Extrinsic Relational Subtyping

ACM Reference Format:
. 2025. Extrinsic Relational Subtyping. 1, 1 (January 2025), 37 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Context. Automatically catching errors in programs is a hard enough problem that many languages require users to provide simple specifications to limit that space of correctness. Languages, such as Java and ML, are *intrinsically typed*, requiring nearly all terms to be associated with some type specified by the user. The clever design of ML allows annotations to be fairly sparse by having types specified at constructor definitions and relying on type inference elsewhere. However, one of the drawbacks of intrinsically typed languages is that they prevent reusing of constructors in contexts that are less precise than their intrinsic specifications. For instance, a *cons* constructor belonging to a list datatype could not be considered amongst a *leaf* constructor defined as belong to a tree datatype. The user would have to define a new datatype that includes both isomorphs of both *cons* and *leaf* and also write functions that translate between the isomorphs and the list and tree constructors. This not only bloats the codebase, but hurts either the runtime or compiler performance.

For various reasons that may include the reusability drawbacks, intrinsically typed languages have lost favor, and untyped or *extrinsically typed* languages, such as Javascript and Python, have increased in popularity. Untyped languages place less initial burden on the programmer to define the upper bounds on specific combinations of constructors. The flexibility and reusability of writing code that doesn't have to fit some predefined restriction may be seen as one of the benefits of these extrinsically typed languages over the well-studied intrinsically-typed languages. Unfortunately, this freedom makes static analysis or type inference much more challenging.

Despite the ever increasing use of untyped languages in production systems, the need to automatically verify precise and expressive properties of systems has never been greater. To this end, researchers have extended the simple types (such as those found in ML) into *refinement types*, *predicate subtyping*, and *dependent types*.

Refinement types offer greater precision than simple types, but still rely on intrinsic type specifications. Dependent types can express detailed relations, but may require users to provide proofs along with detailed annotations. Predicate subtyping offers some of the expressivity of dependent types, but with the automatic subtyping of refinement types. All of these techniques are based on intrinsic typing and therefore require users to provide additional annotations beyond the runtime behavior of their programs.

The challenge with extrinsically typed languages is that they allow using constructors in any possible combination, rather than prescribing the upper bound of combinations as in the datatype mechanism of ML languages. Thus, the crux of typing extrinsically typed programs is to determine

Author's address:

ACM XXXX-XXXX/2025/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a precise type based on how constructors are used. Since the way constructors are use may overlap is various ways, this form of reasoning about types requires a notion of subtyping. Type systems for extrinsically typed languages have relied on unions and intersections between types to represent precise types based on how expressions are used in combination.

Gap. Because extrinsically typed languages do not require users to specify the upper bounds of program expressions, there are many untyped programs that cannot benefit from the typing techniques of intrinsically typed languages. Furthermore, extrinsically typed languages do not require users to provide proofs, that have no runtime behavior, as is sometimes necessary in dependently typed systems to verify more expressive types. For instance, the liquid type system [] can verify and infer some relational properties, but it requires users to specify ML-style base types and a set of logical qualifiers to draw from. On the other hand, existing extrinsically typed techniques can not represent richer notions of relations beyond the mere shapes of expressions. Thus, the challenge is to bring rich expressive types to extrinsically typed languages.

Innovation. To overcome the limitations of intrinsic type systems and expand the kinds of programs and types that can be type checked, we introduce *extrinsic relational type inference*: a novel system that automatically infers expressive properties from untyped functional programs.

The main idea behind relational typing is to leverage subtyping as a means to express relations between objects. This completely obviates the need for the two-level type language used in liquid types or predicate subtyping. There is no special first-order predicate language. In relational typing, a relation is just a type in a subtyping lattice, just as a shape is just a type in a subtyping lattice. A subtyping judgment can degenerate into a typing judgment when the left side or strong side of subtyping is a singleton type (type with a single inhabitant). **TODO: insert example of (succ zero, cons nil) subs nat list** Additionally, two separate relations may be compared via subtyping to say that one relation may hold true for a superset of inhabitants of another. **TODO: insert example of even list subs nat list** By embedding the notion of relations into subtyping the system can reuse techniques for inferring unions and intersections over simple types, which are necessary in an extrinsic setting.

In addition to checking that subtyping holds, the system is able to infer weak parameter types and strong return types of functions, which then serve as constraints to be checked according to the applications of functions.

For comparison, the meaning of subtyping relations in relational types corresponds to the meaning of implication between qualifiers in liquid types.

While the purely functional setting presented in this work is not suitable for practical programming, future work could extend it to incorporate side-effects to make it practical. Alternatively, the purely functional setting could be viewed as an alternative formal foundation more mathematics, allowing for greater proof automation by allowing reuse of proofs across the transitive closure of proposition subtyping.

2 OVERVIEW

TODO: mention somwhere that the second order quantification serves two distinct purposes; 1. polymorphism as in System-F. 2. refinement as in first-order quantification of liquid types. Relational types is able to leverage second-order quantification for refinement, eschewing the first-order quantification used in other systems.

For a given program, type inference constructs a very precise type. Some programs are simple enough such that type inference generates singleton types.

TODO: example of a inference of intersection of function param applied to multiple arguments (not novel)

TODO: example of a inference of intersection of param with multiple functions applied to it (not novel)
TODO: example of a inference of union type of branching (not novel)
TODO: break example program into parts; inline instead of using figure

We illustrate the syntax and semantics of programs and types with the example program shown in Fig. ??.

Path typing. Consider the function `talky`, which completes a simple English phrase:

```
let talky = (
  $ <hello> @ => <world> @
  $ <good> @ => <morning> @
  $ <thank> @ => <you> @
)
```

This program is defined by paths over hardcoded tags. The system infers the type to be an intersection of implication types:

```
TOP
& (<hello> @ -> <world> @)
& (<good> @ -> <morning> @)
& (<thank> @ -> <you> @)
```

Essentially, the program is so simple, that its type has the exact same meaning merely dressed in a different syntax.

Output broadening. Consider the application `talky(x)` where `x` has the type `(<hello> @)|(<thank> @)`. Type inference breaks apart the function type's intersection into paths and constructs the strongest output type possible by expanding the output type for each viable path. Since only two of the three paths match the type of the argument, type inference determines the type of the application to be the type `(<world> @)|(<you> @)`. Broadening from the bottom up contrasts with refinement type systems, which start from the weakest type intrinsic to the constructors and refines down to a stronger type through intersections of types or conjunctions of qualifiers.

Relational typing. Consider the function `repeat` that takes a natural number and returns a list of whose length is that number.

```
let repeat = $ x => loop($ self =>
  $ <zero> @ => <nil> @
  $ <succ> n => <cons>(x,self(n))
)
```

Without specifying any requirements besides the function definition, type inference lifts the function into the definitional property as a type. To construct the type, type inference constructs a relation between nats and lists. The type of `repeat` depends on a least fixed point relation between nats and lists (parametrically named here for readability).

```
natList(α) = LFP[R]( BOT
             | (<zero> @)*(<nil> @)
             | (EXI[N L ; N*L <: R](<succ> N)*(<cons> (α*L))
           )
```

Using the natList relation, type inference then lifts the function repeat into its precise type form.

```
ALL[T] T -> ALL[X] X -> EXI[Y ; X*Y <: natList(T)] Y
```

It may be worth noting that there could be semantically equivalent recursive type in terms of intersections instead of unions.

TODO: forward reference to correctness/model semantics

```
ALL[T] GFP[R]( TOP
  & (<zero> @)->(<nil> @)
  & (ALL[N L ; R <: N->L](<succ> N)->(<cons> T*L)
)
```

Type inference reasons in terms least fixed points, but the greatest fixed point form could be handled with syntactic sugar and rewriting.

Using the precise type form, type inference can leverage solving and checking subtyping constraints to reason in a number ways: it can reason forward from inputs to outputs (just like the runtime semantics), reason backwards from outputs to inputs (like Prolog), and check against weaker specifications.

Fixed point forward broadening. Consider the application `repeat(<succ> <succ> <zero> @)(x)` where `x` has type `T`. Type inference constructs a singleton type, mirroring the results achieved by simply running the program.

```
<cons> T * <cons> T * <nil> @
```

Fixed point backwards broadening. Now suppose we have a function `foo` whose input type is inferred to be an empty list or a singleton list, `<nil> @ | <cons> T * <nil> @`. Given the application `foo(repeat(n)(x))` where `x` has type `T`, type inference can reason backwards to learn that the type of `n` must be either zero or one.

```
<zero> @ | <succ> <zero> @
```

Vertical weakening (Factoring). Now suppose we have a function `woo` whose input type is inferred to be a list over elements of type `T`.

```
LFP[R]( BOT
  | <nil> @
  | <cons> T*R
)
```

Given the application `woo(repeat(n)(x))` where `x` has type `T`, type inference discovers that the argument type depends on the relation natList(`T`), and the relation can be factored into a weaker cross product of nats and lists. Therefore, the argument meets the requirements of `woo` and the type of `n` must be the natural numbers.

```
LFP[R]( BOT
  | <zero> @
  | <succ> R
)
```

Horizontol weakening (infilling). Now consider a function `boo` whose input type is the natural numbers. Suppose we have the application `boo(n)` where n guaranteed to be an even number.

```
LFP[R]( BOT
  | <zero> @
  | <succ> <succ> R
)
```

The application requires that type inference check that the nat type is weaker than the even type. Type inference sees that if both types were to unroll into an infinite sequence of values every value in nat would also be in even, therefore the application type checks. In particular, type inference leverages the inductive hypothesis, by learning that weaker types hold for all the recursive constraints of the stronger type. In the case simple recursive types as shown above, the inductive hypothesis is merely a subtyping constraint on a single variable. In the case of comparing two relations such as natList(`T`) and a corresponding even version, the inductive hypothesis would be a subtyping constraint on a pair of variables, which may not be decomposable into constraints on single variables, so type inference must learn relational constraints, in addition to simple constraints.

Input refinement. Consider two functions: `uno` of type `U->V` and `dos` of type `D->E`. Now suppose these two functions are called on the same variable, e.g. `(uno(x),dos(x))`. Type inference learns that the type of `x` can be no weaker than the intersection of the functions' input types: `U&D`.

Path sensitivity. Consider the function `max` that chooses the maximum of two natural numbers.

```
let lessOrEq = loop($ self =>
  $ (<zero> @),y => <true> @
  $ (<succ> x),(<succ> y) => self(x,y)
  $ (<succ> x),(<zero> @) => <false> @
) in
let max = $ (x,y) =>
  if lessOrEq(x,y) then y else x
```

The function `max` must satisfy the property that the result is greater or equal to each of the inputs. Type inference must learn constraints on the inputs to `max`: `x` and `y`, which depends on the output of `lessOrEq(x,y)`. The application `lessOrEq(x,y)` can evaluate to either `<true>@` or `<false>@`, which result from different paths taken in `lessOrEq`. Type inference considers both cases and maintains the learned constraints exist in different possible worlds, since they are learned from different paths. Finally, type inference connects the inputs to the outputs by considering the two possible paths of the `if-then-else`. It first lifts the function `lessOrEq` into a relational type. For readability, we name the relational type LED (as in "less than or equal decision").

```
LED = LFP[R] ( BOT
        | (EXI [Y] ((<zero> @)*Y)*(<true> @))
        | (EXI [X Y D ; ((X*Y)*Z) <: R] ((<succ> X)*(<succ> Y))*D)
        | (EXI [X] ((<succ> X)*(<zero> @))*(<false> @))
      )
```

using the LED relation, type inference combines the constraints learned for each possible world and combines them into a function type with multiple paths.

```
TOP
& (EXI [D ; D <: (<true> @)]
  (ALL [X Y Z ; Y <: Z ; ((X*Y)*D) <:LED] (X,Y) -> Z))
& (EXI [D ; D <: (<false> @)]
  (ALL [X Y Z ; X <: Z ; ((X*Y)*D) <: LED] (X,Y) -> Z))
```

We could simplify the type by eliminating simple constraints without loss of safety or precision:

```
TOP
& (ALL [X Y ; ((X*Y)*(<true> @)) <: LED] (X,Y) -> Y)
& (ALL [X Y ; ((X*Y)*(<false> @)) <: LED] (X,Y) -> X)
```

However, we have not included this rewriting in the semantics or implementation.

TODO: show type generated from code

TODO: more motivating and elucidating examples

3 DYNAMIC SEMANTICS

The programming language is pure and applicative. A program is an expression of the form in definition 3.1. The forms of expressions enable function abstraction, function application, tagged constructions, record construction, record projection, and pattern matching. These forms enable us to express non-trivial programs. They also allow for compositions that have no reasonable semantics.

Definition 3.1. Expressions

$$
\begin{array}{rcl}
e & ::= & x \mid \texttt{@} \mid \texttt{<}l\texttt{>}e \mid \vec{r} \mid \vec{f} \mid e\texttt{.}l \mid e\texttt{(}e\texttt{)} \mid \texttt{loop(}e\texttt{)} \mid \ldots \\
\vec{r} & ::= & \epsilon \mid \vec{r}\; r \\
r & ::= & \texttt{\$}l\texttt{=>}e \\
\vec{f} & ::= & \epsilon \mid \vec{f}\; f \\
f & ::= & \texttt{\$}p\texttt{=>}e \\
p & ::= & x \mid \texttt{@} \mid \texttt{<}l\texttt{>}p \mid \vec{k} \\
\vec{k} & ::= & \epsilon \mid \vec{k}\; k \\
k & ::= & \texttt{\$}l\texttt{=>}p
\end{array}
$$

Progression of an expression, as in definition 3.2, is a small-step operational semantics. It adheres to typical definitions of applicative languages for the most part. One slight departure is that pattern matching is merely a special case of function application. Likewise, a switch is merely a function abstraction. Records are similar to Functions, except that their entries are guarded by literal identifiers. Additionally, records may be abstracted as patterns, but functions may not. The semantics enables recursion via the fixed point combinator `loop`.

Definition 3.2. $\boxed{e \rightsquigarrow e}$ Progression

$$
\frac{e \rightsquigarrow e'}{\texttt{<}l\texttt{>}e \rightsquigarrow \texttt{<}l\texttt{>}e'}
\qquad
\frac{e \rightsquigarrow e'}{\vec{r}\texttt{\$}l\texttt{=>}e \rightsquigarrow \vec{r}\texttt{\$}l\texttt{=>}e'}
\qquad
\frac{\vec{r} \rightsquigarrow \vec{r}'}{\vec{r}\texttt{\$}l\texttt{=>}v \rightsquigarrow \vec{r}'\texttt{\$}l\texttt{=>}v}
\qquad
\frac{e \rightsquigarrow e'}{e\texttt{.}l \rightsquigarrow e'\texttt{.}l}
$$

$$
\frac{\texttt{\$}l\texttt{=>}v \in \vec{g} \qquad \forall\, e.\ \texttt{\$}l\texttt{=>}e \in \vec{g} \implies e = v}{\vec{g}\texttt{.}l \rightsquigarrow v}
\qquad
\frac{e_f \rightsquigarrow e'_f}{e_f\texttt{(}e\texttt{)} \rightsquigarrow e'_f\texttt{(}e\texttt{)}}
\qquad
\frac{e \rightsquigarrow e'}{\vec{f}\texttt{(}e\texttt{)} \rightsquigarrow \vec{f}\texttt{(}e'\texttt{)}}
$$

$$
\frac{\vec{f}\texttt{(}v\texttt{)} \rightsquigarrow e' \qquad \mathrm{FV}(e) \subseteq \mathrm{FV}(p)}{(\vec{f}\texttt{\$}p\texttt{=>}e)\texttt{(}v\texttt{)} \rightsquigarrow e'}
\qquad
\frac{\nexists e'.\ \vec{f}\texttt{(}v\texttt{)} \rightsquigarrow e' \qquad p \equiv v \dashv \vec{\sigma}}{(\vec{f}\texttt{\$}p\texttt{=>}e)\texttt{(}v\texttt{)} \rightsquigarrow e[\vec{\sigma}]}
\qquad
\frac{e \rightsquigarrow e'}{\texttt{loop(}e\texttt{)} \rightsquigarrow \texttt{loop(}e'\texttt{)}}
$$

$$
\frac{}{\texttt{loop(\$}x\texttt{=>}e\texttt{)} \rightsquigarrow e[x/\texttt{loop(\$}x\texttt{=>}e\texttt{)}]}
$$

4 STATIC SEMANTICS

The static semantics is a system for checking if the construction of an expression is viable. The system leverages types of the forms in Definition 4.1, which allow expressing properties of expressions with varying levels of precisions.

