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

Mark Lemay

December 8, 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 it's constructors. If every constructor is accounted for, only blameable data remains. Quantifying over casts allows blame to be redirected, so if the program gets stuck in a pattern branch it can blame the 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 a 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 $Vec\,A\,(Sn)$ 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 $Cons\,Nat\,3\,\ldots\,\ldots :: Vec\,Bool\,5$

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

```
case x {
| (Cons _ a _ _) :: p => a::InTc0(p)
| (Nil _ ) :: p => !InTc1(p)
}
```

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

1.2 Sum

Consider the more involved example that sums the first 2 numbers of a vector

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

the elaboration procedure will produce

```
case x {
| (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 : Vec Nat" (S n") = Vec Nat' n', p2 : Vec Nat' (S n') = Vec Nat 2, i:Nat', j:Nat"
  - this means we can deduce InTc0(p2) : Nat' = Nat, and InTc0(p1) InTc0(p2) : Nat" = Nat
- in the 2nd branch, p : Vec Nat' 0 = Vec Nat' 2
  - which is unsatisfiable, by InTc1(p) : 0 = 2
- in the 3rd branch, p1 : Vec Nat" 0 = Vec Nat' n', p2 : Vec Nat' (S n') = Vec Nat 2
  - which is unsatisfiable by InTc1(p1) 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

But 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 {
| (refl Nat' a)::p => (rep (a::InTc0(p))):: Cong (x=> (A : *) -> A -> Vec Nat x)
                                                 (CastL (InTC0(p)) (InTC1 p))
   Nat 9
}
```

since we have p: Id Nat' a a = Nat 2 2,

(a::InTc0(p)) casts a to a Nat but then we need to evidence that (A : *) -> A -> Vec Nat (a::InTc0(p)) = (A : *) -> A -> Vec Nat 2

this means we fist need congruence to select the portion of interest . Ideally we would have a path from (a::InTc0(p)) = 2, but we only have a path (InTc1(p)): a = 2 . we will use the meta operation $CastL$ to produce the path $CastL_{InTC_0(p)}\,(InTC_1 p) : (a :: InTc0(p)) \approx 2$.

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 $last : (n : Nat) \to Vec\,A\,(S\,N) \to A$ 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(TCon1(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 requires 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 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[1]. 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, $\{\lambda x \Rightarrow x + 1 \sim_{k,o,\ell} \lambda x \Rightarrow 1 + x\} \rightsquigarrow_* \lambda x \Rightarrow \{x + 1 \sim_{k,o.App[x],\ell} 1 + x\}$.

[1] it would also be possible to extend the system with meta variables, though this seems harder to formalize

path var,
x_p
assertion index,
k
assertion assumption,
$kin \quad ::= \quad k = left \mid k = right$
assertion maps,
$kmap \quad ::= \quad \overline{kin},$
casts under assumption,
$kcast \quad ::= \quad \overline{kmap, p};$
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,
$patc \quad ::= \quad x \mid \left(d\, \overline{patc}\right) :: x_p$
cast expression,

$a...$ ::=	...	
|	D	type cons.
|	d	data cons.
|	$\mathsf{case}\, \overline{a},\ \left\{ \overline{\mid \overline{patc} \Rightarrow b} \mid \overline{patc' \Rightarrow !_\ell} \right\}$	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 $refl$

$a :: kmap, p; kmap', q; ...$ written $a :: \begin{matrix} kmap, & p \\ kmap', & q \\ ... & ... \end{matrix}$

$a :: \begin{matrix} kmap, k = left, & p \\ kmap, k = right, & p \\ kmap', k = left, & q \\ kmap, k = right, & q \\ ... & ... \end{matrix}$ written $a :: \begin{matrix} kmap, & p \\ kmap', & q \\ ... & ... \end{matrix}$ when k is irrelevant

Figure 2: Surface Language Abbreviations

Since this equational construct is already needed for functions, we will use it to handle all equality assertions. For instance, $\{(\lambda x \Rightarrow S\, x)\, Z \sim_{k,o,\ell} 2+2\} \rightsquigarrow_* \{S\, Z \sim_{k,o,\ell} S\ (1+2)\} \rightsquigarrow_* S\ \{Z \sim_{k,o.DCon_0,\ell} 1+2\} \rightsquigarrow_* S\ \{Z \sim_{k,o.DCon_0,\ell} S\, 2\}$. 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

