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

2.1 Language of types

Parametric types. Universal types. Existential type. System F-style. Parameterization of types indexed by types (i.e. second order).

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.

Combination types. One of the advantages of untyped programs is that they may be written in a flexible manner. Subtyping is necessary safely reflect the flexibility of compositions in programs, without too many false failures. Another main advantage of untyped programs is that users don't have to provide type specifications. Thus, a general way of constructing types from compositions encountered in the the program is necessary. Some compositions indicate that a type should strengthen, and some compositions indicate that a type should weaken. To this end, the type language uses intersection and union combinators, whose semantics are degenerate versions of those in set-theory.

For instance, when inferring the type of a function, the system's goal is to infer the weakest valid parameter type and the strongest valid return type for a function definition. It strengthens the parameter type with intersection and weakens the return type with union according to the function body, to arrive at a valid type for the function.

By contrast, the liquid type language relies on the less flexible tagged unions of ML datatypes, which is sufficient in its setting since those types are specified by the user. Likewise, it does not rely on union to weaken to a valid return type. Instead, it weakens to the strongest valid return type by dropping conjunctions from the return type's qualifiers until a valid return type is found.

Inductive types. Similar to ML datatypes.

Constraint types. In addition to expressing the shapes of terms, the system should be able express relations between terms, such as "a list has the length of some natural number". Rather than using a distinct syntax for relational predicates, the type language treats relations as just another type thereby reusing machinery already available for types, such as existential types, union types, and inductive types. Since parametric types are second order, constraining relations requires subtyping. Thus, parametric types are extended with constraints in the form of subtyping.

2.2 Type Inference

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).

$$\text{natList}(\alpha) = \texttt{LFP[R](BOT | (<zero> @)*(<nil> @) | (EXI[N L ; N*L <: R](<succ> N)*(<cons> (}\alpha\texttt{*L)))}$$

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

$$\texttt{ALL[T] T -> ALL[X] X -> EXI[Y ; X*Y <:}\ \text{natList}(\texttt{T})\texttt{] Y}$$

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

Definition 3.1. (Expressions)

$$
\begin{array}{rcl}
e & ::= & x \mid \texttt{@} \mid \texttt{<}l\texttt{>}e \mid R \mid e\texttt{,}e \mid F \mid e\texttt{.}l \mid \texttt{(}e\texttt{)(}e\texttt{)} \mid \\
& & e \texttt{ |> } e \mid \texttt{loop(}e\texttt{)} \mid \texttt{let } x\texttt{:}\tau \texttt{ = } e \texttt{ in } e \mid \texttt{(}e\texttt{)} \\
R & ::= & \epsilon \mid R\, r \\
r & ::= & \texttt{\$}l\texttt{=>}e \\
F & ::= & \epsilon \mid F\, f \\
f & ::= & \texttt{\$}p\texttt{=>}e
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & x \mid \texttt{@} \mid \texttt{<}l\texttt{>}p \mid K \mid p\texttt{,}p \mid \texttt{(}p\texttt{)} \\
K & ::= & \epsilon \mid K\, k \\
k & ::= & \texttt{\$}l\texttt{=>}p
\end{array}
$$

Definition 3.2. Others

$$
\begin{array}{rcl}
v & ::= & \texttt{@} \mid \texttt{<}l\texttt{>}\ v \mid G \mid v\texttt{,}v \mid \texttt{(}v\texttt{)} \mid F \\
G & ::= & \epsilon \mid G\, g \\
g & ::= & \texttt{\$}l\texttt{=>}v
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma & ::= & \epsilon \mid \Sigma\, \sigma \\
\sigma & ::= & x/v
\end{array}
$$

Definition 3.3. $\boxed{e \rightsquigarrow e}$ Progression

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

$$\frac{\texttt{\$}l\texttt{=>}v \in G \qquad \forall\, e.\ \texttt{\$}l\texttt{=>}e \in G \implies e = v}{G\texttt{.}l \rightsquigarrow v} \qquad \frac{e \rightsquigarrow e'}{\texttt{(}e\texttt{)(}e_a\texttt{)} \rightsquigarrow \texttt{(}e'\texttt{)(}e_a\texttt{)}} \qquad \frac{e \rightsquigarrow e'}{\texttt{(}v\texttt{)(}e\texttt{)} \rightsquigarrow \texttt{(}v\texttt{)(}e'\texttt{)}}$$

$$\frac{F\texttt{(}v\texttt{)} \rightsquigarrow e' \qquad \mathrm{FV}(e) \subseteq \mathrm{FV}(p)}{\texttt{(}F\texttt{\$}p\texttt{=>}e\texttt{)(}v\texttt{)} \rightsquigarrow e'} \qquad \frac{p \equiv v \dashv \Sigma \qquad \forall e'.\ \neg F\texttt{(}v\texttt{)} \rightsquigarrow e'}{\texttt{(}F\texttt{\$}p\texttt{=>}e\texttt{)(}v\texttt{)} \rightsquigarrow e[\Sigma]}$$

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

Definition 3.4. $\boxed{e \rightsquigarrow e}$ Progression sugar

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

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

The programming language is pure and functional. Its syntax and dynamic semantics are fairly standard. The main departure from tradition is that its function and application rules subsume pattern matching. This departure enables a more direct correspondence between the structures of programs and their types, but it is not a necessary condition. The syntax is given in Fig. ??. It includes functions with pattern matching, records, a fixed point combinator, let binding, tags for

discriminating cases, and application. A function consists of a sequence of paths, where each path maps a pattern to an expression. A record consists of a sequence of fields, where each field maps a unique label to an expression. The type language includes tag types, field types, implications, unions, intersections, inductions, existentials, universals, top, and bottom. The existential type consists of multiple bound variables, a payload containing the bound variables, and a subtyping constraint over the bound variables. If the bound variables aren't indicated, then all variables in the payload are assumed to be bound variables. If the subtyping constraint isn't indicated, then it is assumed to be a tautology, such as (`unit<:unit`). The universal type consists of a bound variable, the variable's upper bound, and a payload. If the upper bound is not indicated, it is assumed to be the top type (`top`). The typing semantics rely on a typing environment for keeping track of typings of term variables. The subtyping semantics rely on subtyping environment for keeping track of and constraints on type variables. Note that the syntax of the subtyping environment allows an upper bound constraint over a type rather than merely a type variable to allow for relational constraints.
TODO: update tag syntax: `cons;cons;e:cons//cons//T nil;():nil//unit`

4 STATIC SEMANTICS

The static semantics is a system for proving that an expression satisfies a type. The type syntax is rich and allows free type variables along with universally and existentially quantified type variables. Thus, One can think of the static semantics and either type checking or type inference, since checking against free type variable or an existentially quantified type, requires constructing types to witness the variables allowed by the type syntax. Furthermore, when verifying elimination forms of expressions, such as application, the static semantics translates the eliminations into subtyping constraints and then learns constraints on type variables necessary for the subtyping constraint to hold.

