Chapter 5 (draft): Plausible Symbolic Reduction

Mark Lemay

December 17, 2021

Not yet implemanted (earlier version was, but this contains substantial changes)

Part I

Symbolic Execution

One of the values of type checking is the immediacy of feedback. We have outlined here a system that will give warning messages immediacy, but requires evaluation to give the detailed error messages that are most helpful to correct a program. This is especially important if the user wants to use the system as a proof language, and will not generally execute their proofs. The symbolic evaluation system recaptures some of that quicker feed back, by specifying a system that passively tries to find errors.

This process is semi-decidable in general (by testing against all well typed syntax). But enumerating every term is infeasibly inefficient for type syntax and function syntax, So we will present a procedure that generates plausible partial functions and types. Additionally the search space can be reduced by only engaging with values of data type instead of all expressions. Finally, we present a scheme to handle path variables.

Since this procedure operates over the cast language, we must decide what constitutes a reasonable testing environment

- on one extreme, our testing code could introduce new blame that is exercised
- the testing code could introduce casts as long as no blame error occurs
- the testing code could insist that no blame is possible

we will attempt to use the middle ground criterion, since that will allow us to test with terms such as $Id\ (Nat \to Nat)\ f\ g$ as long as $fx = gx$ when relevant.

Since the procedure remains semi-decidable, we intend to run it incrementally, with whatever resources are available. The procedure is intended to be run during the continuous integration phase of standard software development.

We call our method plausible symbolic execution. Though plausible symbolic execution is an approximation (it is possible to flag errors that are not reachable in code) and is in some ways inefficient. It appears to be the first symbolic execution framework to deal with fully typed higher order functions, and certainly is the procedure of it's kind to deal with dependent types.

Part II

The functional fragment

1 Environments and Observations

The function fragment of the cast language defined in Chapter 3, already provides a rich language to consider for symbolic execution. Most of the issues can be highlighted in that setting. So we will first define symbolic execution for that fragment.

environment,

$I \;::=\; \Diamond$
$\quad|\quad I, X : \star$
$\quad|\quad I, X = \star$
$\quad|\quad I, X = (x : Y) \to Fx$
$\quad|\quad I, f : (x : A) \to B$
$\quad|\quad I, fa = y$

pure observations

$e \;::=\; \Diamond$
$\quad|\quad e.App[a]$
$\quad|\quad e.Bod[a]$
$\quad|\quad e.Arg$

blame observations

$o \;::=\; \Diamond$
$\quad|\quad o.App[a]$
$\quad|\quad o.Bod[a]$
$\quad|\quad o.Arg$
$\quad|\quad Inspect$ look at input

Environmental judgment,

$I \,\Box\, a : A \; o \rhd b : B$

ape lambda syntax with $fa \Rightarrow x$

Figure 1: Environments and Observations

$fa = g, ga' = h$	written	$f\,a\,a' = h$	
$X = (x : Y) \to \star, Y = a$	written	$X = (x : a) \to \star$	
$X = (x : Y) \to Fx, Y = a$	written	$X = (x : a) \to Fx$	
$f : (x : Y) \to Fx, Y = a$	written	$f : (x : a) \to Fx$	
$X = (x : Y) \to Fx, Fa = h$	written	$X.Bod[a] = h$	(?)
$X = (x : Y) \to Fx, Y = h$	written	$X.Arg = h$	(?)
$\star :: (\star \to \star) :: A$	written	$!!$	blame that can inhabit any type

need to properly abbreviate e

colorize errors?

Figure 2: Environments and Observations

Figure1 lists the environment and allowed observations. To explore well cast terms, we will allow an environment to take on partial assumptions about how undefined variables might evaluate. Pure observations are used internally by the environment and blame observations are used to unpack the term under consideration. By construction, environments cannot omit blame.

The environment assignments are restrictive by design, each assignment allowed the least possible output. However this is notationally cumbersome, so we will make heavy use of the abbreviations in 2. We will summarize a "hard error" of !! out of the existing blamable casts.

1.1 Examples

turn this into a table? clean up the realizability to use a context

Consider some examples before continuing,

if we have $\lambda x \Rightarrow !! \quad : \star \to \star$, we can unwrap the blamable !! by applying a freely assumed element of type $\star$

$x : \star \,\Box\, \lambda x \Rightarrow !! \quad : \star \to \star.app[x] \rhd !! \quad : \star$

in this case, the environment can be realized by $x = \star$.

