \text{Vec Nat' } 0 = \text{Vec Nat' } 2$ 
  - which is unsatisfiable, by  $\text{InTc1}(p) : 0 = 2$
- in the 3rd branch,  $p1 : \text{Vec Nat'' } 0 = \text{Vec Nat' } n'$ ,  $p2 : \text{Vec Nat' } (S n') = \text{Vec Nat } 2$ 
  - which is unsatisfiable by  $\text{InTc1}(p1) \text{ InTc1}(p2) : 0 = 2$ . We don't need to know which path is problematic beforehand, only that the combination causes trouble.

We know from unification what the type and value of every term is supposed to be, so casts can be injected using evidence from the pattern

### 1.3 assignment at the type level

There are more complicated possibilities to consider

```
case x <_: Id Nat 2 2 => Vec Bool 2 > {
| refl _ a => rep a Bool True
}
```

this will elaborate to

```
case x <_: Id Nat 2 2 => Vec Bool 2 > {
| (refl Nat' a)::p => (rep (a::InTC0(p))):: Cong (x=> (A : *) -> A -> Vec Nat x)
(CastL (InTC0(p)) (InTC1 p))

Nat 9
}
```

where  $Nat' : \star$ ,  $a : Nat'$ , and  $p : Id Nat' a a \approx Id Id 2 2$ . Since we have  $p : Id Nat' a a \approx Id Id 2 2$ ,  $(a :: InTC_0(p))$  casts  $a$  to a  $Nat$  but then we need to evidence that  $(A : \star) \rightarrow A \rightarrow Vec Nat (a :: InTC_0(p)) \approx (A : \star) \rightarrow A \rightarrow Vec Nat 2$ . This means we first need congruence to select the portion of interest. Ideally we would have a path from  $(a :: InTC_0(p)) \approx 2$ , but we only have a path  $(InTC_1(p)) : a \approx 2$ . We will use the meta operation  $CastL$  to produce the path  $CastL_{InTC_0(p)}(InTC_1 p) : (a :: InTC_0(p)) \approx 2$ .

### 1.4 need to remove a cast

Consider this surface language case that extracts the last element from a non-empty list. Assume the function  $last : (n : Nat) \rightarrow Vec A (S n) \rightarrow A$  is in scope.

```
case v <_: Vec A (S x) => A > {
| Cons _ a (Z) _ => a
| Cons _ _ (S n) rest => last n rest
}
```

this will elaborate into

```
case v <_: Vec A (S x) => A > {
| (Cons A' a' (Z)::q rest) :: p => a' :: rev(inTC1(p))
| (Cons A' a' (S n)::q rest)::p => last n (rest :: p')
| (Nil A')::p => !TCon1(p)
}
```

In the 2nd branch we have  $A' : \star$ ,  $a' : A'$ ,  $n : Nat$ ,  $q : Nat \approx Nat$ ,  $rest : Vec A' ((S n) :: q)$ , and  $p : Vec A' (S((S n) :: q)) \approx Vec A (S x)$ . Note that we cannot unify a solution unless we can freely remove casts from endpoints, otherwise it becomes impossible to construct a path from  $Vec A' ((S n) :: q) \approx Vec A (S n)$ . We will need an operator that can remove casts from the endpoints of paths that arise from unification, we will call these operators  $uncastL$  and  $uncastR$ . With these operations we can match the process of surface level unification so that

$$p' = Cong_{x \Rightarrow Vec A' x} (UncastR(refl)) \circ Cong_{x \Rightarrow Vec x (S n)} (TCon_0 p) : Vec A' ((S n) :: q) \approx Vec A (S n)$$

In the first branch we have,  $A' : \star$ ,  $a' : A'$ ,  $q : Nat \approx Nat$ ,  $rest : Vec A' (Z :: q)$ , and  $p : Vec A' (S(Z :: q)) \approx Vec A (S x)$ . Unification can proceed to derive  $TCon_1(Con_0(p))^{-1} : x \approx Z :: q$  and  $TCon_1(p)^{-1} : A \approx A'$ .

In the “unreachable” branch we have  $p : Vec A' Z \approx Vec A' (S x)$ , which is contradicted by  $TCon_1 p : Z \approx S x$

Example, translate out to motive

## 2 Cast Language

Pattern matching across dependent data types will allow for observations that were impossible before. We can now observe specific arguments in type constructors and term constructors. Since function terms can appear directly in data type constructors it is now possible to observe functions directly. Unfortunately formalizing this syntax seems to require some heavy constructs, see 1 for the extended path syntax.

In this thesis we have taken an extremely extensional perspective, terms are only different if an observation recognizes a difference. For instance this approach justifies equating the functions  $\lambda x \Rightarrow x + 1 = \lambda x \Rightarrow 1 + x$  without proof, even though they are usually definitionally distinct. Therefore we will only blame inequality across

path var,		
$x_p$		
assertion index,		
$k$		
assertion assumption,		
$kin ::= k = left \mid k = right$		
assertion maps,		
$kmap ::= \overline{kin},$		
casts under assumption,		
$kcast ::= \overline{kmap, p};$		
path exp.,		
$p, p' ::= x_p$		
$Assert_{k \Rightarrow C}$	concrete assumption	
$\overline{p\circ}$	concatenated paths	
$cong_{x \Rightarrow a p}$	congruence	
$p^{-1}$		
$inTC_i p$		
$inC_i p$		
cast pattern,		
$patc ::= x \mid (d \overline{patc}) :: x_p$		
cast expression,		
$a... ::= ...$		
$D$	type cons.	
$d$	data cons.	
$case \overline{a}, \{ \overline{patc \Rightarrow b} \mid \overline{patc' \Rightarrow !_\ell} \}$	data elim.	
$!_{kcast}$	force blame	
$a :: kcast$	cast	
$\{a \sim_{k,o,\ell} b\}$	assert same	
observations,		
$o ::= ...$		
$o.App[a]$	application	
$o.TCon_i$	type cons. arg.	
$o.DCon_i$	data cons. arg.	
$inEx_{\overline{patc}}[\overline{a}]$	in-exhaustive pattern match	

extend H ctxs

Figure 1: Cast Language Data

(the empty path)	written	$refl$	
		$kmap, \quad p$	
$a :: kmap, p; kmap', q; ...$	written	$a :: kmap', \quad q$	
		$... \quad ...$	
$kmap, k = left, \quad p$		$kmap, \quad p$	
$kmap, k = right, \quad p$		$kmap, \quad p$	
$a :: kmap', k = left, \quad q$	written	$a :: kmap', \quad q$	when $k$ is irrelevant
$kmap, k = right, \quad q$		$... \quad ...$	
$... \quad ...$			

Figure 2: Surface Language Abbreviations

functions if two functions that were asserted to be equal return different observations for “the same” input. Tracking that 2 functions should be equal becomes complicated, the system must be sensible under context, functions can take other higher order inputs, and function terms can be copied freely.

Example back to pattern matching?