4.1 Types

ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd ddddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd dddddd

Definition 4.1. Types

$$
\begin{aligned}
\tau &::= \alpha \mid \texttt{@} \mid \texttt{<}l\texttt{>}\tau \mid l\texttt{->}\tau \mid \tau\texttt{->}\tau \mid \tau\texttt{|}\tau \mid \tau\texttt{\&}\tau \mid \tau\texttt{\textbackslash}\tau \mid \\
&\quad \texttt{EXI[}\mathrm{A}\,\Delta\texttt{]}\tau \mid \texttt{ALL[}\mathrm{A}\,\Delta\texttt{]}\tau \mid \texttt{LFP[}\alpha\texttt{]}\tau \\
\mathrm{A} &::= \epsilon \mid \mathrm{A}\,\alpha \\
\Delta &::= \epsilon \mid \Delta\,\delta \\
\delta &::= \texttt{;}\tau\texttt{<:}\tau
\end{aligned}
$$

ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd ddddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd dddddd ddddddddd ddddddd ddddd ddddddd dddddd dddddd dddddd dddddd dddddd dddddd

Definition 4.2. Internals

$$
\begin{array}{rcl}
\Gamma & ::= & \epsilon \mid \Gamma\ \gamma \\
\gamma & ::= & x : \tau \\
\\
M & ::= & \epsilon \mid M\ m \\
m & ::= & \alpha \\
N & ::= & \epsilon \mid N\ n \\
n & ::= & \alpha \\
\\
Z & ::= & \epsilon \mid Z\ z \\
z & ::= & \langle M, \Delta \rangle \\
\\
\mathrm{T} & ::= & \epsilon \mid \mathrm{T}\ \tau \\
\\
\Pi & ::= & \epsilon \mid \Pi\ \pi \\
\pi & ::= & \langle M, \Delta, \tau \texttt{->} \tau \rangle \\
\\
\Omega & ::= & \epsilon \mid \Omega\ \omega \\
\omega & ::= & \alpha / \tau
\end{array}
$$

4.2 Proof Typing

Definition 4.3. Proof Typing $\boxed{\Gamma \vdash e : \tau \dashv Z}$

$$\frac{\langle M, \Delta \rangle \in Z}{\Gamma \vdash \texttt{@} : \texttt{@} \dashv Z} \qquad \frac{\langle M, \Delta \rangle \in Z \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau \dashv Z} \qquad \frac{\Gamma \vdash e : \tau \dashv Z}{\Gamma \vdash \texttt{<}l\texttt{>}e : \texttt{<}l\texttt{>}\tau \dashv Z}$$

$$\frac{\Gamma \vdash R : \tau_0 \dashv Z_0 \qquad Z_0 \rightsquigarrow Z_1 \qquad \Gamma \vdash e : \tau_1 \dashv Z_1}{\Gamma \vdash R\ \texttt{*}l\texttt{=}e : \tau_0\ \texttt{\&}\ l : \tau_1 \dashv Z_1} \qquad \frac{Z, \Gamma \vdash F \blacktriangle \Pi, \mathrm{T}_n \qquad \Gamma \vdash \Pi \equiv \mathrm{T}}{\Gamma \vdash F : \texttt{\&}(\mathrm{T}) \dashv Z}$$

$$\frac{\Gamma \vdash e : \tau_0 \dashv Z_0 \qquad Z_0 \rightsquigarrow Z_1 \qquad \tau_0 <: l : \alpha \dashv Z_1}{\Gamma \vdash e\,.\,l : \alpha \dashv Z_1}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \qquad Z_0 \rightsquigarrow Z_1 \qquad \Gamma \vdash e_1 : \tau_1 \dashv Z_1 \qquad Z_1 \rightsquigarrow Z_2 \qquad \tau_0 <: \tau_1 \texttt{->} \alpha \dashv Z_2}{\Gamma \vdash e_0\texttt{(}e_1\texttt{)} : \alpha \dashv Z_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \alpha_{h^+} \texttt{->} \tau \dashv Z_0 \qquad Z_0 \rightsquigarrow Z_1 \qquad \tau <: \alpha_l \texttt{->} \alpha_r \dashv Z_1 \\ \mathrm{FTV}(\Gamma) \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l \texttt{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}\end{array}}{\Gamma \vdash \texttt{loop(}e\texttt{)} : \texttt{ALL[}\alpha'_l\texttt{]}\alpha'_l\texttt{->EXI[}\alpha'_r\texttt{;}\alpha'_l\texttt{*}\alpha'_r\texttt{<:LFP[}\alpha_{h^-}\texttt{] |(T)]]}\alpha'_r \dashv Z_0}$$

Definition 4.4. $\boxed{Z, \Gamma \vdash F \blacktriangle \Pi, T}$

$$\frac{\begin{array}{c}Z_0, \Gamma_0 \vdash f \blacktriangle \Pi, T \qquad p : \tau_l \dashv \Gamma_1 \qquad \mathrm{neg}(\tau_l, T) = \tau'_l \\ Z_0 \rightsquigarrow Z_1 \qquad \Gamma_0 \sqcup \Gamma_1 \vdash e : \tau_r \dashv Z_1\end{array}}{Z_0, \Gamma_0 \vdash F\ \texttt{\$}p\texttt{=>}e \blacktriangle \Pi \sqcup \overline{\left\langle M, \Delta, \tau'_l \texttt{->} \tau_r \right\rangle}^{\langle M, \Delta \rangle \in Z_1}, T\ \tau_0}$$

Definition 4.5. $\boxed{Z \rightarrowtail Z}$ (Universe ordering)

$$\frac{\forall M_1\ \Delta_1.\ \langle M_1, \Delta_1\rangle \in Z_1 \implies \exists M_0\ \Delta_0.\ \langle M_0, \Delta_0\rangle \in Z_0 \wedge M_0 \preceq M_1 \wedge \Delta_0 \preceq \Delta_1}{Z_0 \rightarrowtail Z_1}$$

The typing is given in Fig. ??. Most of the rules are fairly standard. The rule for function typing is a bit special in that it treats a function as a sequence of paths whose type is an intersection of implications, rather than having a separate pattern matching rule. Likewise, the type of a record is an intersection of field types. The let-binding rule allows for prenex polymorphism by generalizing via subtyping.