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

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 *kmap*s 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 $\lfloor - \rfloor_{k=-}$ filters a term along k taking it out of scope. For instance $\lfloor \{7 \sim_{k,o,\ell} True\} \rfloor_{k=left} = 7$ and $\left\lfloor 3 :: \begin{matrix} k=left & j=left & Bool \\ k=left & j=right & Nat \\ k=right & j=left & String \\ k=right & j=right & Unit \end{matrix} \right\rfloor_{k=right} = 3 :: \begin{matrix} j=left & String \\ j=right & Unit \end{matrix}$. The function $\lceil - \rceil^k$ puts an assumption k into scope extending it in every cast. For instance, $\left\lceil 3 :: \begin{matrix} j=left & String \\ j=right & Unit \end{matrix} \right\rceil^k = 3 :: \begin{matrix} j=left & k=left & String \\ j=left & k=right & String \\ j=right & k=left & Unit \\ j=right & k=right & Unit \end{matrix}$ and $\lceil !_{x_p} \rceil^k = ! \begin{matrix} k=left & x_p \\ k=right & x_p \end{matrix}$. The subscript $kcast_{kmap}$ selects the appropriate assumption from the *kcast*. For instance $\begin{matrix} k=left & j=left & x_p \\ k=left & j=right & refl \\ k=right & j=left & y_p \\ k=right & j=right & x_p \end{matrix}_{k=right,j=left} = y_p$. 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.

Blame the blame conditions of Chapter 3 are simplified via the sameness assertion4. 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 [2]. 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.

[2] This runtime error is conventional in ML style languages, and is even how Agda handles incomplete matches

$$
\begin{array}{rll}
\mathsf{case}\,\overline{b},\,\left\{\overline{|\,\overline{patc} \Rightarrow c}|\,\overline{patc' \Rightarrow !_\ell}\right\} & [x := a] = & \mathsf{case}\,\overline{b\,[x := a]},\,\left\{\overline{|\,\overline{patc} \Rightarrow c\,[x := a]}|\,\overline{patc' \Rightarrow !_\ell}\right\} \\
!_{kcast} & [x := a] = & !_{kcast[x:=a]} \\
b :: kcast & [x := a] = & b\,[x := a] :: kcast\,[x := a] \\
\{b \sim_{k,o,\ell} c\} & [x := a] = & \left\{b\left[x := \lfloor a \rfloor_{k=left}\right] \sim_{k,o[x:=a],\ell} c\left[x := \lfloor a \rfloor_{k=right}\right]\right\} \\
\ldots & \ldots & \ldots \\
x_p & [x := a] = & x_p \\
Assert_{k \Rightarrow C} & [x := a] = & Assert_{k \Rightarrow C[x:=\lceil a \rceil^k]} \\
\overline{p \circ} & [x := a] = & \overline{p\,[x := a] \circ} \\
cong_{y \Rightarrow b} p & [x := a] = & cong_{y \Rightarrow b[x:=a]} p\,[x := a] \\
p^{-1} & [x := a] = & p\,[x := a]^{-1} \\
inTC_i\,p & [x := a] = & inTC_i\ (p\,[x := a]) \\
inC_i\,p & [x := a] = & inC_i\ (p\,[x := a]) \\
\overline{kmap, p;} & [x := a] = & \overline{kmap, p\left[x := \lfloor a \rfloor_{kmap}\right];} \\
\ldots & & \ldots \\
x_p & [x_p := p] = & p \\
\mathsf{case}\,\overline{a},\,\left\{\overline{|\,\overline{patc} \Rightarrow c}|\,\overline{patc' \Rightarrow !_\ell}\right\} & [x_p := kcast] = & \mathsf{case}\,\overline{a\,[x_p := kcast]},\,\left\{\overline{|\,\overline{patc} \Rightarrow c\,[x_p := kcast]}|\,\overline{patc' \Rightarrow !_\ell}\right\} \\
!_{kcast} & [x_p := kcast] = & !_{kcast[x_p:=kcast]}\ \text{(issue)} \\
a :: kcast' & [x_p := kcast] = & a\,[x_p := kcast] :: kcast'\,[x_p := kcast] \\
\{b \sim_{k,o,\ell} c\} & [x_p := kcast] = & \left\{b\left[x := \lfloor kcast \rfloor_{k=left}\right] \sim_{k,o[x:=a],\ell} c\left[x := \lfloor kcast \rfloor_{k=right}\right]\right\} \\
\ldots & \ldots & \ldots \\
\overline{kmap, p;} & [x_p := kcast] = & \overline{kmap, p\,[x_p := kcast_{kmap}];}
\end{array}
$$