The cleanest way I could find to encode the a dynamic check functions for equality, was with a new term level construct<sup>1</sup>. This assertion that two terms are the same is written as  $\{a \sim_{k,o,\ell} b\}$  and will evaluate  $a$  and  $b$  in parallel until a head constructor is reached on each branch. If the constructor is the same it will commute out of the term. If the head constructor is different the term will get stuck with the information for the final blame message. For instance,  $\{\lambda x \Rightarrow x + 1 \sim_{k,o,\ell} \lambda x \Rightarrow 1 + x\} \rightsquigarrow_* \lambda x \Rightarrow \{x + 1 \sim_{k,o,App[x],\ell} 1 + x\}$ .

Since this equational construct is already needed for functions, we will use it to handle all equality assertions. For instance,  $\{(\lambda x \Rightarrow S x) Z \sim_{k,o,\ell} 2 + 2\} \rightsquigarrow_* \{S Z \sim_{k,o,\ell} S (1 + 2)\} \rightsquigarrow_* S \{Z \sim_{k,o,DCon0,\ell} 1 + 2\} \rightsquigarrow_* S \{Z \sim_{k,o,DCon0,\ell} S 2\}$ . We compute past the first  $S$  constructor and blame the predecessor for not being equal.

Unfortunately this dynamic assertion complicates other aspects of the system. Specifically,

- How do same assertions interact with casts? For instance  $\{1 :: Bool \sim_{k,o,\ell} 2 :: Bool\}$ .
- How do sameness assertions cast check? This is difficult, because there is no requirement that a user asserted equality is of the same type. For instance what type should the term  $\{1 \sim_{k,o,\ell} True\}$  have?

Since there will only ever be a bounded number of assumptions, we can give each assumption a unique index  $k$  and consider all computations and judgments point-wise for every different combinations of  $ks$ . We will extend the notion of cast so different casts are possible for every assignment of  $ks$  in scope. So

$$\{1 :: Bool \sim_{k,o,\ell} 2 :: Bool\} \rightsquigarrow_* \{1 \sim_{k,o,\ell} 2 :: Bool\} :: \begin{matrix} k = left, Bool \\ k = left, Bool \end{matrix} \rightsquigarrow_* \{1 \sim_{k,o,\ell} 2\} :: \begin{matrix} k = right, Bool \\ k = left, Bool \end{matrix} = \{1 \sim_{k,o,\ell} 2\} :: Bool$$

where we allow syntactic sugar to summarize the cast when they are the same over all branches (2).

We will also index typing judgments by the choice of  $k$  in scope so that,  $k = left \vdash \{1 \sim_{k,o,\ell} True\} : Nat$  and  $k = right \vdash \{1 \sim_{k,o,\ell} True\} : Bool$ .

To control the assumptions in scope, in a closed term every  $k$  will be bound in the  $C$  expression of an  $Assert_{k \Rightarrow C}$  path.

Now we must consider how patterns would evaluate under assumptions. The original inspiration was to allow abstraction over casts as represented by a path, but now casts contain a bundle of paths each indexed by assumption. Luckily, we can maintain the operational behavior by allowing uniform substitution into path variables.

For simplicity of formalization we will require that  $kmaps$  always uniquely map every  $k$  index in scope. We will also assume that  $kcast$  handles all possible mappings of  $ks$  in scope.

Substitution is outlined in table 3. The function  $[-]_{k=}$  filters a term along  $k$  taking it out of scope. For instance

$$[\{7 \sim_{k,o,\ell} True\}]_{k=left} = 7 \text{ and } \left[ 3 :: \begin{matrix} k = left & j = left & Bool \\ k = left & j = right & Nat \\ k = right & j = left & String \\ k = right & j = right & Unit \end{matrix} \right]_{k=right} = 3 :: \begin{matrix} j = left & String \\ j = right & Unit \end{matrix} . \text{ The}$$

function  $[-]^k$  puts an assumption  $k$  into scope extending it in every cast. For instance,  $\left[ 3 :: \begin{matrix} j = left & String \\ j = right & Unit \end{matrix} \right]^k =$

$$3 :: \begin{matrix} j = left & k = left & String \\ j = left & k = right & String \\ j = right & k = left & Unit \\ j = right & k = right & Unit \end{matrix} \text{ and } [!x_p]^k =! \begin{matrix} k = left & x_p \\ k = right & x_p \end{matrix} . \text{ The subscript } kcast_{kmap} \text{ selects the appropriate}$$

assumption from the  $kcast$ . For instance  $\begin{matrix} k = left & j = left & x_p \\ k = left & j = right & refl \\ k = right & j = left & y_p \\ k = right & j = right & x_p \end{matrix} \xrightarrow{k=right, j=left} y_p$ . Since we have assumed that every choice in scope is handled, this result always exists.

### 3 Cast Value, Blame, and Reductions

We need to extend the notion of value, Blame and call-by-value reduction from Chapter 3.

<sup>1</sup>it would also be possible to extend the system with meta variables, though this seems harder to formalize

$\text{case } \overline{b}, \left\{ \overline{[patc \Rightarrow c] [patc' \Rightarrow !_\ell]} \right\}$ $\begin{array}{l} !_{kcast} \\ b :: kcast \\ \{b \sim_{k,o,\ell} c\} \\ \dots \\ x_p \\ \text{Assert}_{k \Rightarrow C} \\ \overline{p \circ} \\ cong_{y \Rightarrow b} p \\ p^{-1} \\ inTC_i p \\ inC_i p \\ \overline{kmap, p}; \end{array}$	$\begin{array}{l} [x := a] = \\ [x := a] = \\ [x := a] = \\ [x := a] = \\ \{b [x := [a]_{k=left}] \sim_{k,o[x:=a],\ell} c [x := [a]_{k=right}]\} \\ \dots \\ [x := a] = \\ [x := a] = \\ [x := a] = \\ [x := a] = \\ [x := a] = \\ [x := a] = \\ [x := a] = \\ [x := a] = \end{array}$	$\text{case } \overline{b [x := a]}, \left\{ \overline{[patc \Rightarrow c [x := a]] [patc' \Rightarrow !_\ell]} \right\}$ $\begin{array}{l} !_{kcast[x:=a]} \\ b [x := a] :: kcast [x := a] \\ \{b [x := [a]_{k=left}] \sim_{k,o[x:=a],\ell} c [x := [a]_{k=right}]\} \\ \dots \\ x_p \\ \text{Assert}_{k \Rightarrow C[x := [a]^k]} \\ \overline{p [x := a] \circ} \\ cong_{y \Rightarrow b[x:=a]} p [x := a] \\ p [x := a]^{-1} \\ inTC_i (p [x := a]) \\ inC_i (p [x := a]) \\ \overline{kmap, p [x := [a]_{kmap}]}; \end{array}$
$\text{case } \overline{a}, \left\{ \overline{[patc \Rightarrow c] [patc' \Rightarrow !_\ell]} \right\}$ $\begin{array}{l} !_{kcast} \\ a :: kcast' \\ \{b \sim_{k,o,\ell} c\} \\ \dots \\ \overline{kmap, p}; \end{array}$	$\begin{array}{l} [x_p := p] = \\ [x_p := kcast] = \\ [x_p := kcast] = \\ [x_p := kcast] = \\ [x_p := kcast] = \\ \dots \\ [x_p := kcast] = \end{array}$	$\text{case } \overline{a [x_p := kcast]}, \left\{ \overline{[patc \Rightarrow c [x_p := kcast]] [patc' \Rightarrow !_\ell]} \right\}$ $\begin{array}{l} !_{kcast[x_p:=kcast]} \text{ (issue)} \\ a [x_p := kcast] :: kcast' [x_p := kcast] \\ \{b [x := [kcast]_{k=left}] \sim_{k,o[x:=a],\ell} c [x := [kcast]_{k=right}]\} \\ \dots \\ \overline{kmap, p [x_p := kcast_{kmap}]}; \end{array}$