The type inference procedure is given in Fig. ??. The procedure depends on four parameters: an type variable set, a subtyping environment, a typing environment, and an expression. The variable set indicates if a type variable's assigned type is allowed to be strengthened or weakened during unification. The subtyping environment indicates the assumed constraints on types containing type variables, including relational constraints and constraints over single type variables, which we also refer to as type assignments. The typing environment indicates the assumed constraints on term variables. The expression indicates the inhabitant of the type that is to be inferred. The procedure returns a set of triples, where each triple consists of a variable set, a subtyping environment, and a type. The unit case simply returns a singleton set with the unit type and the environments unchanged. The variable case uses the variable as a key to find its type in the typing environment. It returns a singleton set containing the found type along with the adjustment set and subtyping environment. The tag case infers the type of its constituent type and uses its label to construct a tag type. The record case infers the type of its fields and intersects the constructed field types together.

The function case is of particular importance to type inference in an untyped setting. For each path in the function, it extracts the term variables of the pattern and associates the term variables with fresh type variables. It infers the body of each path with an updated variable set and an updated typing environment containing the fresh type variables of the pattern. By adding the pattern's type variables to the adjustment variable set, it implies that the parameter type of the path can be strengthened by applications occurring in the body of the path. This enables adjusting a parameter type to reflect all of its occurrences in the body, rather than reflecting just its first occurrence and failing on subsequent occurrences. Before returning the inferred implication type for the path, the case removes the the pattern's type variables from the adjustment variable set, ensuring that those type variables cannot be strengthened or weakened from the outside. As with the record case, the inferred implications are intersected together.

The projection case infers the type of the record expression. It then calls unify to solve for the projected type by finding a single field type that is subtyped by the record type.

The application case also plays an important role in type inference of untyped programs. It creates a fresh type variable as a placeholder for the inferred type of the application result. It infers the type of the function and the argument, and then solves for the result type by unifying the function's type with an implication from the argument type to the result type. It adds the result type variable to the adjustment variable set when unifying to allow inferring types accounting for all the paths that the function might take. After unifying, it removes the result type variable from the adjustment variable set to ensure that the type cannot be modified from the outside. Additionally, since inference and unification actually return sets of solutions, the application must type check for all function types and all argument types. If even one combination cannot be unified then the inference of application fails, indicated by breaking the for-loop, which then implicitly returns an empty set.

TODO: update application to merely require some function cases to type check

the let-binding case infers the type of the argument and checks that all possible types subtype the annotation and result in a well-typed body. It generalizes type variables in the inferred argument type while maintaining the constraints indicated by the subtyping environment.
TODO: update let-binding to generalize only if something is a function-type: e.g. intersection, univeral, implication
TODO: update let-binding to merely require some bodies to type check

The fix case infers inductive types for the parameters and bodies of its target expression. Moreover, it inductively relates the parameter and body types to each other. First, it infers the type of the target and calls unify to deconstruct it into the antecedent and consequent of an implication type. According to the semantics of fix, the antecedent represents and inductive hypothesis, while the consequent represents and inductive conclusion. Thus, it uses the inferred structures of the antecedent and consequent to construct an inductive relation by ensuring that the antecedent subtypes the inductively bound variable of the resulting inductive type. Finally, the implication type is projected from the constructed relation and generalized with universal.

4.3 Proof Subtyping

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 an annex. 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.
TODO: separate out syntactic sugar like pairs, pipe, and let, parens
TODO: bifurcate proof subtyping into either duals or weakening vs strengthening rules
TODO: separate out semantic sugar like combination of implication with union/intersection
TODO: note that constraint types are insufficient to represent union, when union is on the rhs (or intersection on the left)
TODO: add color to program and type syntax for readabilitiy

Definition 4.6. $\boxed{\tau <: \tau \dashv M, \Delta}$ Proof Subtyping
TODO: double check LFP rules against code

$$\frac{}{\tau <: \tau \dashv M, \Delta} \qquad \frac{\tau_{ll} <: \tau_r \dashv M_0, \Delta_0 \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_{lr} <: \tau_r \dashv M_1, \Delta_1}{\tau_{ll} \mid \tau_{lr} <: \tau_r \dashv M_1, \Delta_1}$$

$$\frac{Q \dashv M_0, \Delta_0 \qquad \mathrm{A} \# \tau_r \qquad M_0 \sqcup \mathrm{A} \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_l <: \tau_r \dashv M_1, \Delta_1}{\mathtt{EXI[}\,\mathrm{A}\ Q\,\mathtt{]}\,\tau_l <: \tau_r \dashv M_1, \Delta_1}$$

$$\frac{\alpha \notin M_0 \qquad M_0, \Delta_0 \vdash \alpha/\tau_l <:^{\star} \Delta_m \qquad M_0, \Delta_0 \vdash \alpha <:^{\dagger} \mathrm{T} \qquad M_0, \Delta_0 \vdash \alpha/\tau_l \dagger \Delta_{rel} \qquad M_0 \preceq M_1 \qquad \Delta_0 \sqcup \Delta_m \preceq \Delta_1 \qquad \Delta_{rel}\ \tau_l \mathtt{<:\&(T)} \dashv M_1, \Delta_1}{\tau_l <: \alpha \dashv M_1, \Delta_1\ \tau_l <: \alpha}$$

$$\frac{\alpha \in M_0 \qquad M_0, (\Delta_0 \sqcup \mathrm{factor}(\Delta_0, \alpha)) \vdash \alpha <: \mathrm{T} \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \&(\tau) <: \tau_r \dashv M_1, \Delta_1}{\alpha <: \tau_r \dashv M_1, \Delta_1} \qquad \frac{\tau_l <: \tau_r \dashv M_0, \Delta_0 \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad Q \dashv M_1, \Delta_1}{\tau_l <: \mathtt{EXI[}\,\mathrm{A}\ Q\,\mathtt{]}\,\tau_r \dashv M_1, \Delta_1}$$

$$\frac{\tau_l <: \tau_{rl} \dashv M, \Delta}{\tau_l <: \tau_{rl} \mid \tau_{rr} \dashv M, \Delta} \qquad \frac{\tau_l <: \tau_{rr} \dashv M, \Delta}{\tau_l <: \tau_{rl} \mid \tau_{rr} \dashv M, \Delta} \qquad \frac{\tau_l[\alpha/\tau_r] <: \tau_r \dashv M, \Delta}{\mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_l <: \tau_r \dashv M, \Delta}$$

$$\frac{\mathrm{decidable}(\Delta_0, \tau_l, \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r) \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_l <: \tau_r[\alpha/\mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r] \dashv M, \Delta_1}{\tau_l <: \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r \dashv M, \Delta_1}$$