Figure 3: Cast Language Data Sub

$$
\frac{a\ \mathbf{whnf} \quad b\ \mathbf{whnf} \quad \mathbf{head}\,a \neq \mathbf{head}\,b}{\mathbf{Blame}\,\ell\,o\,\{a \sim_{k,o,\ell} b\}}
$$

$$
\frac{\overline{a}\,\mathbf{Match}\,\overline{patc'}_j}{\mathbf{Blame}\,\ell\,inEx_{\overline{patc'}_j}\,[\overline{a}]\,\mathsf{case}\,\overline{a},\,\left\{\overline{|\,\overline{patc} \Rightarrow b}|\,\overline{patc' \Rightarrow !_{\ell_j}}\right\}}
$$

$$
\frac{\overline{a}\ !\mathbf{Match}\ \ell\,o\,\overline{|\overline{patc'}_j}}{\mathbf{Blame}\,\ell\,o\,\mathsf{case}\,\overline{a},\,\left\{\overline{|\,\overline{patc} \Rightarrow -}\right\}}
$$

$$
\frac{!_{kcast} \quad \mathbf{Blame}\,\ell\,o\ kcast}{\mathbf{Blame}\,\ell\,o\,!_{kcast}}
$$

Figure 4: Selection of Cast Language Blame

$$
\frac{\mathbf{Blame}\,\ell\,o\ p}{\left(d'\ \overline{patc}\right) :: p\ !\mathbf{Match}\ \ell\,o\ \left(d\ \overline{patc}\right) :: p}
$$

This figure is not very helpful?

Figure 5: Cast Language Blame

$$\frac{}{\star\ \mathbf{Val}}$$

$$\frac{}{(x : A) \to B\ \mathbf{Val}}$$

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

clean up notation above

$$\frac{\overline{a}\ \mathbf{Val}}{D\,\overline{a}\ \mathbf{Val}}$$

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

clean up notation above

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

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

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

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

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

type and term constructors need not be fully applied.

Figure 6: Selection of Cast Language Values

path reductions

$$\overline{Assert_{-\Rightarrow\star} \rightsquigarrow refl}$$

$$\overline{cong_{x\Rightarrow a}\left(Assert_{k\Rightarrow c}\right) \rightsquigarrow Assert_{k\Rightarrow a[x:=c]}}$$

$$\overline{cong_{x\Rightarrow a} refl \rightsquigarrow refl}$$

$$\overline{cong_{x\Rightarrow a}\left(p \circ q\right) \rightsquigarrow \left(cong_{x\Rightarrow a} p\right) \circ \left(cong_{x\Rightarrow a} q\right)}$$

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

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

$$\overline{\left(q \circ p\right)^{-1} \rightsquigarrow p^{-1} \circ q^{-1}}$$

$$\overline{inTC_i\left(p \circ Assert_{k\Rightarrow D\overline{a}}\right) \rightsquigarrow inTC_i\left(p\right) \circ Assert_{k\Rightarrow a_i}}$$

$$\overline{inTC_i\left(refl\right) \rightsquigarrow refl}$$

$$\overline{inC_i\left(p \circ Assert_{k\Rightarrow (d\overline{a})::kcast}\right) \rightsquigarrow inC_i\left(p\right) \circ Assert_{k\Rightarrow a_i}}$$