Figure 3: Cast Language Data Sub

$$\begin{array}{c}
\frac{a \text{ whnf} \quad b \text{ whnf} \quad \text{head } a \neq \text{head } b}{\text{Blame } \ell \text{ o } \{a \sim_{k, \text{o}, \ell} b\}} \\
\\
\frac{\bar{a} \text{ Match } \overline{\text{patc}'_j}}{\text{Blame } \ell \text{ in } \text{Ex}_{\overline{\text{patc}'_j}}[\bar{a}] \text{ case } \bar{a}, \left\{ \overline{|\text{patc} \Rightarrow b| \overline{\text{patc}' \Rightarrow!_{\ell_j}}} \right\}} \\
\\
\frac{\bar{a} \text{ !Match } \ell \text{ o } \overline{|\text{patc}'_j|}}{\text{Blame } \ell \text{ o case } \bar{a}, \left\{ \overline{|\text{patc} \Rightarrow -|} \right\}} \\
\\
\frac{!_{k\text{cast}} \quad \text{Blame } \ell \text{ o } k\text{cast}}{\text{Blame } \ell \text{ o } !_{k\text{cast}}}
\end{array}$$

Figure 4: Selection of Cast Language Blame

$$\frac{\text{Blame } \ell o p}{(d' \overline{patc}) :: p \text{ !Match } \ell o (d \overline{patc}) :: p}$$

This figure is not very helpful?

Figure 5: Cast Language Blame

Blame the blame conditions of Chapter 3 are simplified via the sameness assertion<sup>4</sup>. There are two new sources of blame from the case construct. The cast language records every “unmatched” branch and if a scrutinee hits one of those branches the case will be blamed for in-exhaustiveness<sup>2</sup>. If a scrutinee list primitively contradicts the pattern coverage via the **!Match** judgment blame will be extracted from the scrutinee. Since our type system will ensure complete coverage (based only on constructors) if a scrutinee escapes the complete pattern match in an empty context, it must be that there was a cast blamable cast to the head constructor. We have elided most of the structural rules that extract blame from terms, paths, and casts. We have left the structural rule for explicit blame for emphasis.

The value forms of the language must also be extended, values are presented in 6. As before type level values must have all type level casts reduced, term level values are allowed casts as long as they have plausible head form.

This extension to the syntax induces many more reduction rules. We include a summary of selected reduction rules in 7. We do not show the value restrictions to avoid clutter<sup>3</sup>. The important properties of reduction are

- Paths reduce into a stack of zero or more  $Assert_{k \Rightarrow A} S$
- Sameness assertions emit observably consistent constructors, and record the needed observations
- Sameness assertions will get stuck on inconsistent constructors
- Casts can commute out of sameness assertions with proper index tracking
- function application can commute around  $kcasts$ , similar to Chapter 3, but will keep  $k$  assumptions properly indexed

Matching is defined in 8. Note that uncast terms are equivalent to refl cast terms.

double check paths are fully applied when needed

The Cast language extension defined in this chapter is now fairly complex. Though all the meta-theory of this section is plausible, we have not fully formalized it in Coq, and there is a potential that some undetected errors exist. To be as clear as possible about the slight uncertainty around the meta-theory proposed in this chapter, I will list what would normally be considered theorems and lemmas as conjectures and postulates.

**Postulate** there is a suitable definitional equality  $\equiv$ , overloaded to all syntactic constructs, such that

- $\equiv$  is an equivalence
- $a \rightsquigarrow_* b$  and  $a' \rightsquigarrow_* b$  implies  $a \equiv a'$
- $p \rightsquigarrow_* q$  and  $p' \rightsquigarrow_* q$  implies  $p \equiv p'$
- $kcast \rightsquigarrow_* kcast''$  and  $kcast' \rightsquigarrow_* kcast''$  implies  $kcast \equiv kcast''$
- if  $\text{Head } a \neq \text{Head } b$  then  $a \not\equiv b$
- if  $kmap$  and  $kmap'$  have consistent assignments then  $kmap \equiv kmap'$
- 

weird place  
note. add  
or back m

settle on a  
for head

supports s

## 4 Cast System and Pathing

The cast system needs to maintain the consistency of well cast terms and also well typed paths. But unlike in Chapter 3 each judgment indexed by choices of  $k$ .

$$\overline{\star \mathbf{Val}}$$

$$\overline{(x : A) \rightarrow B \mathbf{Val}}$$

$$\frac{kcast \mathbf{Val} \quad \forall Assert_{k \Rightarrow (x:A) \rightarrow B} \in kcast}{(fun f x \Rightarrow a) :: kcast \mathbf{Val}}$$

clean up notation above

$$\frac{\bar{a} \mathbf{Val}}{D \bar{a} \mathbf{Val}}$$

$$\frac{\bar{a} \mathbf{Val} \quad kcast \mathbf{Val} \quad \exists D. \forall Assert_{k \Rightarrow D \bar{b}} \in kcast}{(d \bar{a}) :: kcast \mathbf{Val}}$$

clean up notation above

$$\frac{C \mathbf{Val} \quad C \neq \star}{Assert_{k \Rightarrow C} \mathbf{Val}}$$

$$\frac{p \mathbf{Val} \quad q \mathbf{Val}}{p \circ q \mathbf{Val}}$$

$$\overline{kmap, refl \mathbf{Val}}$$

$$\frac{p \mathbf{Val}}{kmap, p \mathbf{Val}}$$

$$\frac{\forall kmap, p \in kcast, kmap, p \mathbf{Val}}{kcast \mathbf{Val}}$$

type and term constructors need not be fully applied.