$$\frac{m \in \mathrm{FTV}(\tau_l) \qquad m \in M_0 \qquad \mathrm{norm}(\tau_l \mathtt{<:} \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r) = \tau_l' \mathtt{<:} \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r' \qquad \tau_l' \mathtt{<:} \tau \in \mathrm{norm}(\Delta_0) \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau <: \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r' \dashv M_1, \Delta_1}{\tau_l <: \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r \dashv M_1, \Delta_1}$$

$$\frac{M_0, \Delta_0, \mathrm{FV}(\tau_l) \vdash \Omega \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_l[\Omega] <: \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r \dashv M_1, \Delta_1 \qquad \mathrm{WF}(\tau_l \mathtt{<:} \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r)}{\tau_l <: \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r \dashv M_1, \Delta_1\ \tau_l <: \mathtt{LFP[}\,\alpha\,\mathtt{]}\,\tau_r}$$

$$\frac{\tau_l <: \tau_r \dashv M, \Delta}{\mathtt{<}l\mathtt{>}\tau_l <: \mathtt{<}l\mathtt{>}\tau_r \dashv M, \Delta} \qquad \frac{\tau_l <: \tau_r \dashv M, \Delta}{l\mathtt{:}\tau_l <: l\mathtt{:}\tau_r \dashv M, \Delta}$$

$$\frac{\tau_{rl} <: \tau_{ll} \dashv M_0, \Delta_0 \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_{lr} <: \tau_{rr} \dashv M_1, \Delta_1}{\tau_{ll}\mathtt{->}\tau_{lr} <: \tau_{rl}\mathtt{->}\tau_{rr} \dashv M_1, \Delta_1}$$

dddd ddd ddd ddd

Definition 4.7. $\boxed{\tau <: \tau \dashv M, \Delta}$ Proof subtyping annex

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

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

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

$$\frac{\alpha \in M_0 \qquad \Delta_0 \vdash \mathrm{T} <: \alpha \qquad M_0 \preceq M_1 \qquad \Delta_0 \preceq \Delta_1 \qquad \tau_l <: \mid (\mathrm{T}) \dashv M_1, \Delta_1}{\tau_l <: \alpha \dashv M_1, \Delta_1}$$

$$\frac{\alpha \notin M_0 \qquad M_0, \Delta_0 \vdash \Delta_m <:^\star \alpha / \tau_r \qquad M_0, \Delta_0 \vdash \mathrm{T} <:^\dagger \alpha \qquad M_0 \preceq M_1 \qquad \Delta_0 \sqcup \Delta_m \preceq \Delta_1 \qquad \mid (\mathrm{T}) <: \tau_r \dashv M_1, \Delta_1}{\alpha <: \tau_r \dashv M_1, \Delta_1 \; \alpha <: \tau_r}$$

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

$$\frac{M, \Delta \vdash \tau_l <: \tau_r \mid \tau_n}{M, \Delta \vdash \tau_l \backslash \tau_n <: \tau_r} \qquad \frac{\mathrm{DF}(\tau_r \backslash \tau_n) \qquad \tau_l <: \tau_r \dashv M, \Delta \qquad \forall M' \; \Delta'. \; M \preceq M' \implies \Delta \preceq \Delta' \implies \neg(\tau_l <: \tau_n \dashv M', \Delta')}{\tau_l <: \tau_r \backslash \tau_n \dashv M, \Delta}$$

The subtyping is given in Fig. ??. Some of the rules are fairly standard, including implication, the union rules, and intersection rules. Note that in addition to left and right rules, union and intersection each have rules for interacting with implication's antecedent and consequent, respectively. The constraint rule checks that a subtyping relation exists as a constraint in the subtyping environment. The right induction rule is standard and simply unrolls the induction. The left induction rule relies on the induction principle to construct an inductive constraint hypothesis. The field and tag rules simply check that the labels match and subtyping holds for their constituent types. The existential rules are quite special, as they involve a subtyping constraint as part of a second-order comprehension. The left existential rule checks that subtyping holds for all variations of the payload where the subtyping constraint holds. The right existential rule checks that subtyping holds for some variation of the payload where the constraint holds. The left universal rule checks that subtyping holds for some variation of the payload consistent with the variable's upper bound. The right universal rule checks that subtyping holds for all variations of the payload consistent with the variable's upper bound.

The subtype unification procedure is given in Fig. ??. Unification depends on four parameters: a set of type variables, a subtyping environment, and two types. The set of type variables indicates the type of variables that may be adjusted (triggered by the function and application cases of inference).

The subtyping environment indicates constraints on type variables. the left type indicates the stronger type, and the right type indicates the weaker type. Both types may contain type variables that are unsolved or partially solved. The procedure returns a set of subtyping environments.

The procedure begins by checking if the two types are syntactically equal. If equal then subtyping holds but there is nothing that can be unified without circular references so the assumed subtyping environment is returned unchanged in a singleton set.

The procedure then pattern matches on the left and right types. The first two cases handle assigning variables to types. The right-variable case first looks up a solution for the variable. If a solution is found, then unification proceeds between the original left type and the solution for the right type. In the case where unification fails, but the type variable is adjustable, the procedure updates the type variable with the union of the left type and the found right type (assuming the constraint is well-formed, which includes avoiding circular references). If no solution is found for the right variable, then the procedure looks for any relational constraints in the subtyping environment where the left side contains the type variable. It checks that the left type is consistent with the relational constraints. If there are no relational constraints, then the variable is assigned to the left type (if the constraint is well-formed). The left-variable rule is similar to the right-variable rule with one difference being that it uses intersection to strengthen the type variable in the case of adjustment. In the case of checking relational constraints, it ensures that the right type is weaker than what the relational constraints would allow.

The subsequent cases decompose the types into subproblems. The decomposition is given in Fig. ??. 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.

The left existential case first tries to unify the constraint. It's possible that the constraint cannot be solved, but also isn't invalid, in which case unifying the existential type's constraint simply updates the subtyping environment with the unsolved relational constraint. With the updated environment, the procedure then unifies the existential type's payload with the right type. It must do a safety check to ensure that the left existential's variables are merely assumed and not witnessed. If the constraint is solved then the procedure checks that the payload subtypes the right type for every constraint solution.

The right universal case updates the subtyping environment with the universal's variable constraint and unifies the left type with the payload. As with left existential, the procedure checks that the universal's type variable is merely assumed and not instantiated.