given suitable condition on the kcast

$$\overline{inC_i\left(refl\right) \rightsquigarrow refl}$$

structural rules

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

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

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

assumption reductions

$$\frac{p \rightsquigarrow p'}{\begin{array}{ccc} kcast & & kcast \\ \overline{kin},p; & \rightsquigarrow & \overline{kin},p'; \\ kcast' & & kcast' \end{array}}$$

$$\frac{a \rightsquigarrow a'}{a :: kcast \rightsquigarrow a' :: kcast}$$

$$\frac{kcast \rightsquigarrow kcast'}{a :: kcast \rightsquigarrow a :: kcast'}$$

term reductions

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

$$\overline{\left\{\begin{array}{lll} a :: & kcast & \\ & \overline{kin},p \circ Assert_{k\Rightarrow C}; & \sim_{k,o,\ell} b \\ & kcast' & \end{array}\right\} \rightsquigarrow \left\{\begin{array}{lll} a :: & kcast & \\ & \overline{kin},p; & \sim_{k,o,\ell} b \\ & kcast' & \end{array}\right\} :: \overline{kin},k = left\, Assert_{k\Rightarrow C};}$$

could remove more then just assertions

$$\frac{}{a \ \mathbf{Match} \ x}$$

$$\frac{\overline{a} \ \mathbf{Match} \ \overline{patc}}{d\,\overline{a} \ \mathbf{Match} \ \left(d\,\overline{patc}\right) :: x_p}$$

$$\frac{\overline{a} \ \mathbf{Match} \ \overline{patc}}{(d\,\overline{a}) :: kcast \ \mathbf{Match} \ \left(d\,\overline{patc}\right) :: x_p}$$

$$\frac{\overline{a} \ \mathbf{Match} \ \overline{patc}}{(d\,\overline{a}) :: kcast \ \mathbf{Match} \ \left(d\,\overline{patc}\right) :: x_p}$$

$$\frac{}{. \ \mathbf{Match} \ .}$$

$$\frac{b \ \mathbf{Match} \ patc' \quad \overline{a} \ \mathbf{Match} \ \overline{patc}}{b\overline{a} \ \mathbf{Match} \ patc'\overline{patc}}$$

substitution abbreviation

$$-\left[\left(d\,\overline{patc}\right) :: x_p \coloneqq d\,\overline{a}\right] = -\left[\overline{patc} \coloneqq \overline{a}\right]\left[x_p \coloneqq refl\right]$$

$$-\left[\left(d\,\overline{patc}\right) :: x_p \coloneqq (d\,\overline{a}) :: kcast\right] = -\left[\overline{patc} \coloneqq \overline{a}\right]\left[x_p \coloneqq kcast\right]$$

Figure 8: Cast Language Matching

This extension to the syntax induces many more reduction rules. We include a summery of selected reduction rules in 7. We do not show the value restrictions to avoid clutter[3]. 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.

weird plac
note. add
or back m

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 **Head** $a \neq$ **Head** b then $a \not\equiv b$

settle on a
for head

- if $kmap$ and $kmap'$ have consistent assignments then $kmap \equiv kmap'$
-

supports s

[3]there are also multiple ways to lay them out. For instance we could evaluate paths left to right or right to left.

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.

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

does A=A'?

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

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

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

$$\frac{HK \vdash p : (D\,\overline{a}) :: kcast \;\approx\; \left(D\,\overline{b}\right) :: kcast'}{HK \vdash inTC_i\, p : a_i \approx b_i}$$

and fully applied!

$$\frac{HK \vdash p : (d\,\overline{a}) :: kcast \;\approx\; \left(d\,\overline{b}\right) :: kcast'}{HK \vdash inC_i\, p : a_i \approx b_i}$$

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

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

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

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

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

pattern expansion and pattern on context may need further exposition

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

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

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 $CastR_a p = Assert_{k \Rightarrow a :: k = right, p} : a \approx a :: p$, similarly we can define $CastL$.

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

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

where

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.

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.