Definition 4.1. Types

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid \texttt{@} \mid \texttt{<}l\texttt{>}\tau \mid l\texttt{->}\tau \mid \tau\texttt{->}\tau \mid \tau\texttt{|}\tau \mid \tau\texttt{\&}\tau \mid \tau\texttt{\textbackslash}\eta \mid & \triangleright\ \textit{type} \\
 & & \texttt{EXI[}\vec{\alpha}\ \Delta\texttt{]}\tau \mid \texttt{ALL[}\vec{\alpha}\ \Delta\texttt{]}\tau \mid \texttt{LFP[}\alpha\texttt{]}\tau & \\
\eta & ::= & \texttt{EXI[}\vec{\alpha}\ \Delta\texttt{]}\phi \mid \phi & \triangleright\ \textit{subtracted type} \\
\phi & ::= & \alpha \mid \texttt{@} \mid \texttt{<}l\texttt{>}\phi \mid l\texttt{->}\phi \mid \phi\texttt{\&}\phi \mid & \triangleright\ \textit{pattern type} \\
\vec{\alpha} & ::= & \epsilon \mid \vec{\alpha}\ \alpha & \triangleright\ \textit{type sequence} \\
\Delta & ::= & \epsilon \mid \Delta\ \texttt{;}\tau\texttt{<:}\tau & \triangleright\ \textit{subtyping environment}
\end{array}
$$

Proof typing, in Definition 4.2, checks the viability of an expression's form. Additionally, by extending the forms of expressions with type annotations, as in Definition 8.2, proof typing is also able to check that expressions meet abstract specifications. In order to check the viability of constructions and specifications, proof typing lifts expressions into types and leverages subtyping to check compatibility between types. The proof typing predicate represents a typing that holds under the assumptions of a typing environment, and a world, where a world consists of emphclosed type variables and subtyping constraints.

Whether a variable is a closed or not represents how a variable is quantified.

If proof typing holds, then its typing holds under every interpretation of its closed type variables under some interpretation of its open variables. We formalize ths soundness claim in Section 5.

The proof typing predicate ($\Gamma \vdash e : \tau \dashv \Omega'$) can be interpreted as an algorithm that takes a typing environment Γ, an expression e, and a world Ω as inputs and returns a type and a world Ω' as outputs, such that the output world is an extension of the input world ($\Omega \preceq \Omega'$). The input world is not explicitly represented in the predicate, so the algorithm interprets the input world as the smallest world that can satisfy the predicate. Additionally, the outputs of the predicate are not deterministic. Therefore, the algorithm enumerates all possible outputs.

Rule 4.2.1. For the unit expression `@`, proof typing simply returns the singleton type of the same form.

Rule 4.2.2. For a variable type expression α, proof typing looks for a corresponding typing in the environment, and returns the corresponding type if found.

Rule 4.2.3. For a tag expression $\texttt{<}l\texttt{>}e$, proof typing recursively constructs the type of the tag's body, and returns the corresponding tag type.

Rule 4.2.4. For an empty expression, proof typing constructs the top type `TOP`, which is merely syntactic sugar for $\texttt{EXI[}\alpha\texttt{]}\alpha$.

Rule 4.2.5. For a non empty record expression $\vec{r}\texttt{\$}l\texttt{=>}e$, proof typing recursively constructs the type for each entry, and refines their types against each other via intersection.

Rule 4.2.6. For a function expression $\vec{f}$, proof typing delegates the work to two helper predicates: *function lifting*, as in definition 8.8, and *constrained implication congruence*, as in definition 8.34. Function lifting constructs a sequence of implication types, where each associated with a world, represented by $\vec{\pi}$. Constrained implication congruence, constructs a sequence of types congruent with each implication, such that each world is packaged with its corresponding type, resulting in a universally constrained type (if there are type variables in the original implication). During an application, each path of a function is tried in order, which means values matching subsequent patterns, will not be matched by earlier patterns. Thus, function lifting generates types from patterns, and for each pattern, it subtracts the types of previous patterns, represented by prefixes of $\vec{eta}$

Rule 4.2.7. For a projection expression $e\texttt{.}l$, proof typing leverages subtyping to check that there is an entry with label l in the supposed record e, and it learns a lower bound on the type of the body α associated with that label.

Rule 4.2.8. For an application expression$e_f\texttt{(}e_a\texttt{)}$, proof typing leverages subtyping to check that that the function e_f can actually map the argument e_a to a result. It learns a lower bound on the type of the result α associated with that argument.

Rule 4.2.9. For a recursive expression $\texttt{loop(}e\texttt{)}$, proof typing constructionsa complex type containing an implication constrained by a least fixed point type. First, it ensures that argument of the fixpoint combinator e is indeed a singe path function. Then, by leveraging subtyping, it finds a lower bound for every path in the body of e, represented by a sequence of worlds $\vec{\Omega}$, all associated with a variable implication, $\alpha_l\texttt{->}\alpha_r$. Using all these worlds, it delegates to *fixpoint duality*, as in definition 8.35, it order to construct the cases of a relational least fixpoint type, representing the dual of the greatest fixpoint of implication under intersected over its worlds. Finally, it reconstructs a type for a function, by wrapping the relational type in an existential constraint, and wrapping a generalized implication around that.

Rule 4.2.10. For an annotated definition expression $\texttt{let}\ x\texttt{:}\tau_a\ \texttt{=}\ e\ \texttt{in}\ e'$, proof typing checks the definition's source e against the annotation τ_a, and adds the annotation to the typing environment when checking the contintuation e'.

Definition 4.2. $\boxed{\Gamma \vdash e : \tau \dashv \Omega}$ Proof Typing

$$\frac{}{\Gamma \vdash \texttt{@} : \texttt{@} \dashv \Omega}\ [1] \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \dashv \Omega}\ [2] \qquad \frac{\Gamma \vdash e : \tau \dashv \Omega}{\Gamma \vdash \texttt{<}l\texttt{>}e : \texttt{<}l\texttt{>}\tau \dashv \Omega}\ [3] \qquad \frac{\vec{r} = \epsilon}{\Gamma \vdash \vec{r} : \texttt{TOP} \dashv \Omega}\ [4]$$

$$\frac{\Gamma \vdash \vec{r} : \tau \dashv \Omega \qquad \Gamma \vdash e : \tau' \dashv \Omega' \qquad \Omega \preceq \Omega'}{\Gamma \vdash \vec{r}\ \texttt{\$}l\texttt{=>}e : \tau\ \texttt{\&}\ l\texttt{->}\tau' \dashv \Omega'}\ [5] \qquad \frac{\vec{\alpha}, \Delta, \Gamma \vdash \vec{f} \blacktriangle \vec{\pi}, \vec{\eta} \qquad \Gamma \vdash \vec{\pi} \cong \vec{\tau}}{\Gamma \vdash \vec{f} : \text{inter}(\vec{\tau}) \dashv \vec{\alpha}, \Delta}\ [6]$$

$$\frac{\Gamma \vdash e : \tau_0 \dashv \Omega \qquad \tau_0 <: l\texttt{->}\alpha \dashv \Omega' \qquad \Omega \preceq \Omega'}{\Gamma \vdash e\texttt{.}l : \alpha \dashv \Omega'}\ [7] \qquad \frac{\begin{array}{c}\Gamma \vdash e_f : \tau_f \dashv \Omega \qquad \Gamma \vdash e_a : \tau_a \dashv \Omega' \qquad \Omega \preceq \Omega' \\ \tau_f <: \tau_a\texttt{->}\alpha \dashv \Omega'' \qquad \Omega' \preceq \Omega''\end{array}}{\Gamma \vdash e_f\texttt{(}e_a\texttt{)} : \alpha \dashv \Omega''}\ [8]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \alpha^{+}\texttt{->}\tau \dashv \vec{\alpha}, \Delta \\ \forall \vec{\alpha}'\ \Delta'.\ (\vec{\alpha}\ \vec{\alpha}', \Delta\ \Delta') \in \vec{\Omega} \implies \tau <: \alpha_l\texttt{->}\alpha_r \dashv \vec{\alpha}\ \vec{\alpha}', \Delta\ \Delta' \\ \exists \Omega.\ \Omega \in \vec{\Omega} \qquad \text{FTV}(\Gamma) \vdash \nu\alpha^{+}.\vec{\Omega}.\alpha_l \to \alpha_r \fallingdotseq \mu\alpha^{-}.\vec{\tau}\end{array}}{\Gamma \vdash \texttt{loop(}e\texttt{)} : \texttt{ALL[}\alpha_x\texttt{]}\alpha_x\texttt{->EXI[}\alpha_y\texttt{;}\alpha_x\texttt{*}\alpha_y\texttt{<:LFP[}\alpha^{-}\texttt{]}\text{union}(\vec{\tau})\texttt{]}\alpha_y \dashv \vec{\alpha}, \Delta}\ [9]$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \dashv \Omega \qquad \tau <: \tau_a \dashv \Omega' \qquad \Omega \preceq \Omega' \\ \Gamma\ x : \tau_x \vdash e' : \tau' \dashv \Omega'' \qquad \Omega' \preceq \Omega''\end{array}}{\Gamma \vdash \texttt{let}\ x\texttt{:}\tau_a\ \texttt{=}\ e\ \texttt{in}\ e' : \tau' \dashv \Omega''}\ [10]$$

Proof subtyping, as in definition 4.3, checks the viability of one type subtyping another type.
TODO: explain the overall predicate
TODO: explain the algorithmic interpretation The order of the rules is critical to ensure that easier constraints are generated. To that end, cases that strengthen the left side or weaken the right side occur before rules that weaken the left side or strengthen the right side.

Due to the complexity of types along with two positions for types to occur in subtyping, there are many rules needed to define the proof system of subtyping. For clarity, we show only a subset of the rules in this section in order to explain the essence of the system. We leave the remaining rules in the appendix, section 8, definition 8.6. The remaining rules include duals and other forms that adhere to similar proof strategies as the rules shown here, as well as additional rules for increased precision.

Rule 4.3.1. For reflexive subtyping, up to alpha renaming, proof subtyping simply holds without any updates to the world.

Rule 4.3.2. For a lower bound least fixed point type, $\texttt{LFP[}\alpha\texttt{]}\,\tau_l$, proof subtyping attempts a proof by induction, by unrolling the least fixed point and replacing its self referencing variable with the upper bound.

Rule 4.3.3. For an upper bound difference type, $\tau_r\backslash\eta$, proof subtyping, checks that the lower bound subtypes the positive type τ_r and does not subtype the subtracted type η. When relying on negation, one must be careful to preserve soundness. We ensure that proof subtyping is complete for an upper bound of limited form η with no free type variables, which means its negation is sound. We will make the limited notion of completeness precise in section 5.

Rule 4.3.4. For a lower bound union type, $\tau_{ll}\,|\,\tau_{lr}$, proof subtyping ensures that both parts of the lower bound subtype the upper bound.

The dual of this rule is the one for an upper bound intersection type, in the continued definition 8.6.

Rule 4.3.5. for a lower bound existential type, $\texttt{EXI[}\vec{\alpha}\ \Delta\texttt{]}\,\tau_l$, proof subtyping first searches subtyping constraints that satisfy the existential qualifiers Δ, and then checks the body of the existential against the upper bound, under the previously learned subtyping constraints, with the restriction that the existentially quantified variables are closed.

This is analogous to a proof in predicate logic, where the prover declares a variable from an existential statement and cannot see its particular value.

The dual of this rule is the one for an upper bound universal type, in the continued definition 8.6.

The universal dual is analogous to a proof in predicate logic, where the prover generalized from an arbitrary variable to a universal statement.

Rule 4.3.6. For a lower bound closed variable α, proof subtyping finds a strict interpretation of the variable and checks it against the upper bound. Since the subtyping constraints may contain relational constraints, it factors the relational constraints to find constraints over single variables, which it used to construct the strict interpretation inter($\vec{\tau}$).

The dual of this rule is the one for an upper bound closed variable, in the continued definition 8.6.

Rule 4.3.7. For a upper bound open variable α, proof subtyping finds a lenient interpretation of the variable and checks it against the lower bound. If safe, it updates the worlds with the subtyping. To find a lenient interpretation in simple constraints, it searches the world for the first upper bounds that are not closed variables. It also looks for relational upper bounds of pattern type containing the open variable α. Additionally, it learns constraints on closed variables $\Delta^{\dagger}$ that are transitive upper bounds of the open variable α.

The dual of this rule is the one for a lower bound open variable, in the continued definition 8.6.

Rule 4.3.8. For an upper bound existential type, proof subtyping solves the existential's target τ_r against the lower bound, in order to learn necessary constraints on the bound variables for solving the qualifiers. It then uses the world generated from that to solve the existential's qualifiers Δ.

It is analogous to the proof method in predicate logic, in which the prover may choose an expression to be a witness for an existential statement.

The dual of this rule is the one for a lower bound universal, in the continued definition 8.6. The universal dual is analogous to the proof method in predicate logic, in which the prover instantiates a universal statement with an expression of their choice.

Rule 4.3.9. For an upper bound union type, proof subtyping checks that the left part of the union holds against the lower bound.

The dual of this rule is one for an upper bound intersection, in the continued definition 8.6.

Rule 4.3.10. For an upper bound union type, proof subtyping also checks that the right part of the union holds against the lower bound.

The dual of this rule is one for an upper bound intersection, in the continued definition 8.6.

Rule 4.3.11. For a lower bound difference type, proof subtyping simply checks that the positive type subtypes the upper bound union with the subtracted type.

Rule 4.3.12. right decomposable least fixed point.

The right induction case attempts to unroll the inductive type just enough to unify with the left type. To avoid potential infinite unrolling the procedure relies on a heuristic to see if the left type's pattern lines up with parts of the the inductive type that are guaranteed to be well-founded. If the left type is a pattern with variables that prevents it from being reduced, then the procedure checks if it is well-formed, meaning it could be solved if more information were specified, and then the unsolved relational constraint is added to the subtyping environment.

Rule 4.3.13. right assumed least fixed point.

Rule 4.3.14. right consistent least fixed point.