If we have $\lambda f \Rightarrow f\,!! \quad : (\star \to \star) \to \star$, we can apply the free function f and then observe it's argument

$f : \star \to \star \,\Box\, \lambda f \Rightarrow f\,!! \quad : (\star \to \star) \to \star.app[f].Inspect \rhd !! \quad : \star$

in this case the environment could be realized by $f = \lambda x \Rightarrow x$.

$$\frac{I \,\square\, c : C \;\; o \rhd b : (x : A) \to B \quad I \vdash a : A}{I \,\square\, c : C \;\; o.app[a] \rhd b\, a : B\,[x := a]}$$

$$\frac{I \,\square\, c : C \;\; o \rhd (x : A) \to B : \star}{I \,\square\, c : C \;\; o.Arg \rhd A : \star}$$

$$\frac{I \,\square\, c : C \;\; o \rhd (x : A) \to B : \star \quad I \vdash a : A}{I \,\square\, c : C \;\; o.Bod[a] \rhd B\,[x := a] : \star}$$

$$\frac{I \,\square\, c : C \;\; o \rhd f a : C' \quad I \vdash a : A}{I \,\square\, c : C \;\; o.Inspect \rhd a : A}$$

$$\frac{I \,\square\, c : C \;\; o \rhd a : A \quad x \equiv \star \in I}{I \,\square\, c : C \;\; o \rhd a\,[x := \star] : A\,[x := \star]}$$

$$\frac{I \,\square\, c : C \;\; o \rhd a : A' \quad x \equiv (y : A) \to B \in I}{I \,\square\, c : C \;\; o \rhd a\,[x := (y : A) \to B] : A'\,[x := (y : A) \to B]}$$

$$\frac{I \,\square\, c : C \;\; o \rhd a : A' \quad xb \equiv y \in I}{I \,\square\, c : C \;\; o \rhd a\,[xb := y] : A'\,[xa := b]}$$

$$\frac{I \,\square\, c : C \;\; o \rhd a : A \quad a \rightsquigarrow a'}{I \,\square\, c : C \;\; o \rhd a' : A}$$

Figure 3: Symbolic reduction

We can opportunistically pick values for casts to induce errors. For instance, $\lambda X \Rightarrow \star ::_{\star} X \quad : (X : \star) \to X$, will reach blame when

$X : \star, X = - \to - \,\square\, \lambda X \Rightarrow \star ::_{\star} X \quad : (X : \star) \to X.app[X] \rhd \star ::_{\star} - \to - \quad : - \to -$

which is realizable with $X = \star \to \star$

We may be in a situation were casts need to be resolved in order to reach underlying terms $\lambda X \Rightarrow (\lambda x \Rightarrow !!) ::_{\star \to \star} X \quad : \star \to \star$

$X : \star, X = \star \,\square\, \lambda X \Rightarrow (\lambda x \Rightarrow !!) ::_{\star \to \star} X \quad : \star \to \star.app[X] \rhd (\lambda x \Rightarrow !!) ::_{\star \to \star} \star \quad : \star$

is blamable by itself but we may still want to see that !! is reachable

$X : \star, X = \star \to \star, x : \star \,\square\, \lambda X \Rightarrow (\lambda x \Rightarrow !!) ::_{\star \to \star} X \quad : \star \to \star.app[X].app[x] \rhd !! ::_{\star} \star \quad : \star$

double ch
tions here

In general the procedure will search into the term only if it possible for blame to reside their. We may also privilege the search in favor of locations that have not yet been blamed.

Additionally we allow indexing into types to extract blame. For instance, $!! \to \star \quad : \star$

$\square !! \to \star \quad : \star.Arg \rhd !! \quad : \star$

note that this additional extraction is needed since the blame relation does not directly allow extraction from types.

This blame be realizable with $(\star :: !! \to \star :: \star \to \star)\star$

double ch

Similarly we will allow extraction from the dependent body of a function type $(b : \mathbb{B}_c) \to b\star !! \star \quad : \star$

$b : \mathbb{B}_c, b\star = y \,\square\, (b : \mathbb{B}_c) \to b\star !! \star \quad : \star.Bod[b].Inspect \rhd !! \quad : \star$

figure out
this

1.2 Symbolic Reduction

The rules for symbolic reduction are listed in 3. The first 4 rules simplify the well cast term being inspected. The next 3 rules allow free substitutions of assignments from the context[1]. The final rule allows normal call by value steps.

We are left to characterize what makes a reasonable environment. We would like that we can only form environments that can be realized as program terms, but this seems too difficult. So we will instead remove the large classes of bad contexts that can be removed in a simple way.