The left induction rule first tries to factor the inductive type and unify the weakened factored type with the right type. For example, a relational type between a natural number and a list, can be factored into a cross product of a natural number type and a list type. If factoring fails, then the procedure leverages the induction principle and substitutes the right type in for the inductive variable of the left type and unifies the new left type with the right type. Note that factoring is employed for the special case where the right type has variables which cannot be unified using induction due to our circularity restriction.

The cases for antecedent union, consequent intersection, left union, and right intersection decompose into sub-problems in the same way as the declarative subtyping semantics.

The remaining cases are continued in Fig. ??. The right existential case simply unifies the left type with the existential's payload and then unifies the existential's constraint, such that unifying the payload may bear witness to a type that leads to verifying the constraint.

TODO: consider using both forward and backtracking in unification and subtyping rules. Since it is not certain which constraint is stronger (in contrast to prolog). Union solutions together

The right existential case resembles the unification and backtracking approach in Prolog, where unification of the payload corresponds to unification of a query with the head of a horn clause, and solving the existential's constraint after the payload unification corresponds to solving the horn clause's body after the head's unification.

The left universal case also unifies in a backtracking fashion analogous to Prolog. The procedure unifies the universal's payload with the right type, possibly instantiating the universal's type variable, which may then be used to unify and check the universal's constraint.

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.

The remaining cases closely mirror their counterparts in the declarative subtyping rules.

5 CORRECTNESS

TODO: introduce model typing and soundness properties

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

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

Definition 8.2. $\boxed{\tau <: \tau \dashv Z}$ (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 8.3. $\boxed{Q \dashv M, \Delta}$

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

Definition 8.4. (Collection)

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

Definition 8.5. $\boxed{C \sqcup C = C}$

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

Definition 8.6. $\boxed{C \sqcap C = C}$

$$\begin{aligned} C \sqcap \epsilon &= \epsilon \\ C \sqcap (C'\ c) &= \begin{cases} (C \sqcap C')\ c & \text{if } c \in C \\ (C \sqcap C') & \text{otherwise} \end{cases} \end{aligned}$$

Definition 8.7. $\boxed{\overline{c}^{\vec{t}[P]} \triangleq C}$ Comprehension

$$\frac{\forall}{\overline{c}^{A[P]} \triangleq C}$$

Definition 8.8. $\boxed{\mathrm{map}(C, t) = C}$

$$\begin{array}{rcll} \mathrm{map}(\epsilon, t) & = & \epsilon & \triangleright \textit{ empty} \\ \mathrm{map}(C\ c, t) & = & \mathrm{map}(C, t) \sqcup t(c) & \triangleright \textit{ step} \end{array}$$

Definition 8.9. $\boxed{\overline{c}^{c \in C} \triangleq C}$ Map sugar

$$\overline{c'}^{c \in C} \quad \triangleq \quad \mathrm{map}(C, \lambda c.c')$$

Definition 8.10. $\boxed{c \in C}$

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

Definition 8.11. $\boxed{C \preceq C}$

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

Definition 8.12. $\boxed{|\,(\mathrm{T}) = \tau}$

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

Definition 8.13. $\boxed{\&(\mathrm{T}) = \tau}$

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

Definition 8.14. $\boxed{M, \Delta \vdash m <:^\star \alpha}$ Lower skolem subtyping

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

Definition 8.15. $\boxed{M, \Delta \vdash \Delta <:^\star \alpha/\tau}$ Universal lower skolem subtyping

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

Definition 8.16. $\boxed{M, \Delta \vdash \alpha/\tau <:^\star \Delta}$ Universal upper skolem subtyping

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

Definition 8.17. $\boxed{M, \Delta \vdash \tau <:^\dagger \alpha}$ Lower transitive subtyping

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

Definition 8.18. $\boxed{M, \Delta \vdash \mathrm{T} <:^{\dagger} \alpha}$ 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 8.19. $\boxed{M, \Delta \vdash \alpha <:^{\dagger} \tau}$ Upper transitive subtyping

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

Definition 8.20. $\boxed{M, \Delta \vdash \alpha <:^{\dagger} \mathrm{T}}$ 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 8.21. $\boxed{M, \Delta \vdash \alpha \dagger \tau <: \tau}$ Relational subtyping

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

Definition 8.22. $\boxed{M, \Delta \vdash \alpha/\tau \dagger \Delta}$ 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 8.23. $\boxed{M, \Delta, \mathrm{A} \vdash \Omega}$

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

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

Definition 8.24. $\boxed{N, M \vdash \Pi \equiv \mathrm{T}}$

$$\frac{}{N, M \vdash \epsilon \equiv \epsilon} \qquad \frac{\begin{array}{c} N, M \vdash \Pi \equiv \mathrm{T} \qquad \mathrm{FTV}(\tau_l) = A_l \qquad \mathrm{FTV}(\tau_r) = A_r \\ \Delta \vdash N \sqcup M \sqcup A_l \sqcup A_r \Uparrow \Delta' \qquad N, M, \Delta' \vdash \tau_l \text{->} \tau_r \equiv^{+} \tau \end{array}}{N, M \vdash \Pi\ \langle M, \Delta, \tau_l \text{->} \tau_r \rangle \equiv \mathrm{T}\ \tau}$$

Definition 8.25. $\boxed{N \vdash \alpha \cdot Z \cdot \alpha \text{->} \alpha \coloneqq \alpha \cdot \mathrm{T}}$ (Fixpoint duality)

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

$$\frac{}{N \vdash \alpha_{h^+} \cdot \epsilon \cdot \alpha_l \text{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \epsilon} \qquad \frac{\begin{array}{c} N \vdash \alpha_{h^+} \cdot Z \cdot \alpha_l \text{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T} \\ \Delta \vdash \alpha_{h^+} <: \mathrm{T}_h \qquad \Delta \vdash \alpha_l <: \mathrm{T}_l \qquad \Delta \vdash \mathrm{T}_r <: \alpha_r \\ \overline{\tau_l * \tau_r <: \alpha_{h^-}}^{\tau_l \text{->} \tau_r \in \mathrm{T}_h} = \Delta_h \\ N \sqcup M \sqcup \mathrm{FTV}(\mathrm{T}_l) \sqcup \mathrm{FTV}(\mathrm{T}_r)\ \alpha_{h^-} = \mathrm{A} \qquad \Delta \vdash \mathrm{A} \Uparrow \Delta_i \\ N\ \alpha_{h^-}, M, \Delta_i \sqcup \Delta_h \vdash \&(\mathrm{T}_l) * \lvert (\mathrm{T}_r) \equiv^{-} \tau \end{array}}{N \vdash \alpha_{h^+} \cdot Z\ \langle M, \Delta \rangle \cdot \alpha_l \text{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}\ \tau}$$