Rule 4.3.15. tags

Rule 4.3.16. records

Rule 4.3.17. implications

Definition 4.3. $\boxed{\tau <: \tau \dashv \Omega}$ Proof Subtyping

$$\frac{}{\tau <: \tau \dashv \Omega}\ [1] \qquad \frac{\tau_l[\alpha/\tau_r] <: \tau_r \dashv \Omega}{\texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau_l <: \tau_r \dashv \Omega}\ [2]$$

$$\frac{\begin{array}{c}\tau_l <: \tau_r \dashv \Omega \qquad \mathrm{FTV}(\eta) \subseteq \epsilon \\ \nexists \Omega'.\ \tau_l <: \eta \dashv \Omega' \land \Omega \preceq \Omega'\end{array}}{\tau_l <: \tau_r \texttt{\\}\eta \dashv \Omega}\ [3] \qquad \frac{\tau_{ll} <: \tau_r \dashv \Omega \qquad \tau_{lr} <: \tau_r \dashv \Omega' \qquad \Omega \preceq \Omega'}{\tau_{ll}\,\texttt{|}\,\tau_{lr} <: \tau_r \dashv \Omega'}\ [4]$$

$$\frac{\begin{array}{c}\Delta \dashv \vec{\alpha}_w, \Delta_w \qquad \vec{\alpha} \# \tau_r \\ \tau_l <: \tau_r \dashv \Omega \qquad (\vec{\alpha}_w\ \vec{\alpha}, \Delta_w) \preceq \Omega\end{array}}{\texttt{EXI}\,\texttt{[}\vec{\alpha}\ \Delta\texttt{]}\,\tau_l <: \tau_r \dashv \Omega}\ [5] \qquad \frac{\begin{array}{c}\alpha \in \vec{\alpha} \qquad \mathrm{factor}(\Delta, \alpha) = \Delta_f \qquad \vec{\alpha}, (\Delta\ \Delta_f) \vdash \alpha <: \vec{\tau} \\ \mathrm{inter}(\vec{\tau}) <: \tau_r \dashv \vec{\alpha}', \Delta' \qquad (\vec{\alpha}, \Delta\ \Delta_f) \preceq (\vec{\alpha}', \Delta')\end{array}}{\alpha <: \tau_r \dashv \vec{\alpha}', \Delta'}\ [6]$$

$$\frac{\begin{array}{c}\alpha \notin \vec{\alpha} \\ \vec{\alpha}, \Delta \vdash \alpha/\tau_l \dagger \Delta^\dagger \qquad \vec{\alpha}, \Delta \vdash \alpha <:^\dagger \vec{\tau} \qquad \vec{\alpha}, \Delta \vdash \alpha/\tau_l <:^\sharp \Delta^\sharp \\ \Delta^\dagger\ \tau_l\texttt{<:}\mathrm{inter}(\vec{\tau}) \dashv \vec{\alpha}', \Delta' \qquad (\vec{\alpha}, \Delta\ \Delta^\sharp) \preceq (\vec{\alpha}', \Delta')\end{array}}{\tau_l <: \alpha \dashv (\vec{\alpha}', \Delta'\ \tau_l\texttt{<:}\alpha)}\ [7] \qquad \frac{\tau_l <: \tau_r \dashv \Omega \qquad \Delta \dashv \Omega' \qquad \Omega \preceq \Omega'}{\tau_l <: \texttt{EXI}\,\texttt{[}\vec{\alpha}\ \Delta\texttt{]}\,\tau_r \dashv \Omega'}\ [8]$$

$$\frac{\tau_l <: \tau_{rl} \dashv \Omega}{\tau_l <: \tau_{rl}\,\texttt{|}\,\tau_{rr} \dashv \Omega}\ [9] \qquad \frac{\tau_l <: \tau_{rr} \dashv \Omega}{\tau_l <: \tau_{rl}\,\texttt{|}\,\tau_{rr} \dashv \Omega}\ [10] \qquad \frac{\tau_l <: \tau_r\,\texttt{|}\,\eta \dashv \Omega}{\tau_l\texttt{\\}\eta <: \tau_r \dashv \Omega}\ [11]$$

$$\frac{\Omega \vdash \tau_l \circlearrowleft \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau_r \qquad \tau_l <: \tau_r[\alpha/\texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau_r] \dashv \Omega' \qquad \Omega \preceq \Omega'}{\tau_l <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau_r \dashv \Omega'}\ [12]$$

$$\frac{\begin{array}{c}\exists \alpha.\ \alpha \in \mathrm{FTV}(\phi) \land \alpha \in \vec{\alpha} \qquad \phi <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau \cong \phi' <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau' \\ \Delta \vdash \phi' <: \tau_n \sim \qquad \tau_n <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau' \vdash \Omega \qquad (\vec{\alpha}, \Delta) \preceq \Omega\end{array}}{\phi <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau \dashv \Omega}\ [13]$$

$$\frac{\begin{array}{c}\forall \alpha.\ \alpha \in \mathrm{FTV}(\phi) \implies \alpha \notin \vec{\alpha} \qquad \vdash \phi <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau \star \\ \vec{\alpha} \vdash \Delta \wr \mathrm{FTV}(\phi) \dashv \vec{\delta} \qquad \phi[\vec{\delta}] <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau \dashv \vec{\alpha}', \Delta' \qquad (\vec{\alpha}, \Delta) \preceq (\vec{\alpha}', \Delta')\end{array}}{\phi <: \texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau \dashv \vec{\alpha}', \Delta'\ \texttt{;}\phi\texttt{<:}\texttt{LFP}\,\texttt{[}\alpha\texttt{]}\,\tau}\ [14]$$

$$\frac{\tau_l <: \tau_r \dashv \Omega}{\texttt{<}l\texttt{>}\tau_l <: \texttt{<}l\texttt{>}\tau_r \dashv \Omega}\ [15] \qquad \frac{\tau_l <: \tau_r \Omega}{l\,\texttt{->}\tau_l <: l\,\texttt{->}\tau_r \dashv \Omega}\ [16] \qquad \frac{\tau_{rl} <: \tau_{ll} \dashv \Omega \qquad \tau_{lr} <: \tau_{rr} \dashv \Omega' \qquad \Omega \preceq \Omega'}{\tau_{ll}\texttt{->}\tau_{lr} <: \tau_{rl}\texttt{->}\tau_{rr} \dashv \Omega'}\ [17]$$

5 CORRECTNESS

TODO: introduce model typing and soundness properties

Definition 5.1. $\boxed{\vec{\delta}, \Gamma \vDash e : \tau}$ Model typing

$$\frac{\alpha/\tau \in \vec{\delta} \qquad \vec{\delta}, \Gamma \vDash e : \tau}{\vec{\delta}, \Gamma \vDash e : \alpha} \qquad \frac{\alpha/\tau \notin \vec{\delta} \qquad \vec{\delta}, \Gamma \vDash e : \texttt{TOP}}{\vec{\delta}, \Gamma \vDash e : \alpha} \qquad \frac{}{\vec{\delta}, \Gamma \vDash \texttt{@} : \texttt{@}} \qquad \frac{\vec{\delta}, \Gamma \vDash e : \tau}{\vec{\delta}, \Gamma \vDash \texttt{<}l\texttt{>}e : \texttt{<}l\texttt{>}\tau}$$

$$\frac{\begin{array}{c}\texttt{\$}l\texttt{=>}v \in G \qquad \vec{\delta}, \Gamma \vDash v : \tau \\ \forall v'.\ \texttt{\$}l\texttt{=>}v' \in G \implies v' = v\end{array}}{\vec{\delta}, \Gamma \vDash G : l\texttt{->}\tau}$$

$$\frac{\begin{array}{c}\vec{\delta}, \Gamma \sqcup \Gamma' \vDash p \between \tau_p \qquad \vec{\delta}, \Gamma \sqcup \Gamma' \vDash e : \tau_r \\ \forall e.\ \vec{\delta}, \Gamma \vDash e : \tau_l \implies \vec{\delta}, \Gamma \vDash e : \tau_p \land (\forall \tau_n\ \tau.\ \vec{\delta}, \Gamma \vDash F : \tau_n\texttt{->}\tau \implies \neg(\vec{\delta}, \Gamma \vDash e : \tau_n))\end{array}}{\vec{\delta}, \Gamma \vDash F\texttt{*}p\texttt{=>}e : \tau_l\texttt{->}\tau_r}$$

$$\frac{\vec{\delta}, \Gamma \vDash F : \tau_l\texttt{->}\tau_r}{\vec{\delta}, \Gamma \vDash F\texttt{*}p\texttt{=>}e : \tau_l\texttt{->}\tau_r} \qquad \frac{\vec{\delta}, \Gamma \vDash e : \tau_l}{\vec{\delta}, \Gamma \vDash e : \tau_l \texttt{|} \tau_r} \qquad \frac{\vec{\delta}, \Gamma \vDash e : \tau_r}{\vec{\delta}, \Gamma \vDash e : \tau_l \texttt{|} \tau_r}$$

$$\frac{\vec{\delta}, \Gamma \vDash e : \tau_l \qquad \vec{\delta}, \Gamma \vDash e : \tau_r}{\vec{\delta}, \Gamma \vDash e : \tau_l \texttt{\&} \tau_r} \qquad \frac{\vec{\delta}, \Gamma \vDash e : \tau_l \qquad \neg\,(\vec{\delta}, \Gamma \vDash e : \tau_r)}{\vec{\delta}, \Gamma \vDash e : \tau_l \texttt{\textbackslash} \tau_r} \qquad \frac{\vec{\delta} \sqcup \vec{\delta}' \vDash Q \qquad \vec{\delta} \sqcup \vec{\delta}' \vDash e : \tau}{\vec{\delta} \vDash e : \texttt{EXI[}A\ \ Q\texttt{]}\tau}$$

$$\frac{\forall \vec{\delta}'.\ \vec{\delta} \sqcup \vec{\delta}' \vDash Q \implies \vec{\delta} \sqcup \vec{\delta}', \Gamma \vDash e : \tau}{\vec{\delta}, \Gamma \vDash e : \texttt{ALL[}A\ \ Q\texttt{]}\tau} \qquad \frac{\vec{\delta}\ \alpha/\texttt{LFP[}\alpha\texttt{]}\tau \vDash e : \tau}{\vec{\delta}, \Gamma \vDash e : \texttt{LFP[}\alpha\texttt{]}\tau} \qquad \frac{x : \tau \in \Gamma}{\vec{\delta}, \Gamma \vDash x : \tau}$$

$$\frac{\vec{\delta}, \vec{\sigma} \vDash \Gamma \qquad e[\vec{\sigma}] \rightsquigarrow e'[\vec{\sigma}'] \qquad \vec{\delta}, \vec{\sigma}' \vDash \Gamma' \qquad \vec{\delta}, \Gamma' \vDash e' : \tau}{\vec{\delta}, \Gamma \vDash e : \tau}$$

Definition 5.2. $\boxed{\vec{\delta} \vDash \tau <: \tau}$ Model Subtyping

$$\frac{\forall e\ \Gamma.\ \vec{\delta}, \Gamma \vDash e : \tau_l \implies \vec{\delta}, \Gamma \vDash e : \tau_r}{\vec{\delta} \vDash \tau_l <: \tau_r}$$

Definition 5.3. $\boxed{\vec{\delta} \vDash Q}$ Model Sequence Subtyping

$$\frac{}{\vec{\delta} \vDash \epsilon} \qquad \frac{\vec{\delta} \vDash Q \qquad \vec{\delta} \vDash \tau_l <: \tau_r}{\vec{\delta} \vDash Q\ \texttt{.}\tau_l\texttt{<:}\tau_r}$$

6 EXPERIMENTS

TODO: develop 12 tree/list experiments

7 RELATED WORK

Tree interpolation. Exemplified by CHC duality solver.

Hindley-Milner type inference. Exemplified by ML.

Logic programming. Exemplified by Prolog.
Similar: both have backchaining.
Different: RLT is fully declarative, lacks negations, but has implication.
Different: RLT allows comparing inductive relations via subtyping.

Semantic subtyping. Exemplified by XDuce and CDuce. complete subtyping.
Similar: set-like combinators: union and intersection.
Different: RLT uses rigid syntactic rules; incomplete subtyping.
The terminology "semantic subtyping" vs "syntactic subtyping" are confusing terms. "semantics subtyping" means the semantics of types is determined indirectly by the semantics of another structure. "syntactic subtyping" means the semantics of types is determined directly by the type structure

Extrinsic typing. Exemplified by Typescript, which is unsound. Maybe not as lenient? The static behavior of a program is not necessarily specified/prescribed; it may be over-approximated from the program composition. Intrinsic vs extrinsic is orthogonal to static vs dynamic, although static and dynamic are often used to mean the former. All modern languages use a combination of static and dynamic type checking. The term "dynamically typed" some times refers to a language that doesn't prescribe static meaning, even if it uses both static and dynamic type checking. The term "extrinsic typing" is less ambiguous.

Refinement Types. Exemplified by Refinement ML. Base types with intersections and subtyping.

Predicate Subtyping. Exemplified by Liquid Types. An extension of refinement types.
Similar: both use type inference to infer expressive relational properties.
Different: RLT starts with an invalid post-condition, then weakens return type to a valid post-condition from outside in by expanding unions.
Different: RLT starts with an invalid pre-condition, then strengthens parameter type to a valid pre-condition from inside out by adding intersections.
Different: Liquid types starts with an invalid post-condition, then uses iterative weakening by dropping conjunctions until a valid post-condition is reached.

Abstraction Refinement. Similar: type unification over subtyping resembles abstraction refinement where solving for variables and failing on different sides of the subtyping relation corresponds to solving with the abstractor vs solving with the refiner.

Craig interpolation. Similar: extracting an inductive type with unions and intersections from a recursive program without needing to specify a predicate universe might be similar to craig interpolation.

PDR.. exemplified by IC3.
Similar: RLT infers abstract type for return type, then safely constrains the variables in previous step (fix's antecedent) to subtype the least fixed point. This lazily propagates the type for the last step to the previous steps. This is safe as antecedent is stronger than consequent at any step. Seems similar to the notion in PDR of propagating negation of loss points to previous steps.
Different: RLT isn't cartesian

8 APPENDIX

Definition 8.1. Internals

$$
\begin{array}{rcl}
\Gamma & ::= & \epsilon \mid \Gamma\ x : \tau \\
\\
\vec{\Omega} & ::= & \epsilon \mid \vec{\Omega}\ \Omega \\
\Omega & ::= & \vec{\alpha}, \Delta \\
\\
\vec{\tau} & ::= & \epsilon \mid \vec{\tau}\ \tau \\
\vec{\phi} & ::= & \epsilon \mid \vec{\phi}\ \phi \\
\\
\vec{\pi} & ::= & \epsilon \mid \vec{\pi}\ \pi \\
\pi & ::= & \vec{\alpha}, \Delta, \tau \texttt{->} \tau \\
\\
\vec{\delta} & ::= & \epsilon \mid \vec{\delta}\ \delta \\
\delta\ \alpha/\tau & &
\end{array}
$$

Definition 8.2. Sugared Expressions

$$
\begin{array}{rcl}
e & ::= & \ldots \mid e \texttt{,} e \mid e \texttt{|>} e \mid \texttt{let}\ x\ z \texttt{=} e\ \texttt{in}\ e \mid \texttt{(} e \texttt{)} \\
z & ::= & \epsilon \mid \texttt{:} \tau \\
p & ::= & \ldots \mid p \texttt{,} p \mid \texttt{(} p \texttt{)}
\end{array}
$$

Definition 8.3. Values

$$
\begin{array}{rcl}
v & ::= & \texttt{@} \mid \texttt{<}l\texttt{>}\ v \mid \vec{g} \mid v \texttt{,} v \mid \texttt{(} v \texttt{)} \mid \vec{f} \\
\vec{g} & ::= & \epsilon \mid \vec{g}\ g \\
g & ::= & \texttt{\$}l\texttt{=>}v \\
\\
\vec{\sigma} & ::= & \epsilon \mid \vec{\sigma}\ \sigma \\
\sigma & ::= & x/v
\end{array}
$$

Definition 8.4. $\boxed{e \rightsquigarrow e}$ Sugared Progression

$$
\frac{e_b \texttt{(} e_a \texttt{)} \rightsquigarrow e'}{e_a \texttt{|>} e_b \rightsquigarrow e'}
\qquad
\frac{e \rightsquigarrow e'}{\texttt{(} e \texttt{)} \rightsquigarrow e'}
\qquad
\frac{\texttt{\$left=>}e_l\ \texttt{\$right=>}e_r \rightsquigarrow e'}{e_l \texttt{,} e_r \rightsquigarrow e'}
$$

$$
\frac{\texttt{(\$ <true> @=>}e_t\ \texttt{\$ <false> @=>}e_f\texttt{)(}e_c\texttt{)} \rightsquigarrow e'}{\texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_f \rightsquigarrow e'}
\qquad
\frac{\texttt{(\$}x\texttt{=>}e_k\texttt{)(}e\texttt{)} \rightsquigarrow e'}{\texttt{let}\ x \texttt{:} \tau \texttt{=} e\ \texttt{in}\ e_k \rightsquigarrow e'}
$$

Definition 8.5. Sugared Types

$$
\begin{array}{rcl}
\tau & ::= & \ldots \mid \texttt{TOP} \mid \texttt{BOT} \mid \texttt{(} \tau \texttt{)} \\
\phi & ::= & \ldots \mid \texttt{(} \phi \texttt{)}
\end{array}
$$