Figure 6: Selection of Cast Language Values



path reductions

$$\begin{array}{c}
\overline{\text{Assert}_{\rightarrow \star} \rightsquigarrow \text{refl}} \\
\\
\overline{\text{cong}_{x \Rightarrow a} (\text{Assert}_{k \Rightarrow c}) \rightsquigarrow \text{Assert}_{k \Rightarrow a[x:=c]}} \\
\\
\overline{\text{cong}_{x \Rightarrow a} \text{refl} \rightsquigarrow \text{refl}} \\
\\
\overline{\text{cong}_{x \Rightarrow a} (p \circ q) \rightsquigarrow (\text{cong}_{x \Rightarrow a} p) \circ (\text{cong}_{x \Rightarrow a} q)} \\
\\
\overline{(\text{Assert}_{k \Rightarrow C})^{-1} \rightsquigarrow \text{Assert}_{k \Rightarrow \mathbf{Swap}_k C}} \\
\\
\overline{\text{refl}^{-1} \rightsquigarrow \text{refl}} \\
\\
\overline{(q \circ p)^{-1} \rightsquigarrow p^{-1} \circ q^{-1}} \\
\\
\overline{\text{inTC}_i (p \circ \text{Assert}_{k \Rightarrow D\bar{a}}) \rightsquigarrow \text{inTC}_i (p) \circ \text{Assert}_{k \Rightarrow a_i}} \\
\\
\overline{\text{inTC}_i (\text{refl}) \rightsquigarrow \text{refl}} \\
\\
\overline{\text{inC}_i (p \circ \text{Assert}_{k \Rightarrow (d\bar{a})::\text{kcast}}) \rightsquigarrow \text{inC}_i (p) \circ \text{Assert}_{k \Rightarrow a_i}}
\end{array}$$

given suitable condition on the kcast

$$\overline{\text{inC}_i (\text{refl}) \rightsquigarrow \text{refl}}$$

structural rules

$$\begin{array}{c}
\overline{C \rightsquigarrow C'} \\
\overline{\text{Assert}_{k \Rightarrow C} \rightsquigarrow \text{Assert}_{k \Rightarrow C'}} \\
\\
\frac{q \rightsquigarrow q'}{p \circ q \rightsquigarrow p \circ q'} \\
\\
\frac{p \rightsquigarrow p'}{p \circ q \rightsquigarrow p' \circ q}
\end{array}$$

assumption reductions

$$\begin{array}{c}
\overline{p \rightsquigarrow p'} \\
\frac{\text{kcast} \quad \text{kcast}}{\overline{\text{kin}, p; \rightsquigarrow \text{kin}, p';}} \\
\text{kcast}' \quad \text{kcast}' \\
\\
\overline{a \rightsquigarrow a'} \\
\overline{a :: \text{kcast} \rightsquigarrow a' :: \text{kcast}} \\
\\
\overline{\text{kcast} \rightsquigarrow \text{kcast}'} \\
\overline{a :: \text{kcast} \rightsquigarrow a :: \text{kcast}'}
\end{array}$$

term reductions

$$\frac{p \rightsquigarrow p'}{!_p \rightsquigarrow !_p}$$

$$\overline{\left\{ a :: \frac{\text{kcast}}{\overline{\text{kin}, p \circ \text{Assert}_{k \Rightarrow C};}} \rightsquigarrow_{k,o,\ell} b \right\} \rightsquigarrow \left\{ a :: \frac{\text{kcast}}{\overline{\text{kin}, p;}} \rightsquigarrow_{k,o,\ell} b \right\} :: \overline{\text{kin}, k} = \text{left Assert}_{k \Rightarrow C};}$$

could remove more then just assertions

swap, pattern on the left?

record substitutions

$$\begin{array}{c}
\overline{a \text{ Match } x} \\
\\
\frac{\overline{\bar{a} \text{ Match } \overline{patc}}}{d \bar{a} \text{ Match } (d \overline{patc}) :: x_p} \\
\\
\frac{\overline{\bar{a} \text{ Match } \overline{patc}}}{(d \bar{a}) :: kcast \text{ Match } (d \overline{patc}) :: x_p} \\
\\
\frac{\overline{\bar{a} \text{ Match } \overline{patc}}}{(d \bar{a}) :: kcast \text{ Match } (d \overline{patc}) :: x_p} \\
\\
\overline{\cdot \text{ Match } \cdot} \\
\\
\frac{b \text{ Match } patc' \quad \bar{a} \text{ Match } \overline{patc}}{\bar{b} \bar{a} \text{ Match } patc' \overline{patc}}
\end{array}$$

substitution abbreviation

$$\begin{aligned}
- [(d \overline{patc}) :: x_p := d \bar{a}] &= - [\overline{patc} := \bar{a}] [x_p := refl] \\
- [(d \overline{patc}) :: x_p := (d \bar{a}) :: kcast] &= - [\overline{patc} := \bar{a}] [x_p := kcast]
\end{aligned}$$

Figure 8: Cast Language Matching

The typing and pathing judgments are listed in 9. Pathing judgments record the endpoint of paths with the  $\approx$  symbol.

technically speaking, telescopes should generalize to the different syntactic classes

We now conjecture the core lemmas that could be used to prove cast soundness

**Conjecture** substitution of cast terms preserves cast  
equivalently the following rule is admissible

$$\frac{HK \vdash a : A \quad x : A \in H \quad HK \vdash b : B}{H [x := a] K \vdash b [x := a] : B [x := a]}$$

**Conjecture** substitution of typed path preserves type  
equivalently the following rule is admissible

$$\frac{HK \vdash p : a \approx a' \quad x_p : a \approx a' \in H \quad HK \vdash b : B}{H [x_p := p] K \vdash b [x_p := p] : B [x_p := p]}$$

**Conjecture** substitution of *kcasts* preserve cast  
equivalently the following rule is admissible

$$\frac{HK \vdash kcast_K : a \approx a' \quad x_p : a \approx a' \in H \quad HK \vdash b : B}{H [x_p := kcast_K] K \vdash b [x_p := kcast] : B [x_p := kcast]}$$

**Conjecture** substitution of cast terms preserves path endpoints  
equivalently the following rule is admissible

$$\frac{HK \vdash a : A \quad x : A \in H \quad HK \vdash p : b \approx b'}{H [x := a] K \vdash p [x := a] : b [x := a] \approx b' [x := a]}$$

<sup>2</sup>This runtime error is conventional in ML style languages, and is even how Agda handles incomplete matches

<sup>3</sup>there are also multiple ways to lay them out. For instance we could evaluate paths left to right or right to left.

$$\frac{x_p : A \approx A' \in H}{HK \vdash x_p : A \approx A'}$$

$$\frac{HK, k = left \vdash a : A \quad HK, k = right \vdash a : A'}{HK \vdash Assert_{k \Rightarrow a} : [a]_{k=left} \approx [a]_{k=right}}$$

does  $A=A'$ ?

$$\frac{HK \vdash p : A \approx B \quad HK \vdash q : B \approx C}{HK \vdash p \circ q : A \approx C}$$

$$\frac{HK \vdash p : b \approx b \quad H, K \vdash b : B \quad H, K \vdash b' : B \quad H, x : B, K \vdash A}{HK \vdash cong_{x \Rightarrow a} p : a[x := b] \approx a[x := b']}$$

$$\frac{HK \vdash p : A \approx B}{HK \vdash p^{-1} : B \approx A}$$

$$\frac{HK \vdash p : (D\bar{a}) :: kcast \approx (D\bar{b}) :: kcast'}{HK \vdash inTC_i p : a_i \approx b_i}$$

and fully applied!

$$\frac{HK \vdash p : (d\bar{a}) :: kcast \approx (d\bar{b}) :: kcast'}{HK \vdash inC_i p : a_i \approx b_i}$$

and fully applied!