Definition 8.26. $\boxed{\Delta \vdash \mathrm{A} \Uparrow \Delta}$ Influential Filter

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

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

Definition 8.27. $\boxed{\Delta \vdash \mathrm{T} <: \alpha}$

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

Definition 8.28. $\boxed{\Delta \vdash \alpha <: \mathrm{T}}$

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

Definition 8.29. $\boxed{\mathrm{outer}(+|-) = \texttt{ALL}|\texttt{EXI}}$

$$\begin{aligned} \mathrm{outer}(+) &= \texttt{EXI} \quad \triangleright \textit{ positive} \\ \mathrm{outer}(-) &= \texttt{ALL} \quad \triangleright \textit{ negative} \end{aligned}$$

Definition 8.30. $\boxed{\mathrm{inner}(+|-) = \texttt{ALL}|\texttt{EXI}}$

$$\begin{aligned} \mathrm{inner}(+) &= \texttt{ALL} \quad \triangleright \textit{ positive} \\ \mathrm{inner}(-) &= \texttt{EXI} \quad \triangleright \textit{ negative} \end{aligned}$$

Definition 8.31. $\boxed{\mathrm{quantify}^{+|-}(\mathrm{A}, \Delta, \mathrm{A}, \Delta, \tau) = \tau}$

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

Definition 8.32. $\boxed{\mathrm{A}, \mathrm{A}, \Delta \vdash \tau \equiv^{+|-} \tau}$

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

Definition 8.33. $\boxed{\mathrm{A}, \Delta \vdash \Delta \wr \Delta}$

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

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

Definition 8.34. $\boxed{\Vdash e}$

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

Theorem 8.1. (Typing Soundness)

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

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

Theorem 8.2. (Proof typing consistency)

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

TODO: ...

Theorem 8.3. (Proof typing soundness)

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

TODO: ...

Theorem 8.4. (Proof typing weak soundness)

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

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** $e = @ \quad \tau = @$
. . **let** Ω by definition
. . $\Omega, \Gamma \vDash @ : @$ by definition
. . $\Omega, \Gamma \vDash e : \tau$ by substitution
. . $\exists \Omega.\ \Omega, \Gamma \vDash e : \tau$ by witness
. **case** $e = x \quad x : \tau \in \Gamma$
. **wrt** x
. . **let** Ω by definition
. . $\Omega, \Gamma \vDash x : \tau$ by definition
. . $\Omega, \Gamma \vDash e : \tau$ by substitution
. . $\exists \Omega.\ \Omega, \Gamma \vDash e : \tau$ by witness

. **case** $\Gamma \vdash e' : \tau' \dashv Z \quad \tau = \texttt{<}l\texttt{>}\tau' \quad e = \texttt{<}l\texttt{>}e'$
. **hypo** $\Gamma \vdash e' : \tau' \dashv Z \implies \Omega, \Gamma \vDash e' : \tau'$
. **wrt** $e'\ \tau'$
. . **let** Ω by definition
. . $\Omega, \Gamma \vDash e' : \tau'$ by application
. . $\Omega, \Gamma \vDash \texttt{<}l\texttt{>}e' : \texttt{<}l\texttt{>}\tau'$ by definition
. . $\Omega, \Gamma \vDash e : \tau$ by substitution
. . $\exists\Omega.\ \Omega, \Gamma \vDash e : \tau$ by witness
. **TODO: remaining trivial introduction cases**
. **case**
. **hypo**
. **wrt**
. **case** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \quad Z_0 \nrightarrow Z_1 \quad \tau_0 <: l\texttt{->}\alpha \dashv Z_1 \quad e = e_0\texttt{.}l \quad \tau = \alpha \quad Z = Z_1$
. **hypo** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \implies \Omega, \Gamma \vDash e_0 : \tau_0$
. **wrt** $\Omega\ e'\ l\ \alpha\ \tau_0\ Z_0\ Z_1$
. . $\Omega, \Gamma \vDash e_0 : \tau_0$ by application
. . **let** $M\ \Delta$ **s.t.** $\tau_0 <: l\texttt{->}\alpha \dashv M, \Delta$ by theorem 8.18
. . $\Omega \vDash \tau_0 <: l\texttt{->}\alpha$ by theorem 8.23
. . $\Omega, \Gamma \vDash e_0 : l\texttt{->}\alpha$ by theorem 8.19
. . $\Omega, \Gamma \vDash e_0\texttt{.}l : \alpha$ by theorem 8.20
. . $\Omega, \Gamma \vDash e : \tau$ by substitution
. . $\exists\Omega.\ \Omega, \Gamma \vDash e : \tau$ by witness
. **case** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \quad Z_0 \nrightarrow Z_1 \quad \Gamma \vdash e_1 : \tau_1 \dashv Z_1 \quad e = e_0\texttt{(}e_1\texttt{)} \quad \tau = \alpha \quad Z = Z_2$
. $Z_1 \nrightarrow Z_2 \quad \tau_0 <: \tau_1\texttt{->}\alpha \dashv Z_2$
. **hypo** $\Gamma \vdash e_0 : \tau_0 \dashv Z_0 \implies \Omega, \Gamma \vDash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \dashv Z_1 \implies \Omega, \Gamma \vDash e_1 : \tau_1$
. **wrt** $e_0\ e_1\ \alpha\ \tau_0\ \tau_1\ Z_0\ Z_1\ Z_2$
. . $\Omega, \Gamma \vDash e_0 : \tau_0$ by application
. . $\Omega, \Gamma \vDash e_1 : \tau_1$ by application
. . **let** $M\ \Delta$ **s.t.** $\tau_0 <: \tau_1\texttt{->}\alpha \dashv \langle M, \Delta \rangle$ by theorem 8.18
. . $\Omega, \Gamma \vDash \tau_0 <: \tau_1\texttt{->}\alpha$ by theorem 8.23
. . $\Omega \vDash e_0 : \tau_1\texttt{->}\alpha$ by theorem 8.19
. . $\Omega, \Gamma \vDash e_0\texttt{(}e_1\texttt{)} : \alpha$ by theorem 8.39
. . $\Omega, \Gamma \vDash e : \tau$ by substitution
. . $\exists\Omega.\ \Omega, \Gamma \vDash e : \tau$ by substitution
. **case** $e = \texttt{loop(}e'\texttt{)}$
. | $\tau = \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l'\texttt{*}\alpha_r'\texttt{<:LFLFP[}\alpha_{h^-}\texttt{] |(T)]}\alpha_r$
. | $Z = Z_0$
. | $\Gamma \vdash e' : \alpha_{h^+}\texttt{->}\tau' \dashv Z_0 \quad Z_0 \nrightarrow Z_1 \quad \tau' <: \alpha_l\texttt{->}\alpha_r \dashv Z_1$
. | $\mathrm{FTV}(\Gamma) \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l\texttt{->}\alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}$
. **hypo** $\forall\Omega.\ \Omega \vDash Z_0 \implies \Omega, \Gamma \vDash e' : \alpha_{h^+}\texttt{->}\tau'$
. **wrt** $e'\ \tau'\ \alpha_{h^+}\ \alpha_l\ \alpha_r\ \alpha_{h^-}\ \alpha_l'\ \alpha_r'\ \mathrm{T}\ Z_0\ Z_1$
. . **for** Ω **assume** $\Omega \vDash Z$
. . . $\Omega \vDash Z_0$ by substitution
. . . $\Omega, \Gamma \vDash e' : \alpha_{h^+}\texttt{->}\tau'$ by instantiation and application
. . . $\Omega \vDash \tau' <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l'\texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l'\texttt{*}\alpha_r'\texttt{<:LFP[}\alpha_{h^-}\texttt{] |(T)]}\alpha_r$ by theorem 8.5
. . . $\Omega \vDash \tau' <: \tau$ by substitution
. . . $\Omega \vDash e' : \alpha_{h^+}\texttt{->}\tau$ by theorem 8.19
. . . $\Omega \vDash \texttt{loop(}e'\texttt{)} : \tau$ by theorem 8.16