Definition 8.6. $\boxed{\tau <: \tau \dashv M, \Delta}$ Continued Proof Subtyping

$$\frac{}{\texttt{BOT} <: \tau_r \dashv M, \Delta} \qquad \frac{}{\tau_l <: \texttt{TOP} \dashv M, \Delta} \qquad \frac{\tau_l <: (l{:}\tau_{rl})\&(l{:}\tau_{rr}) \dashv M, \Delta}{\tau_l <: l{:}(\tau_{rl}\&\tau_{rr}) \dashv M, \Delta}$$

$$\frac{\begin{array}{cc}\tau_l <: \tau_{rl} \dashv M_0, \Delta_0 & M_0 \preceq M_1 \\ \Delta_0 \preceq \Delta_1 & \tau_l <: \tau_{rr} \dashv M_1, \Delta_1\end{array}}{\tau_l <: \tau_{rl}\&\tau_{rr} \dashv M_1, \Delta_1} \qquad \frac{\tau_l <: (\tau_{ra}\texttt{->}\tau_{rc})\&(\tau_{rb}\texttt{->}\tau_{rc}) \dashv M, \Delta}{\tau_l <: \tau_{ra}|\tau_{rb}\texttt{->}\tau_{rc} \dashv M, \Delta}$$

$$\frac{\tau_l <: (\tau_{ra}\texttt{->}\tau_{rb})\&(\tau_{ra}\texttt{->}\tau_{rc}) \dashv M, \Delta}{\tau_l <: \tau_{ra}\texttt{->}\tau_{rb}\&\tau_{rc} \dashv M, \Delta} \qquad \frac{\begin{array}{ccc} & Q \dashv M_0, \Delta_0 \quad A \# \tau_l & \\ M_0 \sqcup A \preceq M_1 & \Delta_0 \preceq \Delta_1 & \tau_l <: \tau_r \dashv M_1, \Delta_1\end{array}}{\tau_l <: \texttt{ALL[}A\ Q\texttt{]}\tau_r \dashv M_1, \Delta_1}$$

$$\frac{\begin{array}{ccc} & \alpha \in M_0 \quad \Delta_0 \vdash \mathrm{T} <: \alpha & \\ M_0 \preceq M_1 & \Delta_0 \preceq \Delta_1 & \tau_l <: |(\mathrm{T}) \dashv M_1, \Delta_1\end{array}}{\tau_l <: \alpha \dashv M_1, \Delta_1}$$

$$\frac{\begin{array}{cccc} & \alpha \notin M_0 & M_0, \Delta_0 \vdash \Delta_m <:^{\sharp} \alpha/\tau_r & \\ M_0, \Delta_0 \vdash \mathrm{T} <:^{\dagger} \alpha & M_0 \preceq M_1 & \Delta_0 \sqcup \Delta_m \preceq \Delta_1 & |(\mathrm{T}) <: \tau_r \dashv M_1, \Delta_1\end{array}}{\alpha <: \tau_r \dashv M_1, \Delta_1\ \alpha <: \tau_r}$$

$$\frac{\begin{array}{ccc}\tau_l <: \tau_r \dashv M_0, \Delta_0 & & \\ M_0 \preceq M_1 & \Delta_0 \preceq \Delta_1 & Q \dashv M_1, \Delta_1\end{array}}{\texttt{ALL[}A\ \ Q\texttt{]}\tau_l <: \tau_r \dashv M_1, \Delta_1} \qquad \frac{\tau_{ll} <: \tau_r \dashv M, \Delta}{\tau_{ll}\&\tau_{lr} <: \tau_r \dashv M, \Delta} \qquad \frac{\tau_{lr} <: \tau_r \dashv M, \Delta}{\tau_{ll}\&\tau_{lr} <: \tau_r \dashv M, \Delta}$$

Definition 8.7. $\boxed{\Omega \preceq \Omega}$ World Ordering

$$\frac{}{(\vec{\alpha}, \Delta) \preceq (\vec{\alpha}\ \vec{\alpha}', \Delta\ \Delta')}$$

Definition 8.8. $\boxed{\vec{\alpha}, \Delta, \Gamma \vdash \vec{f} \blacktriangle \vec{\pi}, \vec{\eta}}$ Function Lifting

$$\frac{}{\vec{\alpha}, \Delta, \Gamma \vdash \epsilon \blacktriangle \epsilon, \epsilon} \qquad \frac{\begin{array}{c}\vec{\alpha}, \Delta, \Gamma \vdash \vec{f} \blacktriangle \vec{\pi}, \vec{\eta} \qquad p : \phi \dashv \Gamma' \qquad \mathrm{diff}(\phi, \vec{\eta}) = \tau_l \\ \forall \vec{\alpha}'\ \Delta'\ \tau_r.\ (\vec{\alpha}\ \vec{\alpha}', \Delta\ \Delta', \tau_l\texttt{->}\tau_r) \in \vec{\pi}' \implies \Gamma\ \Gamma' \vdash e : \tau_r \dashv \vec{\alpha}\ \vec{\alpha}', \Delta\ \Delta' \\ \exists \pi.\ \pi \in \vec{\pi}' \qquad \mathrm{close}(\phi) = \eta\end{array}}{\vec{\alpha}, \Delta, \Gamma \vdash \vec{f}\ \texttt{\$}p\texttt{=>}e \blacktriangle \vec{\pi}\ \vec{\pi}', \vec{\eta}\ \eta}$$

Definition 8.9. $\boxed{\Omega \vdash \tau \circlearrowleft \tau}$ Decomposable

$$\frac{\textbf{TODO: ...}}{\Omega \vdash \tau \circlearrowleft \tau}$$

Definition 8.10. $\boxed{\tau <: \tau \cong \tau <: \tau}$ Normal Constraint Congruence

$$\frac{\textbf{TODO: ...}}{\tau <: \tau \cong \tau <: \tau}$$

Definition 8.11. $\boxed{\Delta \vdash \tau' <: \tau \sim}$ Normal Constraint Entailment

$$\frac{\textbf{TODO: ...}}{\Delta \vdash \tau' <: \tau \sim}$$

Definition 8.12. $\boxed{\vec{\alpha} \vdash \Delta \wr \vec{\alpha} \dashv \vec{\delta}}$ Modulo Type Solution

$$\frac{\textbf{TODO: ...}}{\vec{\alpha} \vdash \Delta \wr \vec{\alpha} \dashv \vec{\delta}}$$

Definition 8.13. $\boxed{\vdash \tau_l <: \tau \star}$ Constraint Consistency

$$\frac{\textbf{TODO: ...}}{\vdash \tau_l <: \mathsf{LFP}[\alpha]\tau_r \star}$$

Definition 8.14. $\boxed{\mathrm{A} \mathbin{\#} \tau}$ Fresh variables

$$\frac{\forall \alpha.\ \alpha \in \mathrm{A} \implies \alpha \notin \mathrm{FV}(\tau)}{\mathrm{A} \mathbin{\#} \tau}$$

Definition 8.15. $\boxed{\tau <: \tau \dashv Z}$ (Proof universe subtyping)

$$\frac{\langle M, \Delta \rangle \in Z \qquad \forall M\ \Delta.\ \tau_l <: \tau_r \dashv M, \Delta \iff \langle M, \Delta \rangle \in Z}{\tau_l <: \tau_r \dashv Z}$$

Definition 8.16. $\boxed{Q \dashv M, \Delta}$

$$\frac{Q \dashv M_0, \Delta_0 \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_l <: \tau_r \dashv M_1, \Delta_1}{Q\,\mathsf{.}\,\tau_l \mathsf{<:} \tau_r \dashv M_1, \Delta_1} \qquad \frac{}{M, \Delta \vdash \epsilon}$$

Definition 8.17. (Collection)

$$C \ ::= \ \epsilon \ \mid \ C\ c$$

Definition 8.18. $\boxed{C\ C = C}$ Concatenation

$$\begin{array}{rcll} C\ \epsilon & = & C & \triangleright\ \textit{empty} \\ C\ (C'\ c) & = & (C\ C')\ c & \triangleright\ \textit{step} \end{array}$$

Definition 8.19. $\boxed{C \diamond C = C}$ Filter

$$\begin{array}{rcl} C \diamond \epsilon & = & \epsilon \\ C \diamond (C'\ c) & = & \begin{cases} (C \diamond C')\ c & \text{if } c \in C \\ (C \diamond C') & \text{otherwise} \end{cases} \end{array}$$

Definition 8.20. $\boxed{c \in C}$ Collection Containment

$$\frac{}{c \in C\ c} \qquad \frac{c \neq c' \qquad c \in C}{c \in C\ c'}$$

Definition 8.21. $\boxed{C \preceq C}$

$$\frac{}{C \preceq C} \qquad \frac{C \preceq C'}{C \preceq C'\ c}$$

Definition 8.22. $\boxed{\text{union}(\mathrm{T}) = \tau}$ Collective Union

$$\begin{array}{rcll} \text{union}(\epsilon) & = & \texttt{BOT} & \triangleright\ \textit{empty} \\ \text{union}(\mathrm{T}\ \tau) & = & \text{union}(\mathrm{T}) \mid \tau & \triangleright\ \textit{step} \end{array}$$

Definition 8.23. $\boxed{\text{inter}(\mathrm{T}) = \tau}$ Collective Intersection

$$\begin{array}{rcll} \text{inter}(\epsilon) & = & \texttt{TOP} & \triangleright\ \textit{empty} \\ \text{inter}(\mathrm{T}\ \tau) & = & \text{inter}(\mathrm{T}) \& \tau & \triangleright\ \textit{step} \end{array}$$

Definition 8.24. $\boxed{M, \Delta \vdash m <:^{\sharp} \alpha}$ Lower closed subtyping

$$\frac{m \in M \qquad m <: \alpha \in \Delta}{M, \Delta \vdash m <:^{\sharp} \alpha}$$

Definition 8.25. $\boxed{M, \Delta \vdash \Delta <:^{\sharp} \alpha/\tau}$ Universal lower closed subtyping

$$\frac{\forall m\ \tau.\ m <: \tau \in \Delta' \iff M, \Delta \vdash m \in M \wedge m <: \alpha \in \Delta}{M, \Delta \vdash \Delta' <:^{\sharp} \alpha/\tau}$$

Definition 8.26. $\boxed{M, \Delta \vdash \alpha/\tau <:^{\sharp} \Delta}$ Universal upper closed subtyping

$$\frac{\forall \tau\ m.\ \tau <: m \in \Delta' \iff m \in M \wedge \alpha <: m \in \Delta}{M, \Delta \vdash \alpha <:^{\sharp} \Delta'}$$

Definition 8.27. $\boxed{M, \Delta \vdash \tau <:^{\dagger} \alpha}$ Lower transitive subtyping

$$\frac{\tau \notin M \qquad \tau <: \alpha \in \Delta}{M, \Delta \vdash \tau <:^{\dagger} \alpha} \qquad\qquad \frac{m \in M \qquad M, \Delta \vdash \tau <:^{\dagger} m \qquad m <: \alpha \in \Delta}{M, \Delta \vdash \tau <:^{\dagger} \alpha}$$

Definition 8.28. $\boxed{M, \Delta \vdash \mathrm{T} <:^{\dagger} \alpha}$ Universal lower transitive subtyping

$$\frac{\forall \tau.\ \tau \in \mathrm{T} \iff M, \Delta \vdash \tau <:^{\dagger} \alpha}{M, \Delta \vdash \mathrm{T} <:^{\dagger} \alpha}$$

Definition 8.29. $\boxed{M, \Delta \vdash \alpha <:^{\dagger} \tau}$ Upper transitive subtyping

$$\frac{\tau \notin M \qquad \alpha <: \tau \in \Delta}{M, \Delta \vdash \alpha <:^{\dagger} \tau} \qquad\qquad \frac{m \in M \qquad \alpha <: m \in \Delta \qquad M, \Delta \vdash m <:^{\dagger} \tau}{M, \Delta \vdash \alpha <:^{\dagger} \tau}$$

Definition 8.30. $\boxed{M, \Delta \vdash \alpha <:^{\dagger} \mathrm{T}}$ Universal upper transitive subtyping

$$\frac{\forall \tau.\ \tau \in \mathrm{T} \iff M, \Delta \vdash \alpha <:^{\dagger} \mathrm{T}}{M, \Delta \vdash \alpha <:^{\dagger} \mathrm{T}}$$

Definition 8.31. $\boxed{M, \Delta \vdash \alpha \dagger \tau <: \tau}$ Relational subtyping

$$\frac{\alpha \neq \tau_l \qquad \alpha \in \mathrm{FTV}(\tau_l)}{\Delta \vdash \alpha \dagger \tau_l <: \tau_r}$$

Definition 8.32. $\boxed{M, \Delta \vdash \alpha/\tau \dagger \Delta}$ Relational substitution

$$\frac{\forall \tau_l\ \tau_r.\ \tau_l[\alpha/\tau] <: \tau_r \in \Delta' \iff M, \Delta \vdash \alpha \dagger \tau_l <: \tau_r}{M, \Delta \vdash \alpha/\tau \dagger \Delta'}$$

Definition 8.33. $\boxed{M, \Delta, \mathrm{A} \vdash \vec{\delta}}$

$$\frac{}{M, \Delta, \epsilon \vdash \epsilon} \qquad \frac{\alpha \notin M \qquad \forall \tau.\ \tau <: \alpha \notin \Delta \qquad M, \Delta, A \vdash \vec{\delta}}{M, \Delta, A\ \alpha \vdash \vec{\delta}}$$

$$\frac{\alpha \notin M \qquad \exists \tau.\ \tau <: \alpha \in \Delta \qquad M, \Delta, \mathrm{A} \vdash \vec{\delta}}{M, \Delta, \mathrm{A}\ \alpha \vdash \vec{\delta}\ \alpha/\,|\,(\overline{\tau}^{\tau <: \alpha \in \Delta})}$$

Definition 8.34. $\boxed{\vec{\alpha}, \vec{\alpha} \vdash \vec{\pi} \cong \vec{\tau}}$ Constrained Implication Congruence

$$\frac{}{\vec{\alpha}_f, \vec{\alpha}_m \vdash \epsilon \cong \epsilon} \qquad \frac{\begin{array}{c}\vec{\alpha}_f, \vec{\alpha}_m \vdash \vec{\pi} \cong \vec{\tau} \qquad \mathrm{FTV}(\tau_l) = \vec{\alpha}_l \qquad \mathrm{FTV}(\tau_r) = \vec{\alpha}_r \\ \Delta \vdash \vec{\alpha}_f\ \vec{\alpha}_m\ \vec{\alpha}_l\ \vec{\alpha}_r \pitchfork \Delta' \qquad \vec{\alpha}_f, \vec{\alpha}_m, \Delta' \vdash \tau_l \texttt{->} \tau_r \cong^+ \tau\end{array}}{\vec{\alpha}_f, \vec{\alpha}_m \vdash \vec{\pi}\ \langle M, \Delta, \tau_l \texttt{->} \tau_r \rangle \cong \vec{\tau}\ \tau}$$

Definition 8.35. $\boxed{\vec{\alpha} \vdash \nu\alpha.Z.\alpha \longrightarrow \alpha \fallingdotseq \mu\alpha.\vec{\tau}}$ Fixpoint Duality

TODO: Note the reason for excluding rigids and closeds from quantification is a way to improve precision, but not necessary for soundness. (Right?). Need to conjure up an example to support this idea.

$$\frac{}{\vec{\alpha}_f \vdash \alpha_{h^+} \cdot \epsilon \cdot \alpha_l \texttt{->} \alpha_r \fallingdotseq \alpha_{h^-} \cdot \epsilon} \qquad \frac{\begin{array}{c}\vec{\alpha}_f \vdash \alpha_{h^+} \cdot Z \cdot \alpha_l \texttt{->} \alpha_r \fallingdotseq \alpha_{h^-} \cdot \mathrm{T} \\ \Delta \vdash \alpha_{h^+} <: \mathrm{T}_h \qquad \Delta \vdash \alpha_l <: \mathrm{T}_l \qquad \Delta \vdash \mathrm{T}_r <: \alpha_r \\ \overline{\tau_l * \tau_r <: \alpha_{h^-}}^{\tau_l \texttt{->} \tau_r \in \mathrm{T}_h} = \Delta_h \\ \vec{\alpha}_f \sqcup M \sqcup \mathrm{FTV}(\mathrm{T}_l) \sqcup \mathrm{FTV}(\mathrm{T}_r)\ \alpha_{h^-} = \mathrm{A} \qquad \Delta \vdash \mathrm{A} \pitchfork \Delta_i \\ \vec{\alpha}_f\ \alpha_{h^-}, M, \Delta_i \sqcup \Delta_h \vdash \&(\mathrm{T}_l) * |(\mathrm{T}_r) \cong^- \tau\end{array}}{\vec{\alpha}_f \vdash \alpha_{h^+} \cdot Z\ \langle M, \Delta \rangle \cdot \alpha_l \texttt{->} \alpha_r \fallingdotseq \alpha_{h^-} \cdot \mathrm{T}\ \tau}$$