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,

$\{\lambda x \Rightarrow x \sim_{k,o,\ell} \lambda x \Rightarrow x\}\,\{1 \sim_{k,o,\ell} 2\} \rightsquigarrow_* \left(\lambda x \Rightarrow \left\{x \sim_{k,o.app[x],\ell} x\right\}\right)\{1 \sim_{k,o,\ell} 2\} \rightsquigarrow_* \left\{2 \sim_{k,o.app[\{1 \sim_{k,o,\ell} 2\}],\ell} 0\right\}$
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 $\{\lambda x \Rightarrow x \sim_{k,o,\ell} \lambda x \Rightarrow Not\ x\}\,\{true \sim_{k,o,\ell}$
$\left(\lambda x \Rightarrow \left\{x \sim_{k,o.app[x],\ell} Not\ x\right\}\right)\{true \sim_{k,o,\ell} false\} \rightsquigarrow_* \left\{true \sim_{k,o.app[\{true \sim_{k,o,\ell} false\}],\ell} true\right\}$

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

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

$$\frac{U\left(E, u\right) \quad a \equiv a'}{U\left(\{p : a \approx a'\} \cup E, u\right)}$$

$$\frac{U\left(E\left[x \coloneqq a\right], u\left[x \coloneqq a\right]\right)}{U\left(\{p : x \approx a\} \cup E, u \cup \{p : x \approx a\}\right)}$$

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

$$\frac{U\left(E\left[x \coloneqq a\right], u\left[x \coloneqq a\right]\right)}{U\left(\{p : a \approx x\} \cup E, u \cup \{p^{-1} : x \approx a\}\right)}$$

$$\frac{U\left(\{p : a \approx a'\} \cup E, u\right) \quad a \equiv d\bar{b} \quad a' \equiv d\overline{b'}}{U\left(\left\{Con_i p : b_i \approx b_i'\right\}_i \cup E, u\right)}$$

fully applied

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

fully applied

$$\frac{U\left(\{p : a :: q \approx a'\} \cup E, u\right)}{U\left(\left\{uncastLp : a \approx a\right\}_i \cup E, u\right)}$$

$$\frac{U\left(\{p : a \approx a' :: q\} \cup E, u\right)}{U\left(\{uncastRp : a \approx a'\} \cup E, u\right)}$$

fully applied

break cycle, make sure x is assignable

double check constraint order

correct vars in 4a

Figure 10: Surface Language Unification

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 $TCon_1 p$ selects the first position $p : Id\, A\, \underline{a}\, a \approx Id\, Nat\, \underline{2}\, 2$. 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 $p : Id\, A\, a\, \underline{a} \approx Id\, Nat\, 2\, \underline{2}$. 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 $7 :: Nat$ without generating blame. And indeed we only ever asserted that the result was of type 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.

paremetricity

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)[4]. UIP states that every proof of identity is equal to 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 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 $pr := refl\ A\ a$ and under that assumption $refl\ (Id\ A\ a\ a)\ (refl\ A\ a) : Id\ (Id\ A\ a\ a)\ (refl\ A\ a)\ (refl\ A\ a)$. 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 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

chapter 4a additional equations al la ATS +trellis, allowing equality matching (based on typeclass?)

Todo list

[4] Also called **axiom k**

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)) ::

}
```

```
-- 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

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

...

$$\frac{H\,K\,\mathbf{ok}}{H\,K \vdash \Diamond : .}\ ...$$

$$\frac{H, x : A; K \vdash \Delta \quad H; K \vdash A : \star \quad H; K \vdash patc : \Delta}{H\,K \vdash x, patc\ :\ (x : A)\,\Delta}\ ...$$

$$\frac{\begin{array}{c} d\ :\ \Theta \to D\overline{b} \in H \\ H\,K \vdash \overline{patc'} : \Theta \\ H, \left(\overline{patc'} : \Theta\right), x_p : D\overline{b} \approx D\overline{a}, K \vdash patc : \Delta\left[x \coloneqq d\,\overline{patc'} ::_{x_p}\right] \end{array}}{H\,K \vdash d\,\overline{patc'} ::_{x_p}, patc\ :\ (x : D\overline{a})\,, \Delta}\ ...$$

...

$$\frac{H\,K \vdash A : \star}{H\,K \vdash x : A}\ ...$$

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

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

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

...

$$\frac{}{H\,K \vdash x : A}\ ...$$

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

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

ALT, would then need to resolve endpoint def equality

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

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

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

typing rules

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

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

ALT

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

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

...