possibly force a matching type? but then it is unclear what type the conclusion should have

$$\frac{a' \equiv a \quad b' \equiv b \quad HK \vdash p : a \approx b}{HK \vdash p : a' \approx b'}$$

will need to adjust this if moves to a typed conversion rule

$$\frac{k = left \in K \quad HK \vdash a : A}{HK \vdash \{a \sim_{k,o,\ell} b\} : A}$$

$$\frac{k = right \in K \quad HK \vdash b : B}{HK \vdash \{a \sim_{k,o,\ell} b\} : B}$$

$$\frac{HK \vdash a : A \quad HK \vdash kcast_K : A \approx B}{HK \vdash a :: kcast : B}$$

$$\frac{\begin{array}{c} HK \vdash \bar{a} : \Delta \\ H, \Delta, K \vdash B : \star \\ \forall i (HK \vdash \overline{pat}_i : \Delta \quad H, (\overline{pat}_i : \Delta) K \vdash b_i : B) \\ \forall j (HK \vdash \overline{pat}_j : \Delta) \\ HK \vdash \overline{patc} \overline{patc'} : \Delta \text{ complete} \end{array}}{HK \vdash \text{case } \bar{a}, \left\{ \overline{patc}_i \Rightarrow b_i \mid \overline{patc}'_j \Rightarrow !_\ell \right\} : M[\Delta := \bar{a}]}$$

pattern expansion and pattern on context may need further exposition

$$\frac{\begin{array}{c} HK \vdash C : \star \\ HK \vdash kcast_K : a \approx a' \\ HK \vdash a : A \quad HK \vdash a' : A \\ \text{Head}(a) \neq \text{Head}(a') \end{array}}{HK \vdash !_{kcast} : C}$$

REMOVE TYPE RESTRICTION? the extra type restrictions is intended to force blame to the type level when needed, though this will not (cannot?) be invariant over reduction of paths in general

Figure 9: Cast Language typing and pathing rules

**Conjecture** substitution of typed paths preserves path endpoints  
equivalently the following rule is admissible

$$\frac{HK \vdash p : a \approx a' \quad x_p : a \approx a' \in H \quad HK \vdash q : b \approx b'}{H[x_p := p] K \vdash q[x_p := p] : b[x_p := p] \approx b'[x_p := p]}$$

**Conjecture** substitution of *kcasts* preserve cast  
equivalently the following rule is admissible

$$\frac{HK \vdash kcast_K : a \approx a' \quad x_p : a \approx a' \in H \quad HK \vdash b : B}{H[x_p := kcast_K] K \vdash q[x_p := kcast_K] : b[x_p := kcast_K] \approx b'[x_p := kcast_K]}$$

Finally we will conjecture the cast soundness.

**Conjecture** The cast system preserves types and path endpoints over normalization

**Conjecture** a well typed path in an empty context is a value, takes a step, or produces blame

**Conjecture** A well typed term in an empty context is a value, takes a step, or produces blame

## Part II

# 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 path 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.

## 5 Preliminaries

As mentioned in the introduction we will need to add and remove justified casts from the endpoints of arguments. For instance, we will need to be able to generate  $3 \approx x :: Nat$  from  $3 \approx x, x : X$ , and  $X \approx Nat$ . Fortunately the language is already expressive enough to embed these operations using a *Assert* that does not bind a same assertion.

We will specify the shorthand  $CastRap = Assert_{k \Rightarrow a :: k=right, p} : a \approx a :: p$ , similarly we can define  $CastL$ .

We can use a similar construction to remove casts from an endpoint. Given a path  $p : a :: q \approx b$ , we can define the macro  $UnCastRap = Assert_{k \Rightarrow a :: k=right, q} \circ p : a \approx b$ , similarly for  $UnCastL$ .

The surface language needs to be enriched with additional location metadata at each position where the two bidirectional typing modalities would cause a check in the surface language.

$$\begin{array}{lcl} m... & ::= & ... \\ & | & \text{case } \overline{n_{\ell}}, \left\{ \overline{pat \Rightarrow m_{\ell'}} \right\} \quad \text{data elim. without motive} \\ & | & \text{case } \overline{n_{\ell}}, \langle \overline{x \Rightarrow M_{\ell'}} \rangle \left\{ \overline{pat \Rightarrow m_{\ell''}} \right\} \quad \text{data elim. with motive} \end{array}$$

The implementation allows additional annotations along the motive, while this works within the bidirectional framework. The syntax is not presented here since the theory is already quite complicated.

where

## 6 Elaboration

"can", "could", weasel words until implementation is finalized

The biggest extension to the elaboration procedure in Chapter 3 is the path relevant unification and the insertion of casts to simulate surface language pattern matching. The unification and casting processes both work without *k* assumptions in scope, simplifying the possible terms that may appear.

The elaboration procedure uses the extended unification procedure to determine the implied type and assignment of each variable. In the match body casts are made so that variables behave as if they have the types and assignments consistent with the surface language. The original casting mechanism is still active, so it is possible that after all the casting types still don't line up. In this case primitive casts are still made at their given location.

$$\begin{array}{c}
\overline{U(\emptyset, \emptyset)} \\
\\
\frac{U(E, u) \quad a \equiv a'}{U(\{p : a \approx a'\} \cup E, u)} \\
\\
\frac{U(E[x := a], u[x := a])}{U(\{p : x \approx a\} \cup E, u \cup \{p : x \approx a\})}
\end{array}$$

actually a little incorrect, needs to use `conq` to concat the paths

$$\begin{array}{c}
\frac{U(E[x := a], u[x := a])}{U(\{p : a \approx x\} \cup E, u \cup \{p^{-1} : x \approx a\})} \\
\\
\frac{U(\{p : a \approx a'\} \cup E, u) \quad a \equiv d\bar{b} \quad a' \equiv d\bar{b}'}{U(\{Con_i p : b_i \approx b'_i\}_i \cup E, u)}
\end{array}$$

fully applied

$$\frac{U(\{p : a \approx a'\} \cup E, u) \quad a \equiv D\bar{b} \quad a' \equiv D\bar{b}'}{U(\{TCon_i p : b_i \approx b'_i\}_i \cup E, u)}$$

fully applied

$$\begin{array}{c}
\frac{U(\{p : a :: q \approx a'\} \cup E, u)}{U(\{uncastLp : a \approx a'\}_i \cup E, u)} \\
\\
\frac{U(\{p : a \approx a' :: q\} \cup E, u)}{U(\{uncastRp : a \approx a'\} \cup E, u)}
\end{array}$$

fully applied

break cycle, make sure x is assignable

double check constraint order

correct vars in 4a

Figure 10: Surface Language Unification

add explicit rules for elaboration?

The elaboration algorithm is extremely careful to only add casts, this means erasure is preserved and evaluation will be consistent with the surface language.