Definition 8.36. $\boxed{\Delta \vdash \mathrm{A} \pitchfork \Delta}$ Influential Filter

$$\frac{}{\epsilon \vdash \mathrm{A} \pitchfork \epsilon} \qquad \frac{\alpha \in \mathrm{A} \qquad \alpha \in \mathrm{FTV}(\tau_l) \sqcup \mathrm{FTV}(\tau_r) \qquad \Delta \vdash \mathrm{A} \pitchfork \Delta'}{\Delta\ \tau_l <: \tau_r \vdash N \pitchfork \Delta'\ \tau_l <: \tau_r}$$

$$\frac{\forall \alpha.\ \alpha \in \mathrm{A} \implies \alpha \notin \mathrm{FTV}(\tau_l) \sqcup \mathrm{FTV}(\tau_r) \qquad \Delta \vdash \mathrm{A} \pitchfork \Delta'}{\Delta\ \tau_l <: \tau_r \vdash \mathrm{A} \pitchfork \Delta'}$$

Definition 8.37. $\boxed{\Delta \vdash \mathrm{T} <: \alpha}$

$$\frac{}{\epsilon \vdash \epsilon <: \alpha} \qquad \frac{\Delta \vdash \mathrm{T} <: \alpha}{\Delta\ \tau <: \alpha \vdash \mathrm{T}\ \tau <: \alpha} \qquad \frac{\tau_r \neq \alpha \qquad \Delta \vdash \mathrm{T} <: \alpha}{\Delta\ \tau_l <: \tau_r \vdash \mathrm{T} <: \alpha}$$

Definition 8.38. $\boxed{\Delta \vdash \alpha <: \mathrm{T}}$

$$\frac{}{\epsilon \vdash \alpha <: \epsilon} \qquad \frac{\Delta \vdash \alpha <: \mathrm{T}}{\Delta\ \alpha <: \tau \vdash \alpha <: \mathrm{T}\ \tau} \qquad \frac{\tau_l \neq \alpha \qquad \Delta \vdash \alpha <: \mathrm{T}}{\Delta\ \tau_l <: \tau_r \vdash \alpha <: \mathrm{T}}$$

Definition 8.39. $\boxed{\text{outer}(+|-) = \texttt{ALL}|\texttt{EXI}}$

$$\begin{array}{rcll} \text{outer}(+) & = & \texttt{EXI} & \triangleright\ \textit{positive} \\ \text{outer}(-) & = & \texttt{ALL} & \triangleright\ \textit{negative} \end{array}$$

Definition 8.40. $\boxed{\text{inner}(+|-) = \texttt{ALL}|\texttt{EXI}}$

$$\begin{array}{rcll} \text{inner}(+) & = & \texttt{ALL} & \triangleright\ \textit{positive} \\ \text{inner}(-) & = & \texttt{EXI} & \triangleright\ \textit{negative} \end{array}$$

Definition 8.41. $\boxed{\text{quantify}^{+|-}(\mathrm{A}, \Delta, \mathrm{A}, \Delta, \tau) = \tau}$

$$\begin{array}{rcl} \text{quantify}^{+|-}(\epsilon, \epsilon, \epsilon, \epsilon, \tau) & = & \tau \\ \text{quantify}^{+|-}(\epsilon, \epsilon, \mathrm{A}_i, \Delta_i, \tau) & = & \text{inner}(+|-)\,[\,\mathrm{A}_i\Delta_i\,]\,\tau \\ \text{quantify}^{+|-}(\mathrm{A}_o, \Delta_o, \epsilon, \epsilon, \tau) & = & \text{outer}(+|-)\,[\,\mathrm{A}_o\Delta_o\,]\,\tau \\ \text{quantify}^{+|-}(\mathrm{A}_o, \Delta_o, \mathrm{A}_i, \Delta_i, \tau) & = & \text{outer}(+|-)\,[\,\mathrm{A}_o\Delta_o\,]\,\text{inner}(+|-)\,[\,\mathrm{A}_i\Delta_i\,]\,\tau \end{array}$$

Definition 8.42. $\boxed{\mathrm{A}, \mathrm{A}, \Delta \vdash \tau \cong^{+|-} \tau}$

$$\frac{\begin{array}{c} \mathrm{A}_z, \Delta \vdash \Delta_o \wr \Delta_i \\ (\text{FTV}(\Delta)\ \text{FTV}(\tau)) \diamond \mathrm{A}_z = \mathrm{A}_o \qquad (\text{FTV}(\Delta_i)\ \text{FTV}(\tau)) \setminus \mathrm{A}_z \setminus \mathrm{A}_r = \mathrm{A}_i \\ \text{quantify}^{+|-}(\mathrm{A}_o, \Delta_o, \mathrm{A}_i, \Delta_i, \tau) = \tau' \end{array}}{\mathrm{A}_r, \mathrm{A}_z, \Delta \vdash \tau \cong^{+|-} \tau'}$$

Definition 8.43. $\boxed{\mathrm{A}, \Delta \vdash \Delta \wr \Delta}$

$$\frac{}{\mathrm{A}_z, \epsilon \vdash \epsilon \wr \epsilon} \qquad \frac{\text{FTV}(\tau_l)\ \text{FTV}(\tau_r) = \mathrm{A}_q \qquad \forall \alpha.\ \alpha \in \mathrm{A}_q \implies \alpha \in \mathrm{A}_z \qquad \mathrm{A}_z, \Delta \vdash \Delta_o \wr \Delta_i}{\mathrm{A}_z, \Delta\ \tau_l <: \tau_r \vdash \Delta_o\ \tau_l <: \tau_r \wr \Delta_i}$$

$$\frac{\text{FTV}(\tau_l)\ \text{FTV}(\tau_r) = \mathrm{A}_q \qquad \alpha \in \mathrm{A}_q \qquad \alpha \notin \mathrm{A}_z \qquad \mathrm{A}_z, \Delta \vdash \Delta_o \wr \Delta_i}{\mathrm{A}_z, \Delta\ \tau_l <: \tau_r \vdash \Delta_o \wr \Delta_i\ \tau_l <: \tau_r}$$

Definition 8.44. $\boxed{\Vdash e}$

$$\frac{e = v}{\Vdash e} \qquad\qquad \frac{e \rightsquigarrow e' \qquad \Vdash e'}{\Vdash e}$$

Theorem 8.1. (Typing Soundness)

$$\frac{\vdash e : \tau \dashv Z}{\Vdash e}$$

Proof:
assume $\vdash e : \tau \dashv Z$
. **let** $\vec{\delta}\ \Gamma'\ \tau'$ **s.t.** $\vec{\delta}, \Gamma' \vDash e : \tau'$ by Lemma 8.3
. $\vec{\delta}, \vec{\sigma} \vDash \Gamma'$ by ...
. $\Vdash e[\vec{\sigma}]$ by theorem 8.51
. $e[\vec{\sigma}] = e$ by ...
. $\Vdash e$ by substitution
□

Theorem 8.2. (Proof typing consistency)

$$\frac{\Gamma \vdash e : \tau \dashv Z}{\exists \vec{\delta}.\ \vec{\delta} \vDash Z}$$

TODO: ...

Theorem 8.3. (Proof typing soundness)

$$\frac{\Gamma \vdash e : \tau \dashv Z}{\exists \vec{\delta}.\ \vec{\delta}, \Gamma \vDash e : \tau}$$

TODO: ...

Theorem 8.4. (Proof typing weak soundness)

$$\frac{\Gamma \vdash e : \tau \dashv Z}{\forall \vec{\delta}.\ \vec{\delta} \vDash Z \implies \vec{\delta}, \Gamma \vDash e : \tau}$$

TODO: rewrite inductive hypotheses with just the conclusion implied by the case conditions
TODO: rewrite cases with universal/implication in conclusion/hypotheses

Proof:

assume $\Gamma \vdash e : \tau \dashv Z$
. **induct on** $\Gamma \vdash e : \tau \dashv Z$
. **case** $e = \texttt{@} \quad \tau = \texttt{@}$
. . **let** $\vec{\delta}$ by definition
. . $\vec{\delta}, \Gamma \vDash \texttt{@} : \texttt{@}$ by definition
. . $\vec{\delta}, \Gamma \vDash e : \tau$ by substitution
. . $\exists \vec{\delta}.\ \vec{\delta}, \Gamma \vDash e : \tau$ by witness
. **case** $e = x \quad x : \tau \in \Gamma$
. **wrt** x
. . **let** $\vec{\delta}$ by definition
. . $\vec{\delta}, \Gamma \vDash x : \tau$ by definition
. . $\vec{\delta}, \Gamma \vDash e : \tau$ by substitution
. . $\exists \vec{\delta}.\ \vec{\delta}, \Gamma \vDash e : \tau$ by witness
. **case** $\Gamma \vdash e' : \tau' \dashv Z \quad \tau = \texttt{<}l\texttt{>}\tau' \quad e = \texttt{<}l\texttt{>}e'$
. **hypo** $\Gamma \vdash e' : \tau' \dashv Z \implies \vec{\delta}, \Gamma \vDash e' : \tau'$
. **wrt** $e'\ \tau'$
. . **let** $\vec{\delta}$ by definition
. . $\vec{\delta}, \Gamma \vDash e' : \tau'$ by application
. . $\vec{\delta}, \Gamma \vDash \texttt{<}l\texttt{>}e' : \texttt{<}l\texttt{>}\tau'$ by definition
. . $\vec{\delta}, \Gamma \vDash e : \tau$ by substitution
. . $\exists \vec{\delta}.\ \vec{\delta}, \Gamma \vDash e : \tau$ by witness
. **TODO: remaining trivial introduction cases**
. **case**
. **hypo**
. **wrt**
. **case** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \quad Z_0 \looparrowright Z_1 \quad \tau_0 <: l \texttt{->} \alpha \dashv Z_1 \quad e = e_0 \,.\, l \quad \tau = \alpha \quad Z = Z_1$
. **hypo** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \implies \vec{\delta}, \Gamma \vDash e_0 : \tau_0$

. **wrt** $\vec{\delta}\ e'\ l\ \alpha\ \tau_0\ Z_0\ Z_1$
. . $\vec{\delta}, \Gamma \vDash e_0 : \tau_0$ by application
. . **let** $M\ \Delta$ **s.t.** τ_0 <: l->$\alpha \dashv M, \Delta$ by theorem 8.18
. . $\vec{\delta} \vDash \tau_0$ <: l->α by theorem 8.23
. . $\vec{\delta}, \Gamma \vDash e_0 : l$->$\alpha$ by theorem 8.19
. . $\vec{\delta}, \Gamma \vDash e_0.l : \alpha$ by theorem 8.20
. . $\vec{\delta}, \Gamma \vDash e : \tau$ by substitution
. . $\exists \vec{\delta}.\ \vec{\delta}, \Gamma \vDash e : \tau$ by witness
. **case** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \quad Z_0 \looparrowright Z_1 \quad \Gamma \vdash e_1 : \tau_1 \dashv Z_1 \quad e = e_0(e_1) \quad \tau = \alpha \quad Z = Z_2$
. $Z_1 \looparrowright Z_2 \quad \tau_0$ <: τ_1->$\alpha \dashv Z_2$
. **hypo** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \implies \vec{\delta}, \Gamma \vDash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \dashv Z_1 \implies \vec{\delta}, \Gamma \vDash e_1 : \tau_1$
. **wrt** $e_0\ e_1\ \alpha\ \tau_0\ \tau_1\ Z_0\ Z_1\ Z_2$
. . $\vec{\delta}, \Gamma \vDash e_0 : \tau_0$ by application
. . $\vec{\delta}, \Gamma \vDash e_1 : \tau_1$ by application
. . **let** $M\ \Delta$ **s.t.** τ_0 <: τ_1->$\alpha \dashv \langle M, \Delta \rangle$ by theorem 8.18
. . $\vec{\delta}, \Gamma \vDash \tau_0$ <: τ_1->α by theorem 8.23
. . $\vec{\delta} \vDash e_0 : \tau_1$->$\alpha$ by theorem 8.19
. . $\vec{\delta}, \Gamma \vDash e_0(e_1) : \alpha$ by theorem 8.39
. . $\vec{\delta}, \Gamma \vDash e : \tau$ by substitution
. . $\exists \vec{\delta}.\ \vec{\delta}, \Gamma \vDash e : \tau$ by substitution
. **case** e = `loop(`e'`)`
. | τ = ALL[α_l']α_l'->EXI[α_r'.α_l'*α_r'<:LFLFP[α_{h^-}] |(T)]α_r
. | $Z = Z_0$
. | $\Gamma \vdash e' : \alpha_{h^+}$->$\tau' \dashv Z_0 \quad Z_0 \looparrowright Z_1 \quad \tau'$ <: α_l->$\alpha_r \dashv Z_1$
. | $\mathrm{FTV}(\Gamma) \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l$->$\alpha_r \risingdotseq \alpha_{h^-} \cdot \mathrm{T}$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash Z_0 \implies \vec{\delta}, \Gamma \vDash e' : \alpha_{h^+}$->$\tau'$
. **wrt** $e'\ \tau'\ \alpha_{h^+}\ \alpha_l\ \alpha_r\ \alpha_{h^-}\ \alpha_l'\ \alpha_r'\ \mathrm{T}\ Z_0\ Z_1$
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash Z$
. . . $\vec{\delta} \vDash Z_0$ by substitution
. . . $\vec{\delta}, \Gamma \vDash e' : \alpha_{h^+}$->$\tau'$ by instantiation and application
. . . $\vec{\delta} \vDash \tau'$ <: ALL[α_l']α_l'->EXI[α_r'.α_l'*α_r'<:LFP[α_{h^-}] |(T)]α_r by theorem 8.5
. . . $\vec{\delta} \vDash \tau'$ <: τ by substitution
. . . $\vec{\delta} \vDash e' : \alpha_{h^+}$->$\tau$ by theorem 8.19
. . . $\vec{\delta} \vDash$ `loop(`e'`)` : τ by theorem 8.16
. . . $\vec{\delta} \vDash e : \tau$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash Z \implies \vec{\delta} \vDash e : \tau$ by implication and generalization
. $\vec{\delta}, \Gamma \vDash e : \tau$ by induction

□

Theorem 8.5. (Fixpoint duality soundness (new))

$$\frac{\tau \text{ <: } \alpha_l\text{->}\alpha_r \dashv Z_1 \qquad \mathrm{FTV}(\tau) \subseteq N \qquad \alpha_l \notin N \qquad \alpha_r \notin N \qquad N \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l\text{->}\alpha_r \risingdotseq \alpha_{h^-} \cdot \mathrm{T}}{\vec{\delta} \vDash \tau \text{ <: } \mathtt{ALL[}\alpha_l'\mathtt{]}\alpha_l'\mathtt{\text{->}EXI[}\alpha_r'\mathtt{.}\alpha_l'\mathtt{*}\alpha_r'\mathtt{<:LFP[}\alpha_{h^-}\mathtt{]\ |(T)]}\alpha_r'}$$

TODO: ...

