Chapter 5 (draft): Symbolic Execution

Mark Lemay

December 14, 2021

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

Part I
Symbolic Execution

one of the vlaues of type checking is the imeduacy of feedback. We have outlined here a system that will give warning messages immedialy, 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 (clasical) proof language, and will not generally exicute thier proofs. The symbolic evaluation sytem recaptures some of that qiucker feed back, by specifing a system that passively tries to find errors.

This process is semidecidable in general (by testing against all well typed syntax). But this is infeasably inefficient for Type syntax and function syntax, So we will present a procesure that treats elements of those types extentionally. Additionally we can again reduce the search space by only engaging with values of data type instead of all expressions.

Additionally since this preocedure operates over the cast langauge, we must decide what is a reonsable exicution of code

- on one extreame, our testing code could introduce new blame that is exercied
- 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 the procedure remains semidecidable, we intend to run it incrementally, with whatever resources aer availible. The procedure is inteded to be run durring the continous integration phase of standard software development.

1 Examples

subsume into parts

in free functions

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

functions consitent

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

and all varients

but worse

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

but worse (can recursively rely on itself)

```
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

Part II
The functional fragment

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?)

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

from ctx

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

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

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

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

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 [1] [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.
- [DHT03] added QuickCheck style testing to Agda 1.

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 scheme 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 presnes of higher order functions.
- 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

[1] https://github.com/QuickChick/QuickChick

3 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
n 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 .

Part III
The Full Language

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.

[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
TODO

Todo list

4 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.

5 unused

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

}
```

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

...

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

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

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

...

```
-- 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 1: Cast Pattern Matching

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

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

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

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

...

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

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

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

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

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

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

Gen is defined as

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

c?

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

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

...

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\sim$

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

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

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

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

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

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

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

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

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

...

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

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

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

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