Further I conjecture the remaining properties from Chapter 3 hold

- **Conjecture** that every term well typed in the bidirectional surface language elaborates
- **Conjecture** blame never points to something that checked in the bidirectional system

## 7 Discussion and Future Work

### 7.1 Blame is not tight

Though the meta theory in this section is plausible, there are some awkward allowed behaviors. Blame may not be able to zero in as precisely as it seems is possible, when an assumption interacts with itself. For instance take the term under assumption  $k$ ,

$\{\lambda x \Rightarrow x \sim_{k,o,\ell} \lambda x \Rightarrow x\} \{1 \sim_{k,o,\ell} 2\} \rightsquigarrow_* (\lambda x \Rightarrow \{x \sim_{k,o,\text{app}[x],\ell} x\}) \{1 \sim_{k,o,\ell} 2\} \rightsquigarrow_* \{2 \sim_{k,o,\text{app}[\{1 \sim_{k,o,\ell} 2\}],\ell} 0\}$   
 which will mistakenly give blame to the function when it is more reasonable to blame the argument. This situations is more complicated if we want to avoid blame when the two sides are mutually consistent  $\{\lambda x \Rightarrow x \sim_{k,o,\ell} \lambda x \Rightarrow \text{Not } x\} \{true \sim_{k,o,\ell} false\} \rightsquigarrow_* (\lambda x \Rightarrow \{x \sim_{k,o,\text{app}[x],\ell} \text{Not } x\}) \{true \sim_{k,o,\ell} false\} \rightsquigarrow_* \{true \sim_{k,o,\text{app}[\{true \sim_{k,o,\ell} false\}],\ell} true\}$

### 7.2 Types invariance along paths

It turns out that the system defined in Chapter 3 had the advantage of only dealing with equalities in the type universe. Extending to equalities over arbitrary type has vastly increased the complexity of the system. To make the system work paths are untyped, which seems at least inelegant. There is nothing currently preventing blame across type. For instance,

$\{1 \sim_{k,o,\ell} false\}$  will generate blame  $1 \neq false$ . While blame of  $Nat \neq Bool$  will certainly result in a better error message. Several attempts were made to encode the type into the type assumption, but the resulting systems quickly became too complicated to work with. Some vestigial typing constraints are still in the system (such as on the explicit blame) to encourage this cleaner blame.

### 7.3 Elaboration is non-deterministic with regard to blame

Consider the case

```
case x <_:Id Nat 2 2 => S 2> {
| refl _ a => s a
}
```

that can elaborate to

```
case x <_:Id Nat 2 2 => S 2> {
| (refl A a)::p => (s (a::TCon0(p)) :: Cong uncastL(TCon1(p)))
}
```

where  $p : Id A a a \approx Id Nat 2 2$ , where  $TCon1p$  selects the first position  $p : Id A a a \approx Id Nat 2 2$ . But this could also have elaborated to

```
case x <_:Id Nat 2 2 => S 2> {
| (refl A a)::p => (s (a::TCon0(p)) :: Cong uncastL(TCon2(p)))
}
```

relying on  $p : Id A a a \approx Id Nat 2 2$ . This can make a difference if the scrutinee is

$refl Nat 2 :: Id Nat 3 2 :: Id Nat 2 2$

in one case blame will be triggered, in the other it will not. In this case it is possible to mix the blame from both positions, though this does not seem to extend in general since the consequences of inequality are undecidable in general and we intend to allow running programs if they can maintain their intended types.

## 7.4 Extending to Call-by-Value

As in chapter 3, the system presented here does the minimal amount of checking to maintain type safety. This can lead to unexpected results, for instance consider the surface term

```
case (refl Nat 7 :: Id Nat 2 2) <_ => Nat> {  
| refl _ a => a  
}
```

This will elaborate into

```
case (refl Nat 7 :: Id Nat 2 2) <_ => Nat> {  
| (refl A a)::p => a::TCon0(p)  
}
```

which will evaluate to  $7 :: \text{Nat}$  without generating blame. And indeed we only ever asserted that the result was of type  $\text{Nat}$ .

In the implementation, some of this behavior is avoided by requiring type arguments in a cast be run call-by-value. This restriction will blame  $7 \neq 2$  before the cast is even evaluated.

## 7.5 Efficiency

The system defined here is brutally inefficient.

In theory the system has an arbitrary slow down. As in Chapter 3, a cast that relies on non-terminating code can itself cause additional non-termination as paths are resolved.

As written there are many redundant computations, and trying every combination of assumptions is very inefficient. Currently the implementation is quite slow, though there are several ways to speed things up. Caching redundant computation would help. Having a smarter embedding of  $k$  assignments would remove redundant work directly. To some extent Cong and Assert can be made multi-arity to allow fewer jumps. But most helpful of all would be simplifying away unneeded casts. More advanced options include using proof search to find casts that will never cause an error.

parametricity

relation to fun-ext

warnings

## 7.6 Relation to UIP

Pattern matching as outlined in the last Chapter (which follows from [Coq92]) implies the **uniqueness of identity proofs** (UIP)<sup>4</sup>. UIP states that every proof of identity is equal to  $\text{refl}$  (and thus unique), and is not provable in many type theories. In univalent type theories UIP is directly contradicted by the “non-trivial” equalities, required to equate isomorphisms and  $\text{Id}$ . UIP is derivable in the surface language by following pattern match

```
case x <pr : Id A a a => Id (Id A a a) pr (refl A a) > {  
| refl A a => refl (Id A a a) (refl A a)  
}
```

This type checks since unification will assign  $pr := \text{refl } A a$  and under that assumption  $\text{refl } (\text{Id } A a a) (\text{refl } A a) : \text{Id } (\text{Id } A a a) (\text{refl } A a) (\text{refl } A a)$ . Like univalent type theories, the cast language has its own nontrivial equalities, so it might seem that the cast language would also contradict UIP. But it is perfectly compatible, and will elaborate. One interpretation is that though there are multiple “proofs” of identity, we don’t care which one is used.

interpreta  
aways?

## 7.7 Future work

Make equalities visible in the surface syntax

The system here has some simple inspiration that could be extended into pattern matching syntax more generally. It seems useful to be able to read equational information out of patterns, especially in settings with rich treatment

<sup>4</sup>Also called **axiom K**

of equality. Matching equalities directly could be a semi-useful feature in Agda, or in univalent type theories such as CTT.

## Part III

# Related work

previously published as an extended abstract in ...

CTT data is related?

## References

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83. Citeseer, 1992.

## Part IV

# TODO

## Todo list

“gradual correctness”?	1
Example, translate out to motive	3
extend H ctxs	4
Example back to pattern matching?	5
This figure is not very helpful?	7
double check paths are fully applied when needed	7
weird place to make this note. add it to the front or back matter?	7
settle on a capitalization for head	7
supports substitutivity	7
clean up notation above	8
clean up notation above	8
given suitable condition on the kcast	9
could remove more then just assertions	9
by shorthand	9
by shorthand	9
double check	9
swap, pattern on the left?	10
record substitutions	10
technically speaking, telescopes should generalize to the different syntactic classes	10
does $A=A'$ ?	11
and fully applied!	11
and fully applied!	11
possibly force a matching type? but then it is unclear what type the conclusion should have	11
will need to adjust this if moves to a typed conversion rule	11
pattern expansion and pattern on context may need further exposition	11
REMOVE TYPE RESTRICTION? the extra type restrictions is intended to force blame to the type level	
when needed, though this will not (cannot?) be invariant over reduction of paths in general	11
move note somewhere else	12
"can", "could", weasel words until implementation is finalized	12
add explicit rules for elaboration?	12
actually a little incorrect, needs to use conq to concat the paths	13
fully applied	13



fully applied . . . . .	13
fully applied . . . . .	13
break cycle, make sure x is assignable . . . . .	13
double check constraint order . . . . .	13
correct vars in 4a . . . . .	13
paremetricity . . . . .	15
relation to fun-ext . . . . .	15
warnings . . . . .	15
interpretation + take aways? . . . . .	15
Make equalities visible in the surface syntax . . . . .	15
CTT data is related? . . . . .	16
to stylize consistently, should use math font, or like a nice image . . . . .	18
break into smaller more relevant examples . . . . .	18
c? . . . . .	20
probly need to modify substution . . . . .	22