TODO: Cretin's corresponding theorem is Theorem 101 on p. 134

TODO: See how Cretin proves this without using subject reduction

Theorem 8.6. (Fixpoint duality soundness old)

$$\frac{Z_0 \looparrowright Z_1 \qquad N \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l\texttt{->}\alpha_r \doteqdot \alpha_{h^-} \cdot \mathrm{T}}{\forall \tau.\ \tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l'\texttt{*}\alpha_r'\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\texttt{)]}\alpha_r' \dashv Z_0}$$

Proof:

assume $Z_0 \looparrowright Z_1 \quad N \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l\texttt{->}\alpha_r \doteqdot \alpha_{h^-} \cdot \mathrm{T}$

. **induct on** $N \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l\texttt{->}\alpha_r \doteqdot \alpha_{h^-} \cdot \mathrm{T}$

. **case** $Z_1 = \epsilon \quad \mathrm{T} = \epsilon$

. . **for** τ

. . . **assume** $\tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1$

. . . . $\tau <: \alpha_l\texttt{->}\alpha_r \dashv \epsilon$ by substitution

. . . . **let** $M\ \Delta$ **s.t.** $\langle M, \Delta\rangle \in \epsilon$ by theorem 8.14

. . . . $\bot$ by theorem 8.15

. . . $\tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l'\texttt{*}\alpha_r'\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\texttt{)]}\alpha_r' \dashv Z_0$ by implication

. . $\forall \tau.\ \tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l'\texttt{*}\alpha_r'\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\texttt{)]}\alpha_r' \dashv Z_0$ by generalization

. **case** $Z_1 = Z\ \langle M, \Delta\rangle \qquad \mathrm{T} = \mathrm{T}_i\ \tau_i$

. | $N \vdash \alpha_{h^+} \cdot Z \cdot \alpha_l\texttt{->}\alpha_r \doteqdot \alpha_{h^-} \cdot \mathrm{T}_i$

. | $M, \Delta, \Delta \vdash \alpha_{h^+} <: \mathrm{T}_h \qquad M, \Delta, \Delta \vdash \alpha_l <: \mathrm{T}_l \qquad M, \Delta, \Delta \vdash \mathrm{T}_r <: \alpha_r$

. | $\overline{\tau_l\texttt{*}\tau_r\texttt{<:}\alpha_{h^-}}^{\tau_l\ \tau_r.\ \tau_l\texttt{->}\tau_r \in \mathrm{T}_h} = \Delta_h$

. | $N \sqcup M \sqcup \mathrm{FTV}(\mathrm{T}_l) \sqcup \mathrm{FTV}(\mathrm{T}_r)\ \alpha_{h^-} = \mathrm{A} \qquad \Delta \vdash \mathrm{A} \pitchfork \Delta_i$

. | $N\ \alpha_{h^-}, M, \Delta_i \sqcup \Delta_h \vdash \texttt{\&(}\mathrm{T}_l\texttt{)*|(}\mathrm{T}_r\texttt{)} \cong^- \tau_i$

. **hypo** $N \vdash \alpha_{h^+} \cdot Z \cdot \alpha_l\texttt{->}\alpha_r \doteqdot \alpha_{h^-} \cdot \mathrm{T}_i \implies$
$\forall \tau.\ \tau <: \alpha_l\texttt{->}\alpha_r \dashv Z \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}_i\texttt{)]}\alpha_r' \dashv Z_0$

. **wrt** $Z\ M\ \Delta\ \mathrm{T}_i\ \tau_i$

. . **for** τ **assume** $\tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1$

. . . $\tau <: \alpha_l\texttt{->}\alpha_r \dashv Z\ \langle M, \Delta\rangle$ by substitution

. . . $\tau <: \alpha_l\texttt{->}\alpha_r \dashv \langle M, \Delta\rangle$ by theorem 8.17

. . . $\tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}_i\texttt{)]}\alpha_r' \dashv Z_0$ by instantiation and application

. . . $\tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}_i\ \ \tau_i\texttt{)]}\alpha_r' \dashv Z_0$ by theorem 8.7

. . $\forall \tau.\ \tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\texttt{)]}\alpha_r' \dashv Z_0$ by implication and generalization

. $\forall \tau.\ \tau <: \alpha_l\texttt{->}\alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\texttt{)]}\alpha_r' \dashv Z_0$ by induction

□ by implication

Theorem 8.7. (Universe proof typing fixpoint extension)

$$\frac{\begin{array}{c}\vec{\delta} \vDash \tau <: \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.(}\alpha_l\texttt{,}\alpha_r\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}_i\texttt{)]}\alpha_r \\ \tau <: \alpha_l\texttt{->}\alpha_r \dashv \langle M, \Delta\rangle \\ \Delta \vdash \alpha_{h^+} <: \mathrm{T}_h \qquad \Delta \vdash \alpha_l <: \mathrm{T}_l \qquad \Delta \vdash \mathrm{T}_r <: \alpha_r \\ \overline{\tau_l\texttt{*}\tau_r\texttt{<:}\alpha_{h^-}}^{\tau_l\texttt{->}\tau_r \in \mathrm{T}_h} = \Delta_h \qquad N \sqcup M \sqcup \mathrm{FTV}(\mathrm{T}_l) \sqcup \mathrm{FTV}(\mathrm{T}_r)\ \alpha_{h^-} = \mathrm{A} \\ \Delta \vdash \mathrm{A} \pitchfork \Delta_i \qquad N\ \alpha_{h^-}, M, \Delta_i \sqcup \Delta_h \vdash \texttt{\&(}\mathrm{T}_l\texttt{)*|(}\mathrm{T}_r\texttt{)} \cong^- \tau_i\end{array}}{\vec{\delta} \vDash \tau <: \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.}\alpha_l\texttt{*}\alpha_r\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}_i\ \ \tau_i\texttt{)]}\alpha_r}$$

Proof:
TODO: ...
□

Theorem 8.8. (Universe proof typing case soundness)

$$\frac{\begin{array}{c}\tau \mathrel{<:} \alpha_l\texttt{->}\alpha_r \dashv \langle M, \Delta\rangle \\ \Delta \vdash \alpha_{h^+} \mathrel{<:} \mathrm{T}_h \qquad \Delta \vdash \alpha_l \mathrel{<:} \mathrm{T}_l \qquad \Delta \vdash \mathrm{T}_r \mathrel{<:} \alpha_r \\ \overline{\tau_l\texttt{*}\tau_r\texttt{<:}\alpha_{h^-}}^{\tau_l\texttt{->}\tau_r \in \mathrm{T}_h} = \Delta_h \qquad N \sqcup M \sqcup \mathrm{FTV}(\mathrm{T}_l) \sqcup \mathrm{FTV}(\mathrm{T}_r)\ \alpha_{h^-} = \mathrm{A} \\ \Delta \vdash \mathrm{A} \pitchfork \Delta_i \qquad N\ \alpha_{h^-}, M, \Delta_i \sqcup \Delta_h \vdash \texttt{\&}(\mathrm{T}_l)\texttt{*|}(\mathrm{T}_r) \cong^- \tau_i\end{array}}{\vec{\delta} \vDash \tau \mathrel{<:} \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.}\alpha_l\texttt{*}\alpha_r\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \tau_i\texttt{]}\alpha_r}$$

Proof:
TODO: ...
□

Theorem 8.9. (Universe proof typing fixpoint union)

$$\frac{\begin{array}{c}\vec{\delta} \vDash \tau \mathrel{<:} \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.}\alpha_l\texttt{*}\alpha_r\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\texttt{)]}\alpha_r \\ \vec{\delta} \vDash \tau \mathrel{<:} \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.}\alpha_l\texttt{*}\alpha_r\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \tau\texttt{]}\alpha_r\end{array}}{\vec{\delta} \vDash \tau \mathrel{<:} \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.}\alpha_l\texttt{*}\alpha_r\texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \ \texttt{|(}\mathrm{T}\ \ \tau\texttt{)]}\alpha_r}$$

Proof:
TODO: ...
□

Theorem 8.10. Influential soundness
TODO: Prove that any constraints on non-influential variables with have been transitively applied to the influential variables

$$\frac{\begin{array}{c}\tau \mathrel{<:} \alpha_l\texttt{->}\alpha_r \dashv M, \Delta \qquad \Delta \vdash \alpha_l \mathrel{<:} \mathrm{T}_l \qquad \Delta \vdash \mathrm{T}_r \mathrel{<:} \alpha_r \\ \Delta \vdash \mathrm{A} \pitchfork \Delta_i \qquad \mathrm{FTV}(\tau) \subseteq \mathrm{A} \\ N, M, \Delta \sqcup \Delta' \dashv \texttt{\&}(\mathrm{T}_l)\texttt{*|}(\mathrm{T}_r) \cong^- \tau \qquad N, M, \Delta_i \sqcup \Delta' \dashv \texttt{\&}(\mathrm{T}_l)\texttt{*|}(\mathrm{T}_r) \cong^- \tau_i\end{array}}{\vec{\delta} \vDash \tau \mathrel{<:} \tau_i \land \vec{\delta} \vDash \tau_i \mathrel{<:} \tau}$$

Proof:
TODO: ...
□

Theorem 8.11. (Universe proof typing implication expansion)

$$\frac{}{\tau_l\texttt{->}\tau_r \mathrel{<:} \texttt{ALL[}\alpha_l\texttt{]}\alpha_l\texttt{->EXI[}\alpha_r\texttt{.(}\alpha_l\texttt{,}\alpha_r\texttt{)<:(}\tau_l\texttt{,}\tau_r\texttt{)]}\alpha_r \dashv M, \Delta}$$

Proof:
TODO: $P \implies Q$ is equivalent to $\neg(P \land \neg Q)$
TODO: $X \to Y$ is equivalent to $\forall x \in X.\ \exists y.\ (x, y) \in (X \times Y)$
TODO: $X \to Y$ is equivalent to $\neg(\exists x \in X \land \nexists y \in Y)$
□

Theorem 8.12. (Upper bound interpretation sound)

$$\frac{M, \Delta, \Delta \vdash \alpha \mathrel{<:} \mathrm{T}}{\alpha \mathrel{<:} \texttt{\&}(\mathrm{T}) \dashv \langle M, \Delta\rangle}$$

Proof:
TODO: ...
□

Theorem 8.13. (Lower bound interpretation sound)

$$\frac{M, \Delta, \Delta \vdash \mathrm{T} <: \alpha}{|(\mathrm{T}) <: \alpha \dashv \langle M, \Delta \rangle}$$

Proof:
TODO: ...
□

Theorem 8.14. (Universe proof typing worldly)

$$\frac{\tau_l <: \tau_r \dashv Z}{\exists M\ \Delta\ .\ \langle M, \Delta \rangle \in Z}$$

Proof:
TODO: ...
□

Theorem 8.15. (Empty containment absurd)

$$\frac{\langle M, \Delta \rangle \in \epsilon}{\bot}$$

Proof:
TODO: ...
□

Theorem 8.16. (Model typing implication independence)

$$\frac{\vec{\delta}, \Gamma \vDash e : \tau_l \texttt{->} \tau_r \qquad \vec{\delta} \vDash \tau_r}{\vec{\delta}, \Gamma \vDash \texttt{loop(}e\texttt{)} : \tau_r}$$

Proof:
TODO: ...
□

Theorem 8.17. (Proof subtyping decomposition)

$$\frac{\tau_l <: \tau_r \dashv Z \quad \langle M, \Delta \rangle}{\tau_l <: \tau_r \dashv \langle M, \Delta \rangle}$$

Proof:
TODO: ...
□

Theorem 8.18. (Proof subtyping choice)

$$\frac{\tau_l <: \tau_r \dashv Z}{\exists M\ \Delta.\ \tau_l <: \tau_r \dashv M, \Delta}$$

Proof:
TODO: ...
□

Theorem 8.19. (Model subtyping elimination)

$$\frac{\vec{\delta} \vDash \tau_l <: \tau_r \qquad \vec{\delta}, \Gamma \vDash e : \tau_l}{\vec{\delta}, \Gamma \vDash e : \tau_r}$$

Proof:
assume $\vec{\delta} \vDash \tau_l <: \tau_r \quad \vec{\delta}, \Gamma \vDash e : \tau_l$
. **invert on** $\vec{\delta} \vDash \tau_l <: \tau_r$
. **case** $\forall e'\ \Gamma'.\ \vec{\delta}, \Gamma' \vDash e' : \tau_l \implies \vec{\delta}, \Gamma' \vDash e' : \tau_r$
. . $\vec{\delta}, \Gamma \vDash e : \tau_l \implies \vec{\delta}, \Gamma \vDash e : \tau_r$ by instantiation
. . $\vec{\delta}, \Gamma \vDash e : \tau_r$ by application
. $\vec{\delta}, \Gamma \vDash e : \tau_r$ by inversion
□ by implication

Theorem 8.20. (Model typing record elimination)

$$\frac{\vec{\delta}, \Gamma \vDash e : l\text{->}\tau}{\vec{\delta}, \Gamma \vDash e.l : \tau}$$

Proof:
assume $\vec{\delta}, \Gamma \vDash e : l\text{->}\tau$
. **induct on** $\vec{\delta}, \Gamma \vDash e : l\text{->}\tau$
. **case** $e = G \quad \$l\text{=>}v \in G \quad \vec{\delta}, \Gamma \vDash v : \tau \quad \forall v'.\ \$l\text{=>}v' \in G \implies v' = v$
. **wrt** $G\ v$
. . $\vec{\delta}, \Gamma \vDash G : l\text{->}\tau$ by substitution
. . $G.l \rightsquigarrow v$ by definition
. . **let** $\vec{\sigma}$ **s.t.** $\vec{\delta}, \Gamma \vDash \vec{\sigma}$ by theorem ??
. . $G.l[\vec{\sigma}] \rightsquigarrow v$ by definition
. . $v = v[\vec{\sigma} \sqcup \epsilon]$ by definition
. . $G.l[\vec{\sigma}] \rightsquigarrow v[\vec{\sigma} \sqcup \epsilon]$ by substitution
. . $\vec{\delta}, \epsilon \vDash \epsilon$ by definition
. . $\vec{\delta}, \Gamma \sqcup \epsilon \vDash v : \tau$ by definition
. . $\vec{\delta}, \Gamma \vDash G.l : \tau$ by definition
. . $\vec{\delta}, \Gamma \vDash e.l : \tau$ by substitution
. **case** $\vec{\delta}, \vec{\sigma} \vDash \Gamma \quad e[\vec{\sigma}] \rightsquigarrow e'[\vec{\sigma} \sqcup \vec{\sigma}'] \quad \vec{\delta}, \vec{\sigma}' \vDash \Gamma' \quad \vec{\delta}, \Gamma \sqcup \Gamma' \vDash e' : l\text{->}\tau$
. **hypo** $\vec{\delta}, \Gamma \sqcup \Gamma' \vDash e' : l\text{->}\tau \implies \vec{\delta}, \Gamma \sqcup \Gamma' \vDash e'.l : \tau$
. **wrt** $\vec{\sigma}\ e'\ \vec{\sigma}'\ \Gamma'$
. . $\vec{\delta}, \Gamma \sqcup \Gamma' \vDash e'.l : \tau$ by application
. . $e[\vec{\sigma}].l \rightsquigarrow e'[\vec{\sigma} \sqcup \vec{\sigma}'].l$ by definition
. . $e[\vec{\sigma}].l = e.l[\vec{\sigma}]$ by definition
. . $e'[\vec{\sigma} \sqcup \vec{\sigma}'].l = e'.l[\vec{\sigma} \sqcup \vec{\sigma}']$ by definition
. . $e.l[\vec{\sigma}] \rightsquigarrow e'.l[\vec{\sigma} \sqcup \vec{\sigma}']$ by substitution
. . $\vec{\delta}, \Gamma \vDash e.l : \tau$ by definition
. $\vec{\delta}, \Gamma \vDash e.l : \tau$ by induction
□

Theorem 8.21. (Proof subtyping consistency)

$$\frac{\tau_l <: \tau_r \dashv M, \Delta}{\exists \vec{\delta}.\ \vec{\delta} \vDash \Delta}$$

Theorem 8.22. (Proof subtyping soundness)

$$\frac{\tau_l <: \tau_r \dashv M, \Delta}{\exists \vec{\delta}.\ \vec{\delta} \vDash \tau_l <: \tau_r}$$

Theorem 8.23. (Proof subtyping weak soundness)

$$\frac{\tau_l <: \tau_r \dashv M, \Delta}{\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r}$$

TODO: closed variabless simply remove variables from soundness consideration Proof:
TODO: think about how to handle the mutually recursive definition

assume $\tau_l <: \tau_r \dashv M, \Delta$
. **induct on** $\tau_l <: \tau_r \dashv M, \Delta$
. **case** $\tau_l = \tau \quad \tau_r = \tau$
. **wrt** τ
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \tau <: \tau$ by theorem 8.37
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_l = \mathtt{BOT}$
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \mathtt{BOT} <: \tau_r$ by definition
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_r = \mathtt{TOP}$
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \tau_l <: \mathtt{TOP}$ by definition
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_r = l\texttt{->}(\tau_{rl}\&\tau_{rr}) \quad \tau_l <: (l\texttt{->}\tau_{rl})\&(l\texttt{->}\tau_{rr}) \dashv M, \Delta$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: (l\texttt{->}\tau_{rl})\&(l\texttt{->}\tau_{rr})$
. **wrt** $l\ \tau_{rl}\ \tau_{rr}$
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \tau_l <: (l\texttt{->}\tau_{rl})\&(l\texttt{->}\tau_{rr})$ by application
. . . **for** $e\ \Gamma$ **assume** $\vec{\delta}, \Gamma \vDash e : \tau_l$
. . . . $\vec{\delta}, \Gamma \vDash e : (l\texttt{->}\tau_{rl})\&(l\texttt{->}\tau_{rr})$ by theorem 8.38
. . . . $\vec{\delta}, \Gamma \vDash e : l\texttt{->}\tau_{rl}$ by theorem 8.36
. . . . $\vec{\delta}, \Gamma \vDash e : l\texttt{->}\tau_{rr}$ by theorem 8.36
. . . . $\vec{\delta}, \Gamma \vDash e.l : \tau_{rl}$ by theorem 8.20
. . . . $\vec{\delta}, \Gamma \vDash e.l : \tau_{rr}$ by theorem 8.20
. . . . $\vec{\delta}, \Gamma \vDash e.l : \tau_{rl}\&\tau_{rr}$ by definition
. . . . $\vec{\delta}, \Gamma \vDash e : l\texttt{->}(\tau_{rl}\&\tau_{rr})$ by theorem 8.34