In 4 environments must obey the typing rules. The judgment $- = y \not\in I$ ensures that y was not assigned already and removes the possibility of bad dependencies that could arise. For instance, $f 1_c = y$, $f 2_c = y$ over specifies f

[1]The notion of substitution must be extended and work over the definitional equivalence class to perform substitution along spines. For instance, $h\,(f\,2)\,[f\,(1+1) := 3] = h\,3$

$$\frac{}{.\ \mathbf{ty}}$$

$$\frac{I\ \mathbf{ty}}{I, X : \star\ \mathbf{ty}}$$

$$\frac{I\ \mathbf{ty} \quad I \vdash X : \star}{I, X = \star\ \mathbf{ty}}$$

$$\frac{I\ \mathbf{ty} \quad I \vdash (x : Y) \to F x \ \ : \star}{I, X = (x : Y) \to F x\ \mathbf{ty}}$$

$$\frac{I\ \mathbf{ty} \quad I \vdash (x : A) \to B \ \ : \star}{I, f : (x : A) \to B\ \mathbf{ty}}$$

$$\frac{I\ \mathbf{ty} \quad f : (x : A) \to B \in I \quad I \vdash a : A \quad I \vdash y : B\,[x \coloneqq a] \quad - = y \not\in I}{I, fa = y\ \mathbf{ty}}$$

Figure 4: Environment type checking

$$\frac{\begin{array}{c} I\ \mathbf{ty} \\ \forall f\bar{e} = b \in I, \forall f\overline{e'} = b \in I. \quad b \neq b' \supset \exists i. I \vdash e_i \neq e'_i \end{array}}{I\ \mathbf{plausible}}$$

$$\frac{}{(x : A) \to B \neq \star}$$

$$\frac{}{\star \neq (x : A) \to B}$$

$$\frac{I \vdash ae \neq a'e}{I \vdash a \neq a'}$$

Figure 5: Environment plausibility

needed
tion the
nt can be
eeded a
ps up in
Fx

and couples the 2 inputs, which should only be possible on individual operations

Type checking alone does not make the system consistent. It is possible for a well typed context to contradict itself. For instance, $X : \star, X = \star, X = - \to -$ is well typed but inconsistent. So we extend it with the plausibility constraint listed in 5

The plausibility constraint insists that if a difference can be observed in a functions output, there must be a difference observed in the functions input. This simulates the call graph of the partially defined higher order function while not needing to fully realize syntax. Specifically the environment may need to make further assignments so that partially defined elements get further definition. For instance,

$F : \mathbb{B}_c \to \star, b : \mathbb{B}_c, F\, true_c = - \to -, F\, b = \star$ is not plausible since it is possible $b = true_c$. Recall that $\mathbb{B}_c \coloneqq (X : \star) \to X \to X \to X$ and $true_c \coloneqq \lambda -\ then - \Rightarrow then$.

If we give an assignment that differentiates b from $true_c$ then this assignment can be made plausible. For instance, $b\ \star\ (\star \to \star)\ \star = \star$[2], therefore

$I = F : \mathbb{B}_c \to \star, b : \mathbb{B}_c, F\, true_c = - \to -, F\, b = \star, b\ \star\ (\star \to \star)\ \star = \star$ is plausible since the pair $F\, true_c = - \to -$ and $F\, b = \star$ can differentiate their arguments by $I \vdash b\ \star\ (\star \to \star)\ \star \neq true_c\ \star\ (\star \to \star)\ \star$.

The plausibility constraint naturally handles self reference that may occur over free variables. For instance,

$F : \star \to \star, F\ (\star \to \star) = - \to -, F\ (F \star) = \star$ is implausible since it is not clear that $(\star \to \star) \neq (F \star)$, this can be made plausible by the further assignment $F \star = \star$, thus

$I = F : \star \to \star, F\ (\star \to \star) = - \to -, F\ (F \star) = \star, F \star = \star$ since $I \vdash (F \star) = \star \neq (\star \to \star)$

Additionally it is possible to partially account for dependent types by redirecting them to functions.

clean up this example

$I = F : \star \to \star, F \star = \star, f : (x : \star) \to F\, x, f\star = (x : \star) \to -$ since $I \vdash (F \star) = \star \neq (\star \to \star)$

[2]Incidentally $false_c\ \star\ (\star \to \star)\ \star = \star$

This is a relatively lightweight constraint, especially when a function has taken on few assignments.

run through one more substantial example

phrase as counter examples

However since there is no way for a terms within the cast language to "observe" a distinction between the type formers plausible environments cannot always be realized back to a term that would witness the bad behavior. For instance, the environment $F : \star \to \star, F \star = \star \to \star, F\ (\star \to \star) = \star$, which might be needed to explore the term $\lambda F \Rightarrow ... :: \star :: F\,(\star \to \star) ...... :: (\star \to \star) :: F \star\ \ : (\star \to \star) \to ...$, cannot be realized as a closed term. In this way the environment is stronger then the cast language. The environment reflects a term language that has a type case construct.

cite

add non termination as a source of unrealizability

add por as a source of unrealizability

It is unclear that this over aggressive behavior is a problem in practice. The user must have already failed the standard type check, and the error does point out a concrete place that the type information does not line up. Recall that the parametricity properties of the cast language are already weakened so that many of the environments where this could not be realized in the surface language can be realized in the cast language with blame, for instance $F : Unit_c, F\ \star\ \star = (\star \to \star)$ can be realized by $\lambda X - \Rightarrow (\star \to \star) :: X$.

make sure
tioned in

2 Related work

2.1 Testing

Many of the testing strategies for typed functional programming trace their heritage to **property based** testing in QuickCheck [CH01].