## 8 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.

## 9 unused

```

case x <pr : Id A a a => Id (Id A a a) pr (refl A a) > {
| (refl A' a') :: p =>
refl (((Id A')::(A -> A -> *)) (a'::A) (a'::A) ) :: (pr' : (Id A a a) -> Id (Id A a a) pr' pr')
(refl A')::((a : A) -> Id A a a) (a'::A)) ::
}

```

Where  $p : Id A' a' a' \approx Id A a a$ , ...

...

$$\begin{array}{c}
\frac{HK \text{ ok}}{HK \vdash \Diamond : .} \dots \\
\\
\frac{H, x : A; K \vdash \Delta \quad H; K \vdash A : \star \quad H; K \vdash patc : \Delta}{HK \vdash x, patc : (x : A) \Delta} \dots \\
\\
\frac{d : \Theta \rightarrow D\bar{b} \in H \quad HK \vdash \overline{patc'} : \Theta \quad H, (\overline{patc'} : \Theta), x_p : D\bar{b} \approx D\bar{a}, K \vdash patc : \Delta [x := d\overline{patc'} ::_{x_p}]}{HK \vdash d\overline{patc'} ::_{x_p}, patc : (x : D\bar{a}), \Delta} \dots
\end{array}$$

...

$$\begin{array}{c}
\frac{HK \vdash A : \star}{HK \vdash x : A} \dots \\
\\
\frac{\Gamma, x : M \vdash \Delta \quad \Gamma \vdash m : M \quad \Gamma \vdash \bar{n} [x := m] : \Delta [x := m]}{\Gamma \vdash m, \bar{n} : x : M, \Delta} \dots \\
\\
\frac{\Gamma \text{ ok} \quad \text{data } D \Delta \in \Gamma}{\Gamma \vdash D : \Delta \rightarrow *} \dots
\end{array}$$

```

-- 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

```

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

break into smaller more relevant examples

Figure 11: Cast Pattern Matching

$$\frac{\Gamma \mathbf{ok} \quad d : \Theta \rightarrow D\overline{m} \in \Gamma}{\Gamma \vdash d : \Theta \rightarrow D\overline{m}} \dots$$

...

$$\overline{HK \vdash x : A} \dots$$

$$\frac{H \vdash A : \star}{H \vdash \mathit{refl} : A \approx A}$$

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

ALT, would then need to resolve endpoint def equality

$$\frac{H \vdash a : A \quad H \vdash a' : A \quad H, x : A \vdash C : \star}{H \vdash \mathit{assert}_{\ell, (a=a':A), x \Rightarrow C} : C[x := a] \approx C[x := a']}$$

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

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

typing rules

$$\frac{H \vdash C : \star \quad H \vdash p : A \approx B \quad A \text{ and } B \text{ Disagree}}{H \vdash A \neq_p B : C}$$

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

ALT

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

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

...

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

ALT

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

$$\frac{\begin{array}{c} H \vdash \overline{a} : \Delta \\ H, \Delta \vdash B : \star \\ \forall i \ (H \vdash \mathit{Gen}(\overline{patc}_i : \Delta, \Theta) \quad \Gamma, \Theta \vdash m : M[\Delta := \overline{patc}_i]) \\ H \vdash \overline{patc} : \Delta \text{ complete} \end{array}}{\text{case } \overline{a}, \langle \overline{\Delta} \Rightarrow B \rangle \left\{ \overline{patc} \Rightarrow b \right\} : M[\Delta := \overline{n}]} \dots$$

Gen is defined as

$$\begin{array}{c} \overline{H \vdash \mathit{Gen}(\cdot, \cdot, \cdot)} \dots \\ \sim H \vdash A : \star \sim \\ \hline H \vdash \mathit{Gen}(x : (x : A), x : A) \dots \end{array}$$

$$\begin{array}{c}
\frac{\sim H \vdash A : \star \sim}{H \vdash \text{Gen}(x : A, x : A)} \cdots \\
\frac{d : \Theta \rightarrow D\bar{a} \in H \quad H \vdash \text{Gen}(\overline{\text{pat}_c} : \Theta, \Delta)}{H \vdash \text{Gen}(\overline{d\text{pat}_c} ::_{x_p} D\bar{b}, \Delta, x_p : D\bar{a} \approx D\bar{b})} \cdots \\
\frac{H \vdash \text{Gen}(\text{pat}_c : A, \Theta) \quad H, \Theta \vdash \text{Gen}(\overline{\text{pat}_c} : \Delta[x := \text{pat}_c], \Theta')}{H \vdash \text{Gen}(\text{pat}_c \overline{\text{pat}_c} : (x : A, \Delta), \Theta\Theta')} \cdots
\end{array}$$

other rules similar to the surface lang

observations,

o ::= ...  
| o.App[a] application  
| o.TCon[i] type cons. arg.  
| o.DCon[i] data cons. arg.

old style red rules

$$\overline{\text{rev}(pp') \rightsquigarrow (\text{rev } p')(\text{rev } p)}$$

$$\overline{\text{inTC}_i(pp') \rightsquigarrow (\text{inTC}_i p')(\text{inTC}_i p)}$$

$$\overline{\text{inC}_i(pp') \rightsquigarrow (\text{inC}_i p')(\text{inC}_i p)}$$

$$\overline{\text{inTC}_i \text{ refl} \rightsquigarrow \text{refl}}$$

$$\overline{\text{inC}_i \text{ refl} \rightsquigarrow \text{refl}}$$

$$\frac{\bar{a}_i = a' \quad \bar{c}_i = c' \quad \bar{b}_i = b'}{\text{inTC}_i(D\bar{a}_{\ell.D}\bar{c}D\bar{b}) \rightsquigarrow a'_{\ell.c'}b'}$$

$$\overline{\text{inC}_i((a :: A)_{\ell.c} b) \rightsquigarrow \text{inC}_i(a_{\ell.c} b)}$$

$$\overline{\text{inC}_i(a_{\ell.c}(b :: B)) \rightsquigarrow \text{inC}_i(a_{\ell.c} b)}$$

$$\overline{\text{inC}_i(a_{\ell.(c::C)} b) \rightsquigarrow \text{inC}_i(a_{\ell.c} b)}$$

$$\frac{\bar{a}_i = a' \quad \bar{c}_i = c' \quad \bar{b}_i = b'}{\text{inTC}_i(d\bar{a}_{\ell.d}\bar{c}d\bar{b}) \rightsquigarrow a'_{\ell.c'}b'}$$