. . . . . $\vec{\delta}, \Gamma \vDash e : \tau_r$ by substitution
. . . $\forall e\ \Gamma.\ \vec{\delta}, \Gamma \vDash e : \tau_l \implies \vec{\delta}, \Gamma \vDash e : \tau_r$ by implication and generalization
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by definition
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_l = \tau_{ll} \mid \tau_{lr} \quad M = M_1 \quad \Delta = \Delta_1$
. | $\tau_{ll} <: \tau_r \dashv M_0, \Delta_0 \quad M_0 \preceq M_1 \quad \Delta_0 \preceq \Delta_1 \quad \tau_{lr} <: \tau_r \dashv M_1, \Delta_1$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_0 \implies \vec{\delta} \vDash \tau_{ll} <: \tau_r \quad \forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_1 \implies \vec{\delta} \vDash \tau_{lr} <: \tau_r$
. **wrt** $\tau_{ll}\ \tau_{lr}$
. . **for** $\vec{\delta}$ **assume** $M, \vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \Delta_1$ by substitution
. . . $\vec{\delta} \vDash \tau_{lr} <: \tau_r$ by application
. . . $\vec{\delta} \vDash \Delta_0$ by theorem ?? **TODO: ...**
. . . $\vec{\delta} \vDash \tau_{ll} <: \tau_r$ by application
. . . $\vec{\delta} \vDash \tau_{ll} \mid \tau_{lr} <: \tau_r$ by definition
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_r = \tau_{rl} \& \tau_{rr} \quad M = M_1 \quad \Delta = \Delta_1$
. | $\tau_l <: \tau_{rl} \dashv M_0, \Delta_0 \quad M_0 \preceq M_1 \quad \Delta_0 \preceq \Delta_1 \quad \tau_l <: \tau_{rr} \dashv M_1, \Delta_1$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_0 \implies \vec{\delta} \vDash \tau_l <: \tau_{rl} \quad \forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_1 \implies \vec{\delta} \vDash \tau_l <: \tau_{rr}$
. **wrt**
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \Delta_1$ by substitution
. . . $\vec{\delta} \vDash \tau_l <: \tau_{rr}$ by instantiation and application
. . . $\vec{\delta} \vDash \Delta_0$ by theorem ?? **TODO: ...**
. . . $\vec{\delta} \vDash \tau_l <: \tau_{rl}$ by instantiation and application
. . . $\vec{\delta} \vDash \tau_l <: \tau_{rl} \& \tau_{rr}$ by definition
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case** $\tau_l = \mathtt{EXI[A\ \ } Q\mathtt{]}\,\tau_b \quad M = M_1 \quad \Delta = \Delta_1$
. | $Q \dashv M_0, \Delta_0 \quad A \# \tau_r \quad M_0 \sqcup \mathrm{A} \preceq M_1 \quad \Delta_0 \preceq \Delta_1 \quad \tau_b <: \tau_r \dashv M_1, \Delta_1$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_1 \implies \vec{\delta} \vDash M_1, \tau_b <: \tau_r$
. **mutu** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_0 \implies \vec{\delta} \vDash Q$ by theorem ?? **TODO: sequence soundness / mutual dependence**
. **wrt** A $Q\ \tau_b\ M_1\ \Delta_1\ M_0\ \Delta_0$
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \Delta_1$ by substitution
. . . $\vec{\delta} \vDash \tau_b <: \tau_r$ by application

. . . $\vec{\delta} \vDash \Delta_0$ by theorem 8.28
. . . $\vec{\delta} \vDash Q$ by application
. . . **for** e **assume** $\vec{\delta} \vDash e : \tau_l$
. . . . $\vec{\delta} \vDash e : \texttt{EXI[A } Q\texttt{]}\tau_b$ by substitution
. . . . $\vec{\delta} \vDash e : \tau_r$ by theorem 8.24
. . . $\forall e.\ \vec{\delta} \vDash e : \tau_l \implies \vec{\delta} \vDash e : \tau_r$
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by definition
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_r = \texttt{ALL[A } Q\texttt{]}\tau_b \quad M = M_1 \quad \Delta = \Delta_1$
. | $Q \dashv M_0, \Delta_0 \quad A \# \tau_l \quad M_0 \sqcup \mathrm{A} \preceq M_1 \quad \Delta_0 \preceq \Delta_1 \quad \tau_l <: \tau_b \dashv M_1, \Delta_1$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_1 \implies \vec{\delta} \vDash \tau_l <: \tau_b$
. **mutu** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_0 \implies \vec{\delta} \vDash Q$ by theorem ?? **TODO: sequence soundness / mutual dependence**
. **wrt** A Q τ_b M_1 Δ_1 M_0 Δ_0
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \Delta_1$ by substitution
. . . $\vec{\delta} \vDash \tau_l <: \tau_b$ by application
. . . $\vec{\delta} \vDash \Delta_0$ by theorem 8.28
. . . $\vec{\delta} \vDash Q$ by application
. . . **for** e **assume** $\vec{\delta} \vDash e : \tau_l$
. . . . $\vec{\delta} \vDash e : \texttt{ALL[A } Q\texttt{]}\tau_b$ by theorem 8.25
. . . . $\vec{\delta} \vDash e : \tau_r$ by substitution
. . . $\forall e.\ \vec{\delta} \vDash e : \tau_l \implies \vec{\delta} \vDash e : \tau_r$
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by definition
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. . **TODO: ...**
. **case** $\tau_l = \alpha \quad M = M_1 \quad \Delta = \Delta_1\ \alpha <: \tau_r$
. | $\alpha \notin M_0 \quad M_0, \Delta_0 \vdash \Delta_m <:^\sharp \alpha/\tau_r \quad M_0, \Delta_0 \vdash \mathrm{T} <:^\dagger \alpha \quad M_0 \preceq M_1$
. | $\Delta_0 \sqcup \Delta_m \preceq \Delta_1 \quad \texttt{|}(\mathrm{T}) <: \tau_r \dashv M_1, \Delta_1$
. **hypo** $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta_1 \implies \vec{\delta} \vDash \texttt{|}(\mathrm{T}) <: \tau_r$
. **wrt** α M_1 Δ_1 M_0 Δ_0 Δ_m T
. . **for** $\vec{\delta}$ **assume** $\vec{\delta} \vDash \Delta$
. . . $\vec{\delta} \vDash \Delta_1\texttt{(}\alpha\texttt{<:}\tau_r\texttt{)}$ by substitution
. . . $\vec{\delta} \vDash \alpha <: \tau_r$ by theorem 8.26
. . . $\vec{\delta} \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by implication and generalization
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**

. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. $\forall \vec{\delta}.\ \vec{\delta} \vDash \Delta \implies \vec{\delta} \vDash \tau_l <: \tau_r$ by induction
□ by implication

Theorem 8.24. Model typing existential elimination

$$\frac{\vec{\delta} \vDash e : \mathtt{EXI[A\ \ }Q\mathtt{]}\tau_l \qquad \vec{\delta} \vDash Q \qquad \vec{\delta} \vDash \tau_l <: \tau_r \qquad \mathrm{A} \mathbin{\#} \tau_r}{\vec{\delta} \vDash e : \tau_r}$$

Proof:
TODO: depends on proof subtyping. can we abstract away proof subtyping?

Theorem 8.25. Model typing universal introduction

$$\frac{\vec{\delta} \vDash e : \tau_l \qquad \vec{\delta} \vDash Q \qquad \vec{\delta} \vDash \tau_l <: \tau_r \qquad \mathrm{A} \mathbin{\#} \tau_l}{\vec{\delta} \vDash e : \mathtt{ALL[A\ \ }Q\mathtt{]}\tau_r}$$

Proof:
TODO: depends on proof subtyping. can we abstract away proof subtyping?

Theorem 8.26. Model subtyping sequence last

$$\frac{\vec{\delta} \vDash \Delta\ \mathtt{(}\tau_l\mathtt{<:}\tau_r\mathtt{)}}{\vec{\delta} \vDash \tau_l <: \tau_r}$$

Proof:
TODO: ...

Theorem 8.27. Model subtyping sequence reduction

$$\frac{\vec{\delta} \vDash \Delta\ \delta}{\vec{\delta} \vDash \Delta}$$

Proof:
TODO: ...

Theorem 8.28. Model subtyping sequence prefix

$$\frac{\vec{\delta} \vDash \Delta' \qquad \Delta \preceq \Delta'}{\vec{\delta} \vDash \Delta}$$

Proof:
TODO: ...

Theorem 8.29. Model subtyping sequence uncat

$$\frac{\vec{\delta} \vDash \Delta \sqcup \Delta'}{\vec{\delta} \vDash \Delta}$$

Proof:
TODO: ...

Theorem 8.30. concatenation prefix

$$\frac{}{\Delta \preceq \Delta \sqcup \Delta'}$$

Proof:
TODO: ...

Theorem 8.31. Model subtyping unsub left

$$\frac{\vec{\delta} \vDash \tau_l <: \tau_r \qquad \alpha/\tau_l \in \vec{\delta}}{\vec{\delta} \vDash \alpha <: \tau_r}$$

Proof:
TODO: ...

Theorem 8.32. Model subtyping unsub right

$$\frac{\vec{\delta} \vDash \tau_l <: \tau_r \qquad \alpha/\tau_r \in \vec{\delta}}{\vec{\delta} \vDash \tau_l <: \alpha}$$

Proof:
TODO: ...

Theorem 8.33. Model subtyping something

$$\frac{\vec{\delta} \vDash \Delta \qquad M, \Delta \vdash \mathrm{T} <:^{\sharp} \alpha}{\alpha/ \,|\, (\mathrm{T}) \in \vec{\delta}}$$

Proof:

TODO: ...

Theorem 8.34. Model typing record introduction

$$\frac{\vec{\delta} \vDash e.l : \tau}{\vec{\delta} \vDash e : l\text{->}\tau}$$

Proof:
TODO: ...

Theorem 8.35. Model typing implication introduction **TODO: this is really messed up**

$$\frac{\vec{\delta} \vDash e_0(e_1) : \tau_r \qquad \vec{\delta} \vDash e_1 : \tau_l \qquad \forall \tau.\ \vec{\delta} \vDash e_1 : \tau \implies \tau_l <: \tau}{\vec{\delta} \vDash e_0 : \tau_l\text{->}\tau_r}$$

Proof:
TODO: ...

Theorem 8.36. Model typing intersection elimination

$$\frac{\vec{\delta} \vDash e : \tau_l \& \tau_r}{\vec{\delta} \vDash e : \tau_l \wedge \vec{\delta} \vDash e : \tau_r}$$

Proof:
TODO: ...

Theorem 8.37. Model typing reflexivity

$$\frac{}{\vec{\delta} \vDash \tau <: \tau}$$

Proof:
for $e\ \Gamma$ **assume** $\vec{\delta}, \Gamma \vDash e : \tau$
. $\vec{\delta}, \Gamma \vDash e : \tau$ by identity
$\forall e\ \Gamma.\ \vec{\delta}, \Gamma \vDash e : \tau \implies \vec{\delta}, \Gamma \vDash e : \tau$ by implication and generalization
□ by definition

Theorem 8.38. (Model typing subsumption)

$$\frac{\vec{\delta}, \Gamma \vDash e : \tau_l \qquad \vec{\delta}, \Gamma \vDash \tau_l <: \tau_r}{\vec{\delta}, \Gamma \vDash e : \tau_r}$$

Proof:
assume $\vec{\delta}, \Gamma \vDash e : \tau_l \quad \vec{\delta}, \Gamma \vDash \tau_l <: \tau_r$
. **invert on** $\vec{\delta}, \Gamma \vDash \tau_l <: \tau_r$
. **case** $\forall e'.\ \vec{\delta}, \Gamma \vDash e' : \tau_l \implies \vec{\delta}, \Gamma \vDash e' : \tau_r$
. . $\forall e'.\ \vec{\delta}, \Gamma \vDash e' : \tau_l \implies \vec{\delta}, \Gamma \vDash e' : \tau_r$ by identity
. $\forall e'.\ \vec{\delta}, \Gamma \vDash e' : \tau_l \implies \vec{\delta}, \Gamma \vDash e' : \tau_r$ by inversion
. $\vec{\delta}, \Gamma \vDash e : \tau_l \implies \vec{\delta}, \Gamma \vDash e : \tau_r$ by instantiation
. $\vec{\delta}, \Gamma \vDash e : \tau_r$ by application
□ by implication

Theorem 8.39. (Model typing implication elimination)

$$\frac{\vec{\delta}, \Gamma \vDash e_0 : \tau_l \texttt{->} \tau_r \qquad \vec{\delta}, \Gamma \vDash e_1 : \tau_l}{\vec{\delta}, \Gamma \vDash e_0 \texttt{(} e_1 \texttt{)} : \tau_r}$$

assume $\vec{\delta}, \Gamma \vDash e_0 : \tau_l \texttt{->} \tau_r \qquad \vec{\delta}, \Gamma \vDash e_1 : \tau_l$
. **let** $\vec{\sigma}$ **s.t.** $\vec{\delta}, \vec{\sigma} \vDash \Gamma$ by theorem **??**
. **induct on** $\vec{\delta}, \Gamma \vDash e_0 : \tau_l \texttt{->} \tau_r$
. **case** **TODO: ...**
. **case** $e_0 = F\$p\texttt{=>}e_2 \qquad \vec{\delta}, \Gamma \vDash F : \tau_l \texttt{->} \tau_r$
. **hypo** $\vec{\delta}, \Gamma \vDash F\texttt{(}e_1\texttt{)} : \tau_r$
. **wrt** $F\ p\ e_2$
. . $\Vdash F\texttt{(}e_1\texttt{)}[\vec{\sigma}]$ by theorem 8.51
. . **invert on** $\Vdash F\texttt{(}e_1\texttt{)}[\vec{\sigma}]$
. . **case** $F\texttt{(}e_1\texttt{)}[\vec{\sigma}] = v$
. . **wrt** v
. . . $F\texttt{(}e_1\texttt{)}[\vec{\sigma}] \neq v$ by definition
. . . $\bot$ by application
. . **case** $(F\texttt{(}e_1\texttt{)})[\vec{\sigma}] \rightsquigarrow e_3 \qquad \vec{\delta}, \Gamma \vDash e_3 : \tau_r$
. . **wrt** e_3
. . . $(F\texttt{(}e_1\texttt{)})[\vec{\sigma}] = F[\vec{\sigma}]\texttt{(}e_1[\vec{\sigma}]\texttt{)}$ by definition
. . . $F[\vec{\sigma}]\texttt{(}e_1[\vec{\sigma}]\texttt{)} \rightsquigarrow e_3$ by substitution
. . . **let** F' **s.t.** $F[\vec{\sigma}] = F'$ by theorem **??** **TODO: ...**
. . . **let** e_1' **s.t.** $e_1[\vec{\sigma}] = e_1'$ by theorem **??** **TODO: ...**
. . . $\mathrm{FV}(e_2[\vec{\sigma}\backslash\mathrm{FV}(p)]) \subseteq \mathrm{FV}(p)$ by theorem **??** **TODO: ...**
. . . $F'\texttt{(}e_1'\texttt{)} \rightsquigarrow e_3$ by substitution
. . . $\texttt{(}F'\$p\texttt{=>}e_2[\vec{\sigma}\backslash\mathsf{FV}(p)]\texttt{)}\texttt{(}e_1'\texttt{)} \rightsquigarrow e_3$ by definition
. . . $\texttt{(}F[\vec{\sigma}]\$p\texttt{=>}e_2[\vec{\sigma}\backslash\mathsf{FV}(p)]\texttt{)}\texttt{(}e_1[\vec{\sigma}]\texttt{)} \rightsquigarrow e_3$ by substitution
. . . $(\texttt{(}F\$p\texttt{=>}e_2\texttt{)}\texttt{(}e_1\texttt{)})[\vec{\sigma}] = \texttt{(}F[\vec{\sigma}]\$p\texttt{=>}e[\vec{\sigma}\backslash\mathsf{FV}(p)]\texttt{)}\texttt{(}e_1[\vec{\sigma}]\texttt{)}$ by definition
. . . $(\texttt{(}F\$p\texttt{=>}e_2\texttt{)}\texttt{(}e_1\texttt{)})[\vec{\sigma}] \rightsquigarrow e_3$ by substitution
. . . $\vec{\delta}, \Gamma \vDash \texttt{(}F\$p\texttt{=>}e\texttt{)}\texttt{(}e_1\texttt{)} : \tau_r$ by definition
. . $\vec{\delta}, \Gamma \vDash \texttt{(}F\$p\texttt{=>}e\texttt{)}\texttt{(}e_1\texttt{)} : \tau_r$ by inversion
. **case** $\vec{\delta}, \vec{\sigma} \vDash \Gamma \qquad e_0[\vec{\sigma}] \rightsquigarrow e_0' \qquad \vec{\delta}, \Gamma \vDash e_0' : \tau_l \texttt{->} \tau_r$
. **hypo** $\vec{\delta}, \Gamma \vDash e_0' : \tau_l \texttt{->} \tau_r \implies \vec{\delta}, \Gamma \vDash e_0'\texttt{(}e_1\texttt{)} : \tau_r$
. **wrt** $\vec{\sigma}\ e_0'$
. . $\vec{\delta}, \Gamma \vDash e_0'\texttt{(}e_1\texttt{)} : \tau_r$ by application
. . $e_0[\vec{\sigma}]\texttt{(}e_1\texttt{)} \rightsquigarrow e_0'\texttt{(}e_1\texttt{)}$
. . $(e_0\texttt{(}e_1\texttt{)})[\vec{\sigma}] \rightsquigarrow e_0'\texttt{(}e_1\texttt{)}$
. . $\vec{\delta}, \Gamma \vDash e_0\texttt{(}e_1\texttt{)} : \tau_r$
. $\vec{\delta}, \Gamma \vDash e_0\texttt{(}e_1\texttt{)} : \tau_r$ by induction