- QuickChick [3] [DHL$^+$14][LPP17, LGWH$^+$17, Lam18] uses type-level predicates to construct generators with soundness and completeness properties, but without support for higher order functions. Current work in this area uses coverage guided techniques in [LHP19] like those in symbolic execution. More recently Benjamin Pierce has used Afl on compiled Coq code as a way to generate counter examples[4].
- [DHT03] added QuickCheck style testing to Agda 1.

review thi
there's a
now?

2.2 Symbolic Execution

Symbolic evaluation is a technique to efficiently extract errors from programs. Usually this happens in the context of an imperative language with the assistance of an SMT solver. Symbolic evaluation can be supplemented with other techniques and a rich literature exists on the topic.

The situation described in this chapter is unusual from the perspective of symbolic execution. But symbolic execution is probably still a good description

- the number of blamable source positions is limited by the location tags. Thus the search is error guided, rather then coverage guided.
- The language is dependently typed. Often the language under test is untyped.
- The language needs higher order execution. often the research in this area focuses on base types that are efficiently handleable with an SMT solver.

This limits the prior work to relatively few papers

- A Symbolic execution engine for Haskell is presented in [HXB$^+$19], but at the time of publication it did not support higher order functions.
- A system for handling higher order functions is presented in [NTHVH17], however the system is designed for Racket and is untyped. Additionally it seems that there might be a state space explosion in the presence of higher order functions.

[3]https://github.com/QuickChick/QuickChick
[4]https://www.youtube.com/watch?v=dfZ94N0hS4I

- [?] extended and corrected some issues with [NTHVH17], but still works in a untyped environment. The authors note that there is still a lot of room to improve performance.
- Closest to the goal here, [LT20] uses game semantics to build a symbolic execution engine for a subset of ML with some nice theoretical properties.
- That crazy stuff B.P. was up to

3 Discussion

move this into the section above

The goal of this chapter has been to describe a procedure that is suitable for implementation. To accomplish this several areas of meta-theory have been ignored. This approach suggests two desirable properties

1. Every error that could be caused by the program can be observed via symbolic execution
2. Every error in observed by symbolic executions can be realized as a program (no error is spurious)

prove,

We strongly conjecture the first property to hold .

The 2nd property is more subtle. We have not described evaluation contexts sufficiently, this is to maintain compatibility with modules that have not been formalized. For instance,

```
f : * ;
f = (x : Bool) -> (Nat :: Bool) ;
```

right f
a weird

will not be able to induce a term level error, since no term level observation can observe the type cast. However we want to observe errors here because f could be exported through the module system and used in an unforeseen type annotation where an error could be observed.

Part III
The Full Language

4 Examples

subsume into parts

in free functions

```
f h = h (\ x => err)
```

functions consistent

```
f h =
  case h 1
    True => case h 1
      False => err
```

and all variants

but worse

```
f h g =
  case h g
    True => case h (\x => g x)
      False => err
```

but worse (can recursively rely on itself) reachability constraint,

I	$::=$	$\Diamond$	
	$\vert$	$X:\star$	
	$\vert$	$X=\star$	
	$\vert$	$X=Y\to\star$	
	$\vert$	$X=(x:Y)\to Fx$	
	$\vert$	$X=D\overline{y}$	?
	$\vert$	$x:X$	abstract element?
	$\vert$	$x:D\overline{y}$	
	$\vert$	$x=d\overline{y}$	concrete element
	$\vert$	$f:(x:Y)\to Fx$	
	$\vert$	$fa=y$	
	$\vert$	$x_p:A\approx B$	
	$\vert$	$x_p=refl$	
	$\vert$	$x_p=cong$	lookup syntax
	$\vert$	$x_p=InTC$	
	$\vert$	$x_p=InC$	
	$\vert$	$A=B$	arbitrary constraint(?)

Figure 6: Cast Language Syntax

```
g: Nat -> Nat ;
h: (Nat -> Nat) -> Nat ;

f h g =
  case h g
    True => case h (\x => h (x (\y => 0))
      False => err
```

handle inflexable recursion

surface

```
f (eq : Id (Nat -> Nat) (\x => x+1) (\x => 1+x)) =
  case eq
    _ => ...?
```