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

ALT

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

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

Gen is defined as

$$\frac{}{H \vdash Gen\,(. : ., .)} \ldots$$

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

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

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

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

other rules similar to the surface lang

observations,

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

old style red rules

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

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

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

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

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

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

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

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

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

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

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

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

c?

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

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

...

path var,
x_p
assertion index,
k
assertion assumption,

$$kin ::= k = left \mid k = right$$

casts under assumption,

$$kcast ::= \overline{\overline{kin},p};$$

path exp.,

$$\begin{array}{lll} p, p' ::= & x_p & \\ \mid & Assert_{k \Rightarrow C} & \text{concrete cast} \\ \mid & refl & \\ \mid & pp' & \\ \mid & p^{-1} & \\ \mid & inTC_i\,p & \\ \mid & inC_i\,p & \\ \mid & uncastL_{kcast}\,p & \\ \mid & uncastR_{kcast}\,p & \end{array}$$

cast pattern,

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

cast expression,

$$\begin{array}{lll} a... ::= & ... & \\ \mid & D & \text{type cons.} \\ \mid & d & \text{data cons.} \\ \mid & \mathsf{case}\,\overline{a},\ \left\{\overline{\mid \overline{patc} \Rightarrow b}\ \overline{\mid \overline{patc'} \Rightarrow !_\ell}\right\} & \text{data elim.} \\ \mid & !_p & \text{force blame} \\ \mid & a :: kcast & \text{cast} \\ \mid & \{a \sim_{k,o,\ell} b\} & \text{assert same} \end{array}$$

observations,

$$\begin{array}{lll} o ::= & ... & \\ \mid & o.App[a] & \text{application} \\ \mid & o.TCon[i] & \text{type cons. arg.} \\ \mid & o.DCon[i] & \text{data cons. arg.} \end{array}$$

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

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

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

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

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

$$\frac{q\ \mathbf{Val} \quad p \rightsquigarrow p'}{q\,p \rightsquigarrow q\,p'}$$

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

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

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

TODO review this

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

probly need to modify substution

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

$$\overline{inTC_i\,(refl) \rightsquigarrow refl}$$

$$\overline{inC_i\,(refl) \rightsquigarrow refl}$$

TODO review this

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

term redcutions

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

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

symetric around $\sim$

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

$$\overline{\{(x : A) \to B \sim_{k,o,\ell} (x : A') \to B'\} \rightsquigarrow (x : \{A \sim_{k,o.arg,\ell} A'\}) \to \left\{B \sim_{k,o.bod[x],\ell} B'\right\}}$$

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

$$\overline{\left\{d\overline{a} \sim_{k,o,\ell} d\overline{a'}\right\} \rightsquigarrow d\overline{\left\{a_i \sim_{k,o.o.DCon[i],\ell} a_i'\right\}}}$$

$$\overline{\left\{D\overline{a} \sim_{k,o,\ell} D\overline{a'}\right\} \rightsquigarrow D\overline{\left\{a_i \sim_{k,o.o.TCon[i],\ell} a_i'\right\}}}$$

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

$$\frac{\textit{pointwise concatination}}{\left(a :: \overline{\overline{kin}, p;}\right) :: \overline{\overline{kin'}, p'}; \rightsquigarrow ...}$$

$$\left(\begin{array}{c} ... \\ a :: \quad kin, q\ Assert_{k \Rightarrow (x:A) \to B}; \\ ... \end{array}\right) b \rightsquigarrow \left(\left(\begin{array}{c} ... \\ a :: \quad kin, q\ Assert_{k \Rightarrow (x:A) \to B}; \\ ... \end{array}\right) \left(b :: kin, Assert_{k \Rightarrow \mathbf{Swap}_k A};\right)\right) :: kin, Assert_{k \Rightarrow B\left[x :=\right.}$$

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

...

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

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

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

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