□

Theorem 8.40. Model typing reduced implication elimination

$$\frac{\vec{\delta}, \Gamma \vDash \texttt{(}F\$p\texttt{=>}e\texttt{)} : \tau_l \texttt{->} \tau_r \qquad \vec{\delta}, \Gamma \vDash e_1 : \tau_l \qquad F = \epsilon \vee \vec{\delta}, \Gamma \vDash F\texttt{(}e_1\texttt{)} : \tau_r}{\vec{\delta}, \Gamma \vDash \texttt{(}F\$p\texttt{=>}e\texttt{)}\texttt{(}e_1\texttt{)} : \tau_r}$$

assume $\Vdash e_1[\vec{\sigma}]$
. **let** $\vec{\sigma}$ **s.t.** $\vec{\delta}, \vec{\sigma} \vDash \Gamma$ by theorem 8.50
. $\Vdash e_1[\vec{\sigma}]$ by theorem 8.51
. **induct on** $\Vdash e_1[\vec{\sigma}]$
. **case** $e_1[\vec{\sigma}] = v_1$
. **wrt** v_1
. . $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r$ by theorem 8.41
. **case** $e_1[\vec{\sigma}] \rightsquigarrow e_1' \quad \Vdash e_1'$
. **hypo** $\Vdash e_1' \implies \vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1') : \tau_r$
. **wrt** e_1'
. . $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1') : \tau_r$ by application
. . $(F\$p\texttt{=>}e)[\vec{\sigma}](e_1[\vec{\sigma}]) \rightsquigarrow (F\$p\texttt{=>}e)[\vec{\sigma}](e_1')$ by definition
. . $\forall x.\ x \notin \mathbf{FV}(e_1')$ by theorem 8.48
. . $e_1' = e_1'[\vec{\sigma}]$ by by theorem 8.49
. . $(F\$p\texttt{=>}e)[\vec{\sigma}](e_1[\vec{\sigma}]) \rightsquigarrow (F\$p\texttt{=>}e)[\vec{\sigma}](e_1'[\vec{\sigma}])$ by substitution
. . $((F\$p\texttt{=>}e)(e_1))[\vec{\sigma}] \rightsquigarrow ((F\$p\texttt{=>}e)(e_1'))[\vec{\sigma}]$ by definition
. . $\vec{\sigma} \sqcup \epsilon = \vec{\sigma}$ by definition
. . $((F\$p\texttt{=>}e)(e_1))[\vec{\sigma}] \rightsquigarrow ((F\$p\texttt{=>}e)(e_1'))[\vec{\sigma} \sqcup \epsilon]$ by substitution
. . $\Gamma \sqcup \epsilon = \Gamma$ by definition
. . $\vec{\delta}, \epsilon \vDash \epsilon$ by definition
. . $\vec{\delta}, \Gamma \sqcup \epsilon \vDash (F\$p\texttt{=>}e)(e_1')$ by substitution
. . $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r$ by definition
. $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r$ by induction
□ by implication

Theorem 8.41. Model typing fully reduced implication elimination

$$\frac{\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e) : \tau_l \texttt{->} \tau_r \qquad F = \epsilon \lor \vec{\delta}, \Gamma \vDash F(e_1) : \tau_r \qquad \vec{\delta} \vDash e_1 : \tau_l \qquad \vec{\delta}, \vec{\sigma} \vDash \Gamma \qquad e_1[\vec{\sigma}] = v_1}{\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r}$$

Proof:
assume $F = \epsilon \lor \vec{\delta}, \Gamma \vDash F : \tau_l \texttt{->} \tau_r$ **TODO: add more assumptions**
. $\vec{\delta} \vDash e_1[\vec{\sigma}] : \tau_l$ by theorem 8.45
. $\vec{\delta} \vDash v_1 : \tau_l$ by substitution
. **invert on** $F = \epsilon \lor \vec{\delta}, \Gamma \vDash F : \tau_l \texttt{->} \tau_r$
. **case** $F = \epsilon$
. . $\vec{\delta}, \Gamma \vDash F\$p\texttt{=>}e : \tau_l \texttt{->} \tau_r$ by theorem **??**
. . $\vec{\delta}, \Gamma \vDash \$p\texttt{=>}e : \tau_l \texttt{->} \tau_r$ by substitution
. . **let** $\vec{\sigma}'$ **s.t.** $p \equiv v_1 \dashv \vec{\sigma}'$ by theorem 8.46
. . **for** e'
. . . $\neg \epsilon[\vec{\sigma}](v_1) \rightsquigarrow e'$ by definition
. . . $\neg F[\vec{\sigma}](v_1) \rightsquigarrow e'$ by substitution
. . $\forall e'.\ \neg F[\vec{\sigma}](v_1) \rightsquigarrow e'$ by generalization
. . $(F[\vec{\sigma}]\$p\texttt{=>}e[\vec{\sigma} \backslash \mathbf{FV}(p)])(v_1) \rightsquigarrow e[\vec{\sigma} \backslash \mathbf{FV}(p)][\vec{\sigma}]$ by definition
. . $\forall x.\ x \in \mathbf{FV}(p) \iff x \in \mathbf{dom}(\vec{\sigma}')$ by theorem 8.42
. . $\vec{\sigma} \backslash \mathbf{FV}(p) = \vec{\sigma} \backslash \mathbf{dom}(\vec{\sigma}')$ by theorem 8.43

. . $e[\vec{\sigma}\backslash\mathbf{dom}(\vec{\sigma}')][\vec{\sigma}'] = e[\vec{\sigma} \sqcup \vec{\sigma}']$ by theorem 8.44
. . $(F[\vec{\sigma}]\$p\texttt{=>}e[\vec{\sigma}\backslash\mathsf{FV}(p)])(v_1) \rightsquigarrow e[\vec{\sigma}\backslash\mathbf{dom}(\vec{\sigma}')][\vec{\sigma}]$ by substitution
. . $(F[\vec{\sigma}]\$p\texttt{=>}e[\vec{\sigma}\backslash\mathsf{FV}(p)])(v_1) \rightsquigarrow e[\vec{\sigma} \sqcup \vec{\sigma}']$ by substitution
. . $(F\$p\texttt{=>}e)[\vec{\sigma}](v_1) \rightsquigarrow e[\vec{\sigma} \sqcup \vec{\sigma}']$ by definition
. . $(F\$p\texttt{=>}e)[\vec{\sigma}](e_1[\vec{\sigma}]) \rightsquigarrow e[\vec{\sigma} \sqcup \vec{\sigma}']$ by substitution
. . $(F\$p\texttt{=>}e)[\vec{\sigma}](e_1[\vec{\sigma}]) = ((F\$p\texttt{=>}e)(e_1))[\vec{\sigma}]$ by definition
. . $((F\$p\texttt{=>}e)(e_1))[\vec{\sigma}] \rightsquigarrow e[\vec{\sigma} \sqcup \vec{\sigma}']$ by substitution
. . $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r$ by definition
. **case** $\vec{\delta}, \Gamma \vDash F(e_1) : \tau_r$
. . **let** e' **s.t.** $(F(e_1))[\vec{\sigma}] \rightsquigarrow e' \land \vec{\delta}, \Gamma \vDash e' : \tau_r$ by theorem **??**
. . $((F\$p\texttt{=>}e)(e_1))[\vec{\sigma}] \rightsquigarrow e'$ by definition
. . $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r$ by definition
. $\vec{\delta}, \Gamma \vDash (F\$p\texttt{=>}e)(e_1) : \tau_r$ by inversion
□ by implication

Theorem 8.42. (Pattern matching consistency)

$$\frac{p \equiv v \dashv \vec{\sigma}}{\forall x.\ x \in \mathbf{FV}(p) \iff x \in \mathbf{dom}(\vec{\sigma})}$$

Proof:
TODO: ...
□

Theorem 8.43. (Consistency diffing)

$$\frac{\forall x.\ x \in X_l \iff x \in X_r}{\vec{\sigma}\backslash X_l = \vec{\sigma}\backslash X_r}$$

Proof:
TODO: ...
□

Theorem 8.44. (Concatenation Substitution)

$$\frac{}{e[\vec{\sigma}\backslash\mathbf{dom}(\vec{\sigma}')][\vec{\sigma}'] = e[\vec{\sigma} \sqcup \vec{\sigma}']}$$

Proof:
TODO: ...
□

Theorem 8.45. (Model typing valuation)

$$\frac{\vec{\delta}, \Gamma \vDash e : \tau \qquad \vec{\delta}, \vec{\sigma} \vDash \Gamma}{\vec{\delta} \vDash e[\vec{\sigma}] : \tau}$$

Proof:
assume $\vec{\delta}, \Gamma \vDash e : \tau \quad \vec{\delta}, \vec{\sigma} \vDash \Gamma$
TODO: ...
□

Theorem 8.46. (Model typing pattern matching)

$$\frac{\vec{\delta}, \Gamma \vDash \texttt{\$}p\texttt{=>}e : \tau_l \texttt{->} \tau_r \qquad \vec{\delta} \vDash v : \tau_l}{\exists \vec{\sigma}.\ p \equiv v \dashv \vec{\sigma}}$$

Proof:
assume $\vec{\delta}, \Gamma \vDash \texttt{\$}p\texttt{=>}e : \tau_l \texttt{->} \tau_r \quad \vec{\delta} \vDash v : \tau_l$
TODO: ...
□

Theorem 8.47. (Well-formed function valuation)

$$\frac{\Vdash F}{\exists v.\ v = F}$$

Proof.
assume $\Vdash F$
. **invert on** $\Vdash F$
. **case** $v = F$
. . $v = F$ by identity
. **case** $F \rightsquigarrow e$
. **wrt** e
. . $\neg F \rightsquigarrow e$ by definition
. . $\bot$ by application
. $v = F$ by inversion
□ □

Theorem 8.48. (Reduction closed)

$$\frac{e \rightsquigarrow e'}{\forall x.\ x \notin \mathbf{FV}(e')}$$

Proof:
assume $e \rightsquigarrow e'$
TODO: ...
□

Theorem 8.49. (Closed substitution)

$$\frac{\forall x.\ x \notin \mathbf{FV}(e)}{e = e[\vec{\sigma}]}$$

Proof:
assume $\forall x.\ x \notin \mathbf{FV}(e)$
TODO: ...
□

Theorem 8.50. (Model typing assignability)

$$\frac{\vec{\delta}, \Gamma \vDash e : \tau}{\exists \vec{\sigma}.\ \vec{\delta}, \vec{\sigma} \vDash \Gamma}$$

Proof:
TODO: ...

Theorem 8.51. (Model typing soundness)

$$\frac{\vec{\delta}, \Gamma \vDash e : \tau}{\forall \vec{\sigma}.\ \vec{\delta}, \vec{\sigma} \vDash \Gamma \implies \Vdash e[\vec{\sigma}]}$$

Proof:

TODO: redo using universal/implication

assume $\vec{\delta}, \vec{\sigma} \vDash \Gamma \quad \vec{\delta}, \Gamma \vDash e : \tau$

. **case** $e = @$

. . **let** v **s.t.** $@ = v$

. . $e[\vec{\sigma}] = v$

. . $\Vdash e[\vec{\sigma}]$

. **case** $\vec{\delta}, \Gamma \vDash e' : \tau' \quad e = <l>e' \quad \tau = <l>\tau'$

. . $\Vdash e'$ by induction hypothesis

. . **case** $e'[\vec{\sigma}] = v$

. . . **let** v' **s.t.** $<l>v = v'$

. . . $<l>e'[\vec{\sigma}] = v'$

. . . $(<l>e')[\vec{\sigma}] = v'$

. . . $e[\vec{\sigma}] = v'$

. . . $\Vdash e[\vec{\sigma}]$

. . **case** $e'[\vec{\sigma}] \rightsquigarrow e'' \quad \Vdash e''$

. . . $<l>e'[\vec{\sigma}] \rightsquigarrow <l>e''$

. . . $\Vdash <l>e''$

. . . $\Vdash <l>e'[\vec{\sigma}]$

. . . $\Vdash (<l>e')[\vec{\sigma}]$

. . . $\Vdash e[\vec{\sigma}]$

. . $\Vdash e[\vec{\sigma}]$ by cases on $\Vdash e'$

. **TODO: remaining introduction cases**

. **case** $x : \tau \in \Gamma \quad x/v \in \vec{\sigma} \quad e = x$

. . $x[\vec{\sigma}] = v$

. . $e[\vec{\sigma}] = v$

. . $\Vdash e[\vec{\sigma}]$

. **case** $e[\vec{\sigma}] \rightsquigarrow e' \quad \vec{\delta}, \Gamma \vDash e' : \tau$

. . $\Vdash e'[\vec{\sigma}]$ by induction hypothesis

. . $\Vdash e[\vec{\sigma}]$

. $\Vdash e[\vec{\sigma}]$ by induction on $\vec{\delta}, \Gamma \vDash e : \tau$

□

TODO: Cretin's corresponding theorem is by definition of pretypes on p. 125

NOTE: The induction hypothesis includes the generalized assumption, e.g. $\forall e'.\ e' < e \implies Q(e')$ if inducting on e or $\forall e'.\ (P(e') \implies P(e)), P(e') \implies Q(e')$ if inducting on predicate P

NOTE: we induct on $\vec{\delta}, \Gamma \vDash e : \tau$ instead of e, as the predicate acts as a guard/ordering in lieu of a decreasing e. This allows us to use the induction hypothesis on the reduction step result in the elimination case.

NOTE: Kozen says, "Intuitively, one can appeal to the coinductive hypothesis as long as there has been progress in observing the elements of the stream (guardedness) and there is no further analysis of the tails (opacity)". Kozen demonstrates a legal proof by induction on infinite streams too

Definition 8.45. $\boxed{\vec{\delta}, \vec{\sigma} \vDash \Gamma}$

$$\frac{}{\vec{\delta}, \vec{\sigma} \vDash \epsilon} \qquad \frac{\vec{\delta}, \vec{\sigma} \vDash \Gamma \qquad \vec{\delta} \vDash v : \tau}{\vec{\delta}, \vec{\sigma}\ x/v \vDash \Gamma\ x : \tau}$$