“paremetric” types

```
f T t t' h =
  case h t t'
    True => case h t t'
      False => err
```

and all varients

surface

```
f T (t :T) (eq : Id _ T (Id Nat 1 2)) =
  case eq
    _ => ...?
```

handle the k binders

the key here being that when paths are inspected the constraints must hold to be plausible in addition to the rules listed above

just define

$$\frac{I\,\Box\,c:C\ \ o\rhd D\overline{a}\quad I\vdash a_i:A}{I\,\Box\,c:C\ \ o.InTC_i\rhd a_i:A}$$

fully appl

$$\frac{I\,\Box\,c:C\ \ o\rhd d\overline{a}\quad I\vdash a_i:A}{I\,\Box\,c:C\ \ o.InC_i\rhd a_i:A}$$

fully appl

5 Discussion and Future Work

The goal of this chapter has been to describe a procedure that is suitable for implementation. To accomplish this several areas of meta-theory have been ignored. This approach suggests two desirable properties

1. Every error that could be caused by the program can be observed via symbolic execution

2. Every error in observed by symbolic executions can be realized as a program (no error is spurious)

prove,

We strongly conjecture the first property to hold .

The 2nd property is more subtle. We have not described evaluation contexts sufficiently, this is to maintain compatibility with modules that have not been formalized. For instance,

```
f : * ;
f = (x : Bool) -> (Nat :: Bool) ;
```

right f
a weird

will not be able to induce a term level error, since no term level observation can observe the type cast. However we want to observe errors here because f could be exported through the module system and used in an unforeseen type annotation where an error could be observed.

More subtle is that the procedure described here will allow f to observe parallel or, even though parallel or cannot be constructed within the language. This hints that the approach presented here could be revised in terms of games semantics (perhaps along lines like [LT20]). Though game semantics for dependent types is a complicated subject in and of itself .

References

[CH01] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2001.

[DHL+14] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[DHT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 188–203. Springer, 2003.

[HXB+19] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 411–424, New York, NY, USA, 2019. Association for Computing Machinery.

[Lam18] Leonidas Lampropoulos. Random testing for language design. 2018.

[LGWH+17] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129, 2017.

[LHP19] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[LPP17] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[LT20] Yu-Yang Lin and Nikos Tzevelekos. Symbolic Execution Game Semantics. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[NTHVH17] Phuc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming*, 27, 2017.

Part IV

Alternative formulations

6 function fragment (inconsistent functions)

enforce that the observer never leaks a cast

I is a "thin ctx" may need to add an evaluation context

$$\frac{I \,\square\, c : C \;\; o \rhd \mathsf{fun}\, f\, x \Rightarrow b : (x : A) \to B \quad I \vdash y : A}{I \,\square\, c : C \;\; o.app[a] \rhd b\,[f := \mathsf{fun}\, f\, x \Rightarrow b, x := y] : B\,[x := y]}$$

a little abuse of notation

$$\frac{I \,\square\, c : C \;\; o \rhd (x : A) \to B : \star}{I \,\square\, c : C \;\; o.Arg \rhd A : \star}$$

$$\frac{I \,\square\, c : C \;\; o \rhd (x : A) \to B : \star \quad I \vdash a : A}{I \,\square\, c : C \;\; o.Bod[a] \rhd B\,[x := a] : \star}$$

(use a var insteead of a?)

$$\frac{I \,\square\, c : C \;\; o \rhd f a : C' \quad I \vdash a : A}{I \,\square\, c : C \;\; o.Inspect \rhd a : A}$$

from ctx

$$\frac{I \,\square\, c : C \;\; o \rhd a : A' \quad x \equiv \star \in I}{I \,\square\, c : C \;\; o \rhd a\,[x := a] : A\,[x := a]}$$

$$\frac{I \,\square\, c : C \;\; o \rhd a : A' \quad x \equiv (y : A) \to B \in I}{I \,\square\, c : C \;\; o \rhd a\,[x := (y : A) \to B] : A\,[x := (y : A) \to B]}$$

$$\frac{I \,\square\, c : C \;\; o \rhd a : A' \quad xb \equiv y \in I}{I \,\square\, c : C \;\; o \rhd a\,[xb := y] : A\,[xa := b]}$$

like conv

$$\frac{I \,\square\, c : C \;\; o \rhd a : A \quad a \rightsquigarrow a'}{I \,\square\, c : C \;\; o \rhd a' : A}$$

the goal is to reach a location with blame
need probes still

need to run an example with Leb. Id (\x=>x+1) (\x=>1+x)

Reachability contexts are not necessarily consistent. For instance, $X : \star, X = \star, X = \star \to \star$.