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

$$\frac{a ::_{A,p A'_{\ell.C''} B', x.C} B \rightsquigarrow}{a ::_{A,p,x.C} C[x := A'] ::_{\ell.C[x := C'']} C[x := B']^c}$$

c?

$$\overline{(a ::_{A,p,x.C} C) \sim_{\ell_o} b \rightsquigarrow a \sim_{\ell_o} b}$$

$$\overline{a \sim_{\ell_o} (b ::_{B,p,x.C} C) \rightsquigarrow a \sim_{\ell_o} b}$$

...

path var,  
 $x_p$   
 assertion index,  
 $k$   
 assertion assumption,  
 $kin ::= k = left \mid k = right$   
 casts under assumption,  
 $kcast ::= \overline{kin, p};$   
 path exp.,  
 $p, p' ::=$ 

$x_p$	$Assert_{k \Rightarrow C}$	concrete cast
$refl$		
$pp'$		
$p^{-1}$		
$inTC_i p$		
$inC_i p$		
$uncastL_{kcast} p$		
$uncastR_{kcast} p$		

cast pattern,

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

cast expression,

$a \dots ::=$ 

$\dots$	$D$	type cons.
	$d$	data cons.
	$\text{case } \overline{a}, \{ \overline{patc \Rightarrow b} \mid \overline{patc' \Rightarrow !_\ell} \}$	data elim.
	$!_p$	force blame
	$a :: kcast$	cast
	$\{a \sim_{k, o, \ell} b\}$	assert same

observations,

$o ::=$ 

$\dots$		
$o.App[a]$		application
$o.TCon[i]$		type cons. arg.
$o.DCon[i]$		data cons. arg.

$$\frac{C \rightsquigarrow C'}{Assert_{k \Rightarrow C} \rightsquigarrow Assert_{k \Rightarrow C'}}$$

$$\overline{refl p \rightsquigarrow p}$$

$$\overline{p refl \rightsquigarrow p}$$

$$\overline{(qp)^{-1} \rightsquigarrow p^{-1} q^{-1}}$$

$$\frac{q \rightsquigarrow q' \quad p}{qp \rightsquigarrow q' p}$$

$$\frac{q \text{ Val } \quad p \rightsquigarrow p'}{qp \rightsquigarrow qp'}$$

$$\overline{(Assert_{k \Rightarrow C})^{-1} \rightsquigarrow Assert_{k \Rightarrow \mathbf{Swap}_k C}}$$

$$\overline{inTC_i (Assert_{k \Rightarrow d\overline{A}}) \rightsquigarrow Assert_{k \Rightarrow A_i}}$$

$$\overline{inC_i (Assert_{k \Rightarrow d\overline{A}}) \rightsquigarrow Assert_{k \Rightarrow A_i}}$$

TODO review this

$$\frac{\text{remove } k = \text{left casts} \quad a \text{ whnf}}{\text{uncastL} \left( \overline{\text{Assert}_{k \Rightarrow a :: \overline{\text{kin}, p};}} \right) \rightsquigarrow \text{Assert}_{k \Rightarrow a :: \overline{\text{kin}', p'};}}$$

probably need to modify substitution

$$\overline{\text{refl}^{-1} \rightsquigarrow \text{refl}}$$

$$\overline{\text{inTC}_i(\text{refl}) \rightsquigarrow \text{refl}}$$

$$\overline{\text{inC}_i(\text{refl}) \rightsquigarrow \text{refl}}$$

TODO review this

$$\overline{\text{uncastL}(\text{refl}) \rightsquigarrow ?}$$

term reductions

$$\frac{p \rightsquigarrow p'}{!_p \rightsquigarrow !_p'}$$

$$\overline{\left\{ a :: \overline{\text{kin}, p; \text{kin}, q} \text{Assert}_{k \Rightarrow C}; \overline{\text{kin}', p'}; \sim_{k, o, \ell} b \right\}} \rightsquigarrow \overline{\left\{ a :: \overline{\text{kin}, p; \text{kin}, q; \text{kin}', p'}; \sim_{k, o, \ell} b \right\}} :: \overline{\text{kin}, k = \text{left Assert}_{k \Rightarrow C};}$$

symetric around  $\sim$

$$\overline{\{\star \sim_{k, o, \ell} \star\}} \rightsquigarrow \star$$

$$\overline{\{(x : A) \rightarrow B \sim_{k, o, \ell} (x : A') \rightarrow B'\}} \rightsquigarrow \overline{(x : \{A \sim_{k, o, \text{arg}, \ell} A'\}) \rightarrow \{B \sim_{k, o, \text{bod}[x], \ell} B'\}}$$

$$\overline{\{\text{fun } f x \Rightarrow b \sim_{k, o, \ell} \text{fun } f x \Rightarrow b'\}} \rightsquigarrow \overline{\text{fun } f x \Rightarrow \{b \sim_{k, o, \text{app}[x], \ell} b'\}}$$

$$\overline{\{d\bar{a} \sim_{k, o, \ell} d\bar{a}'\}} \rightsquigarrow \overline{d\{a_i \sim_{k, o, o, DCon[i], \ell} a'_i\}}$$

$$\overline{\{D\bar{a} \sim_{k, o, \ell} D\bar{a}'\}} \rightsquigarrow \overline{D\{a_i \sim_{k, o, o, TCon[i], \ell} a'_i\}}$$

$$\overline{a :: \overline{\text{kin}, ;}} \rightsquigarrow a$$

$$\frac{\text{pointwise concatenation}}{(a :: \overline{\text{kin}, p};) :: \overline{\text{kin}', p'}; \rightsquigarrow \dots}$$

$$\left( \begin{array}{c} \dots \\ a :: \text{kin}, q \text{Assert}_{k \Rightarrow (x:A) \rightarrow B}; \\ \dots \end{array} \right) b \rightsquigarrow \left( \left( \begin{array}{c} \dots \\ a :: \text{kin}, q \text{Assert}_{k \Rightarrow (x:A) \rightarrow B}; \\ \dots \end{array} \right) (b :: \text{kin}, \text{Assert}_{k \Rightarrow \text{Swap}_k A};) \right) :: \text{kin}, \text{Assert}_{k \Rightarrow B[x := \dots]}$$

$$\frac{\text{Match } \bar{a} \text{ patc}_i}{\text{case } \bar{a}, \left\{ \overline{[\text{patc}_i \Rightarrow b_i] [\text{patc}' \Rightarrow !_\ell]} \rightsquigarrow b_i [\text{patc}_i := \bar{a}] \right\}}$$

...

$$\frac{p \text{ Val}}{q \circ \text{refl} \circ p \rightsquigarrow q \circ p}$$

$$\frac{p \text{ Val } q \text{ Val}}{(q \circ p)^{-1} \rightsquigarrow p^{-1} \circ q^{-1}}$$

$$\frac{q \rightsquigarrow q'}{p \circ q \rightsquigarrow p \circ q'}$$

$$\frac{q \text{ Val } p \rightsquigarrow p'}{p \circ q \rightsquigarrow p' \circ q}$$