. . . $\Omega \vDash e : \tau$ by substitution
. . $\forall \Omega.\ \Omega \vDash Z \implies \Omega \vDash e : \tau$ by implication and generalization
. $\Omega, \Gamma \vDash e : \tau$ by induction
□

Theorem 8.5. (Fixpoint duality soundness (new))

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

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)

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

Proof:
assume $Z_0 \rightsquigarrow Z_1 \quad N \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l \texttt{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}$
. **induct on** $N \vdash \alpha_{h^+} \cdot Z_1 \cdot \alpha_l \texttt{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}$
. **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
. . . $\tau <: \alpha_l \texttt{->} \alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l' \texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l' \texttt{*} \alpha_r' \texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \texttt{|(T)]}\alpha_r' \dashv Z_0$ by implication
. . $\forall \tau.\ \tau <: \alpha_l \texttt{->} \alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l' \texttt{->EXI[}\alpha_r'\texttt{.}\alpha_l' \texttt{*} \alpha_r' \texttt{<:LFP[}\alpha_{h^-}\texttt{]}\ \texttt{|(T)]}\alpha_r' \dashv Z_0$ by generalization
. **case** $Z_1 = Z\ \langle M, \Delta \rangle \quad \mathrm{T} = \mathrm{T}_i\ \tau_i$
. | $N \vdash \alpha_{h^+} \cdot Z \cdot \alpha_l \texttt{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}_i$
. | $M, \Delta, \Delta \vdash \alpha_{h^+} <: \mathrm{T}_h \quad M, \Delta, \Delta \vdash \alpha_l <: \mathrm{T}_l \quad M, \Delta, \Delta \vdash \mathrm{T}_r <: \alpha_r$
. | $\overline{\tau_l \texttt{*} \tau_r \texttt{<:} \alpha_{h^-}}^{\tau_l\ \tau_r.\ \tau_l \texttt{->} \tau_r \in \mathrm{T}_h} = \Delta_h$
. | $N \sqcup M \sqcup \mathrm{FTV}(\mathrm{T}_l) \sqcup \mathrm{FTV}(\mathrm{T}_r)\ \alpha_{h^-} = \mathrm{A} \quad \Delta \vdash \mathrm{A} \Uparrow \Delta_i$
. | $N\ \alpha_{h^-}, M, \Delta_i \sqcup \Delta_h \vdash \&(\mathrm{T}_l) \texttt{*} \texttt{|}(\mathrm{T}_r) \equiv^- \tau_i$
. **hypo** $N \vdash \alpha_{h^+} \cdot Z \cdot \alpha_l \texttt{->} \alpha_r \coloneqq \alpha_{h^-} \cdot \mathrm{T}_i \implies$
$\forall \tau.\ \tau <: \alpha_l \texttt{->} \alpha_r \dashv Z \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l' \texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \texttt{|(T}_i\texttt{)]}\alpha_r' \dashv Z_0$
. **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
. . . $\tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l' \texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \texttt{|(T}_i\texttt{)]}\alpha_r' \dashv Z_0$ by instantiation and application
. . . $\tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l' \texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \texttt{|(T}_i\ \tau_i\texttt{)]}\alpha_r' \dashv Z_0$ by theorem 8.7
. . $\forall \tau.\ \tau <: \alpha_l \texttt{->} \alpha_r \dashv Z_1 \implies \tau <: \texttt{ALL[}\alpha_l'\texttt{]}\alpha_l' \texttt{->EXI[}\alpha_r'\texttt{.(}\alpha_l'\texttt{,}\alpha_r'\texttt{)<:LFP[}\alpha_{h^-}\texttt{]}\ \texttt{|(T)]}\alpha_r' \dashv Z_0$ by implication and generalization

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

Theorem 8.7. (Universe proof typing fixpoint extension)

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

Proof:
TODO: ...
□

Theorem 8.8. (Universe proof typing case soundness)

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

Proof:
TODO: ...
□

Theorem 8.9. (Universe proof typing fixpoint union)

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

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

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

Proof:
TODO: ...
□

Theorem 8.11. (Universe proof typing implication expansion)

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

Proof:
TODO: $P \implies Q$ **is equivalent to** $\neg(P \wedge \neg Q)$
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** $\neg(\exists x \in X \wedge \nexists y \in Y)$
□

Theorem 8.12. (Upper bound interpretation sound)

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

Proof:
TODO: ...
□

Theorem 8.13. (Lower bound interpretation sound)

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

Proof:
TODO: ...
□

Theorem 8.14. (Universe proof typing worldly)

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

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)

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

Proof:
TODO: ...
□

Theorem 8.17. (Proof subtyping decomposition)

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

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)

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

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

Theorem 8.20. (Model typing record elimination)