To prune some, but not all of these unsatisfiable constraints we will define a judgments that will tell when reachability contexts are plausible.

$$\frac{}{.\ \textbf{plausible}}$$

$$\frac{I\, \textbf{plausible}}{I, X : \star\ \textbf{plausible}}$$

$$\frac{I\, \textbf{plausible} \quad X : \star \in I \quad X\, \textbf{unasigned}\, I}{I, X = \star\ \textbf{plausible}}$$

$$\frac{I\, \textbf{plausible} \quad X : \star \in I \quad X\, \textbf{unasigned}\, I \quad I \vdash (x : A) \to B : \star}{I, X = (x : A) \to B\ \textbf{plausible}}$$

$$\frac{I\, \textbf{plausible} \quad X : \star \in I \quad X\, \textbf{unasigned}\, I \quad I \vdash (x : A) \to B : \star}{I, X = (x : A) \to B\ \textbf{plausible}}$$

$$\frac{I\,\textbf{plausible} \quad I \vdash (x:A) \to B : \star}{I, F : (x:A) \to B\ \textbf{plausible}}$$

$$\frac{I\,\textbf{plausible} \quad f : (x:A) \to B \in I \quad I \vdash a : A \quad I \vdash b : B\,[x := a]}{I, f\,a = b\ \textbf{plausible}}$$

note that for the sake of simplicity, there is no condition that $f\,a$ **unasigned** I since it is hard to characterize equality over the function sublanguage. Specifically this allows
$F : \star \to \star, F\star = \star, F\star = \star \to \star$ **plausible**.

$I \vdash b$ abbreviation is a little sketchy in terms of partially defined functions

consider restricting this, which would formally give the environment typecase

Control flow is degenerate in this setting, since all control flow would be handled through church encodings. For instance,
$\lambda n \Rightarrow n\,\mathbb{B}_c\,true_c\ (\lambda x \Rightarrow bad)\ : \mathbb{N}_c \to \mathbb{B}_c$
Is an encoding of a loop with bad behavior in its body. but since control follow is complete delegated to the higher order n, the environment is free to inspect each input independently in it's own environment.

handle bad inputs that fall out via call by value?ned to work directly against the blame judgment

Unlike with the full language there is little reason to calculate the specific output of a given term.
$\lambda n \Rightarrow n\,\mathbb{B}_c\,true_c\,false_c\ : \mathbb{N}_c \to \mathbb{B}_c$

justify this since type inhabitation is HARD

7 function fragment (global check)

reachability constraint,

I	$::=$	$\lozenge$	environment
	$\mid$	$X : \star$	
	$\mid$	$X = \star$	
	$\mid$	$X = (x:Y) \to F x$	
	$\mid$	$F : Y \to \star$	
	$\mid$	$f : (x:Y) \to F x$	? Think it should be $f : (x:A) \to B$
	$\mid$	$f a = y$	
e	$::=$	a	
	$\mid$	$.Bod[a]$	
	$\mid$	$.Arg$	

ape lambda syntax with $\Rightarrow$

by construction, environments cannot omit blame

$f a = g, g a' = h$	written	$f\,a\,a' = h$	
$X = (x:Y) \to \star, Y = a$	written	$X = (x:a) \to \star$	
$X = (x:Y) \to F x, Y = a$	written	$X = (x:a) \to F x$	
$f : (x:Y) \to F x, Y = a$	written	$f : (x:a) \to F x$	
$X = (x:Y) \to F x, F a = h$	written	$X.Bod[a] = h$	(?)
$X = (x:Y) \to F x, Y = h$	written	$X.Arg = h$	(?)

$$\frac{}{.\ \textbf{ty}}$$

$$\frac{I\,\textbf{ty}}{I, X : \star\ \textbf{ty}}$$

$$\frac{I\,\textbf{ty} \quad X : \star \in I}{I, X = \star\ \textbf{ty}}$$

$$\frac{I\,\textbf{ty} \quad X : \star \in I \quad I \vdash Y \to \star\ : \star}{I, X = Y \to \star\ \textbf{ty}}$$

$$\frac{I\ \mathbf{ty} \quad X : \star \in I \quad I \vdash (x : Y) \to F x \ : \star}{I, X = (x : Y) \to F x\ \mathbf{ty}}$$

$$\frac{I\ \mathbf{ty} \quad f : (x : A) \to B \in I \quad I \vdash a : A \quad I \vdash y : B\,[x \coloneqq a] \quad - = y \not\in I}{I, f a = y\ \mathbf{ty}}$$

$$\frac{\begin{array}{c} I\ \mathbf{ty} \\ \forall f \overline{e} = b \in I, \forall f \overline{e'} = b \in I. \quad b \neq b' \supset \exists i. I \vdash e_i \neq e'_i \end{array}}{I\ \mathbf{plausible}}$$

$$\frac{}{(x : A) \to B \neq \star}$$

$$\frac{}{\star \neq (x : A) \to B}$$

$$\frac{I \vdash a\overline{e} \neq a'\overline{e}}{I \vdash a \neq a'}$$

The judgment $- = y \not\in I$ ensures that y was not assigned already and removes the possibility of bad dependencies that could arise. For instance, $f 1_c = y$, $f 2_c = y$ over specifies f and couples the 2 inputs, which should only be possible on individual operations

This may since as a better ass chosen

explain how the plausibility constraint might induce more assignments

The plausibility constraint insists that if a difference can be observed in a functions output, there must be a difference observed in the functions input. This simulates the call graph of the partially defined higher order function while making as few commitments as possible. Specifically the environment may need to make further assignments so that partially defined elements get further definition. For instance,

$F : \mathbb{B}_c \to \star, b : \mathbb{B}_c, F\,true_c = - \to -, F\,b = \star$ is not plausible since it is possible $b = true_c$. Recall that $\mathbb{B}_c \coloneqq (X : \star) \to X \to X \to X$ and $true_c \coloneqq \lambda - then - \Rightarrow then.$

If we give an assignment that differentiates b from $true_c$ then this assignment can be made plausible. For instance, $b \star (\star \to \star) \star = \star$[5], therefore

$I = F : \mathbb{B}_c \to \star, b : \mathbb{B}_c, F\,true_c = - \to -, F\,b = \star, b \star (\star \to \star) \star = \star$ is plausible since the pair $F\,true_c = - \to -$ and $F\,b = \star$ can differentiate their arguments by $I \vdash b \star (\star \to \star) \star \neq true_c \star (\star \to \star) \star$.

The plausibility constraint naturally handles self reference that may occur over free variables. For instance,

$F : \star \to \star, F\ (\star \to \star) = - \to -, F\ (F \star) = \star$ is implausible since it is not clear that $(\star \to \star) \neq (F \star)$, this can be made plausible by the further assignment $F \star = \star$, thus

$I = F : \star \to \star, F\ (\star \to \star) = - \to -, F\ (F \star) = \star, F \star = \star$ since $I \vdash (F \star) = \star \neq (\star \to \star)$

Additionally it is possible to partially account for dependent types by redirecting them to functions.

clean up this example

$I = F : \star \to \star, F \star = \star, f : (x : \star) \to F\,x, f \star = (x : \star) \to -$ since $I \vdash (F \star) = \star \neq (\star \to \star)$

This is a relatively lightweight constraint, especially when a function has taken on few assignments.

run through one more substantial example

phrase as counter examples

add non termination as a source of unsoundness

However since there is no way for a term to "observe" a distinction between the type formers plausible environments cannot always be realized back to a term that would witness the bad behavior. For instance, the environment $F : \star \to \star, F \star = \star \to \star, F\ (\star \to \star) = \star$ cannot be realized as a closed term. In this way the symbolic environment is stronger then the cast language. The environment reflects a term language that has a type case construct.

cite

$\lambda F \Rightarrow ... :: \star :: F\,(\star \to \star) ...... :: (\star \to \star) :: F \star \ : (\star \to \star) \to ...$

It is unclear that this over aggressive warning is a problem in practice. The user must have already failed the standard type check, and the error does point out a concrete place that the type information does not line up. Recall that the parametricity properties of the cast language are already weakened so that many of the environments where this could not be realized in the surface language can be realized in the cast language with blame, for instance $F : Unit_c, F \star \star = (\star \to \star)$ can be realized by $\lambda X - \Rightarrow (\star \to \star) :: X$.

make sure tioned in

[5] Incidentally $false_c \star (\star \to \star) \star = \star$

7.1 bad example

consider the more intricate example that would arise if the user implicitly assumed the equality of $\lambda x \Rightarrow x +_c 1_c$ and $\lambda x \Rightarrow 1_c +_c x$. After elaboration we might be left with a term such as

$D = \lambda C \Rightarrow (\lambda x \Rightarrow \star) ::_{(x:C(\lambda x \Rightarrow 1_c +_c x)) \to \star} (x : C\,(\lambda x \Rightarrow x +_c 1_c)) \to \star \quad : (C : (\mathbb{N}_c \to \mathbb{N}_c) \to \star) \to C\,(\lambda x \Rightarrow x +_c 1_c) \to \star$

The goal is to find a plausible I and o such that $a$can be blamed

$I \,\square\, D\ o \rhd a$

The procedure can see that a possible error can be exercised within the body of the function. Since the terms is a function it can only be applied

$I = C : ((\mathbb{N}_c \to \mathbb{N}_c) \to \star) \ldots$

leaving the simpler problem

$I \,\square\, D\ App[C] \rhd (\lambda x \Rightarrow \star) ::_{(x:C(\lambda x \Rightarrow 1_c +_c x)) \to \star} (x : C\,(\lambda x \Rightarrow x +_c 1_c)) \to \star$

Again procedure can see that a possible error can be exercised within the body of the function, under casts that agree it is a function

$I = C : ((\mathbb{N}_c \to \mathbb{N}_c) \to \star), x : C\,(\lambda x \Rightarrow x +_c 1_c) \ldots$

$I \,\square\, D\ App[C].App[x] \rhd \star ::_\star \star$

7.2 bad example

Bool or even type example

consider the more intricate example that would arise if the user implicitly assumed the equality of $\lambda x \Rightarrow x +_c 1_c$ and $\lambda x \Rightarrow 1_c +_c x$. after elaboration we would explore the term

$refl_{\lambda x \Rightarrow x +_c 1_c : \mathbb{N}_c \to \mathbb{N}_c} ::_{(\lambda x \Rightarrow x +_c 1_c) \doteq_{\mathbb{N}_c \to \mathbb{N}_c} (\lambda x \Rightarrow x +_c 1_c)} (\lambda x \Rightarrow x +_c 1_c) \doteq_{\mathbb{N}_c \to \mathbb{N}_c} (\lambda x \Rightarrow 1_c +_c x)$

Recall the definitions in Chapter 2

$(\lambda x \Rightarrow x +_c 1_c) \doteq_{\mathbb{N}_c \to \mathbb{N}_c} (\lambda x \Rightarrow 1_c +_c x) = (C : ((\mathbb{N}_c \to \mathbb{N}_c) \to \star)) \to C\ (\lambda x \Rightarrow x +_c 1_c) \to C\ (\lambda x \Rightarrow 1_c +_c x)$

$(\lambda x \Rightarrow x +_c 1_c) \doteq_{\mathbb{N}_c \to \mathbb{N}_c} (\lambda x \Rightarrow x +_c 1_c) = (C : ((\mathbb{N}_c \to \mathbb{N}_c) \to \star)) \to C\ (\lambda x \Rightarrow x +_c 1_c) \to C\ (\lambda x \Rightarrow x +_c 1_c)$

$refl_{\lambda x \Rightarrow x +_c 1_c : \mathbb{N}_c \to \mathbb{N}_c} = \lambda - cx \Rightarrow cx$

The goal is to find an I and o such that $a$can be blamed

$I \,\square\, refl_{\lambda x \Rightarrow x +_c 1_c : \mathbb{N}_c \to \mathbb{N}_c} ::_{(\lambda x \Rightarrow x +_c 1_c) \doteq_{\mathbb{N}_c \to \mathbb{N}_c} (\lambda x \Rightarrow x +_c 1_c)} (\lambda x \Rightarrow x +_c 1_c) \doteq_{\mathbb{N}_c \to \mathbb{N}_c} (\lambda x \Rightarrow 1_c +_c x)\ \ o \rhd a$

since $refl$ is a function, and the type constructor on the $\doteq$ match apply free variable to all the arguments

$I = C : ((\mathbb{N}_c \to \mathbb{N}_c) \to \star), C\ (\lambda x \Rightarrow x +_c 1_c) \ldots$

$I \,\square \ldots o \rhd a$

phrase as counter example

add non termination as a source of unsoundness

8 alt function fragment (incremental plausibility)

$$\frac{\begin{array}{c} I\,\textbf{plausible} \quad f : (x : A) \to B \in I \\ I \vdash a' : A \quad I \vdash b' : B\,[x \coloneqq a'] \\ \forall f\overline{a} = b \in I. \quad b \neq b' \supset a \neq a' \end{array}}{I, fa' = b'\ \textbf{plausible}}$$

the last line is incorrect, hard to build out a piecemeal check

would need to define the inequality judgments

Additionally since there is no way for a term to "observe" a distinction between types so the constraint context $F : \star \to \star, F\star = \bot_c, F\ \bot_c = \star$ cannot be realized as a closed term. In this way the symbolic environment is stronger then the cast language and has access to a type case construct.

Part V
TODO

Todo list

9 notes

10 unused

...

$$\frac{A \neq A'}{(x:A) \to B \neq (x:A') \to B'}$$

here is the hard bit, you cannot check the inequality of a midpoint

$$\frac{B \neq B'}{(x:A) \to B \neq (x:A') \to B'}$$

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

...

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

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

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

...

$$\frac{HK \vdash A : \star}{HK \vdash x : A}\ldots$$

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

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

```
-- 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 7: Cast Pattern Matching

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

...

$$\frac{}{HK \vdash x : A} \cdots$$

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

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

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

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

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

Gen is defined as

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

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

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

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

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

other rules similar to the surface lang
 observations,

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

old style red rules

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

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

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

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

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

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

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

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

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

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

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

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

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,
$kin \quad ::= \quad k = left \mid k = right$
casts under assumption,
$kcast \quad ::= \quad \overline{\overline{kin},p};$
path exp.,

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

cast pattern,
$patc \quad ::= \quad x \mid d\,\overline{patc} ::_{x_p}$
cast expression,

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

observations,

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

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

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

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

$$\frac{}{(q\, p)^{-1} \leadsto p^{-1}\, q^{-1}}$$

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

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

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

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

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

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

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

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

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

TODO review this

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

term redcutions

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

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

symetric around $\sim$

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

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

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

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

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

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

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

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

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

...

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

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

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

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