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

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

. **hypo** $\Omega, \Gamma \sqcup \Gamma' \vDash e' : l\text{->}\tau \implies \Omega, \Gamma \sqcup \Gamma' \vDash e'.l : \tau$
. **wrt** $\Sigma\ e'\ \Sigma'\ \Gamma'$
. . $\Omega, \Gamma \sqcup \Gamma' \vDash e'.l : \tau$ by application
. . $e[\Sigma].l \rightsquigarrow e'[\Sigma \sqcup \Sigma'].l$ by definition
. . $e[\Sigma].l = e.l[\Sigma]$ by definition
. . $e'[\Sigma \sqcup \Sigma'].l = e'.l[\Sigma \sqcup \Sigma']$ by definition
. . $e.l[\Sigma] \rightsquigarrow e'.l[\Sigma \sqcup \Sigma']$ by substitution
. . $\Omega, \Gamma \vDash e.l : \tau$ by definition
. $\Omega, \Gamma \vDash e.l : \tau$ by induction
□

Theorem 8.21. (Proof subtyping consistency)

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

Theorem 8.22. (Proof subtyping soundness)

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

Theorem 8.23. (Proof subtyping weak soundness)

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

TODO: skolems 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** Ω **assume** $\Omega \vDash \Delta$
. . . $\Omega \vDash \tau <: \tau$ by theorem 8.37
. . . $\Omega \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_l = \texttt{BOT}$
. . **for** Ω **assume** $\Omega \vDash \Delta$
. . . $\Omega \vDash \texttt{BOT} <: \tau_r$ by definition
. . . $\Omega \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_r = \texttt{TOP}$
. . **for** Ω **assume** $\Omega \vDash \Delta$
. . . $\Omega \vDash \tau_l <: \texttt{TOP}$ by definition
. . . $\Omega \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_r = l\text{->}(\tau_{rl} \& \tau_{rr}) \quad \tau_l <: (l\text{->}\tau_{rl}) \& (l\text{->}\tau_{rr}) \dashv M, \Delta$
. **hypo** $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: (l\text{->}\tau_{rl}) \& (l\text{->}\tau_{rr})$
. **wrt** $l\ \tau_{rl}\ \tau_{rr}$
. . **for** Ω **assume** $\Omega \vDash \Delta$
. . . $\Omega \vDash \tau_l <: (l\text{->}\tau_{rl}) \& (l\text{->}\tau_{rr})$ by application

. . . **for** $e\ \Gamma$ **assume** $\Omega, \Gamma \vDash e : \tau_l$
. . . . $\Omega, \Gamma \vDash e : (l\text{->}\tau_{rl})\&(l\text{->}\tau_{rr})$ by theorem 8.38
. . . . $\Omega, \Gamma \vDash e : l\text{->}\tau_{rl}$ by theorem 8.36
. . . . $\Omega, \Gamma \vDash e : l\text{->}\tau_{rr}$ by theorem 8.36
. . . . $\Omega, \Gamma \vDash e.l : \tau_{rl}$ by theorem 8.20
. . . . $\Omega, \Gamma \vDash e.l : \tau_{rr}$ by theorem 8.20
. . . . $\Omega, \Gamma \vDash e.l : \tau_{rl}\&\tau_{rr}$ by definition
. . . . $\Omega, \Gamma \vDash e : l\text{->}(\tau_{rl}\&\tau_{rr})$ by theorem 8.34
. . . . $\Omega, \Gamma \vDash e : \tau_r$ by substitution
. . . $\forall e\ \Gamma.\ \Omega, \Gamma \vDash e : \tau_l \implies \Omega, \Gamma \vDash e : \tau_r$ by implication and generalization
. . . $\Omega \vDash \tau_l <: \tau_r$ by definition
. . $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: \tau_r$ by implication and generalization
. **case** $\tau_l = \tau_{ll} \,|\, \tau_{lr} \quad M = M_1 \quad \Delta = \Delta_1$
. | $\tau_{ll} <: \tau_r \dashv M_0, \Delta_0 \quad M_0 \preceq M_1 \quad \Delta_0 \preceq \Delta_1 \quad \tau_{lr} <: \tau_r \dashv M_1, \Delta_1$
. **hypo** $\forall \Omega.\ \Omega \vDash \Delta_0 \implies \Omega \vDash \tau_{ll} <: \tau_r \quad \forall \Omega.\ \Omega \vDash \Delta_1 \implies \Omega \vDash \tau_{lr} <: \tau_r$
. **wrt** $\tau_{ll}\ \tau_{lr}$
. . **for** Ω **assume** $M, \Omega \vDash \Delta$
. . . $\Omega \vDash \Delta_1$ by substitution
. . . $\Omega \vDash \tau_{lr} <: \tau_r$ by application
. . . $\Omega \vDash \Delta_0$ by theorem **??** **TODO: ...**
. . . $\Omega \vDash \tau_{ll} <: \tau_r$ by application
. . . $\Omega \vDash \tau_{ll} \,|\, \tau_{lr} <: \tau_r$ by definition
. . . $\Omega \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: \tau_r$ 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** $\forall \Omega.\ \Omega \vDash \Delta_0 \implies \Omega \vDash \tau_l <: \tau_{rl} \quad \forall \Omega.\ \Omega \vDash \Delta_1 \implies \Omega \vDash \tau_l <: \tau_{rr}$
. **wrt**
. . **for** Ω **assume** $\Omega \vDash \Delta$
. . . $\Omega \vDash \Delta_1$ by substitution
. . . $\Omega \vDash \tau_l <: \tau_{rr}$ by instantiation and application
. . . $\Omega \vDash \Delta_0$ by theorem **??** **TODO: ...**
. . . $\Omega \vDash \tau_l <: \tau_{rl}$ by instantiation and application
. . . $\Omega \vDash \tau_l <: \tau_{rl}\&\tau_{rr}$ by definition
. . . $\Omega \vDash \tau_l <: \tau_r$ by substitution
. . $\forall \Omega.\ \Omega \vDash \Delta \implies \Omega \vDash \tau_l <: \tau_r$ by implication and generalization
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case**
. **hypo**
. **wrt**
. . **TODO: ...**
. **case** $\tau_l = \texttt{EXI[A Q]}\tau_b \quad M = M_1 \quad \Delta = \Delta_1$
. | $Q \dashv M_0, \Delta_0 \quad A \# \tau_r \quad M_0 \sqcup A \preceq M_1 \quad \Delta_0 \preceq \Delta_1 \quad \tau_b <: \tau_r \dashv M_1, \Delta_1$
. **hypo** $\forall \Omega.\ \Omega \vDash \Delta_1 \implies \Omega \vDash M_1, \tau_b <: \tau_r$

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

Theorem 8.24. Model typing existential elimination

$$\frac{\Omega \vDash e : \texttt{EXI[A}\ Q\texttt{]}\tau_l \qquad \Omega \vDash Q \qquad \Omega \vDash \tau_l <: \tau_r \qquad \mathrm{A} \# \tau_r}{\Omega \vDash e : \tau_r}$$

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

Theorem 8.25. Model typing universal introduction

$$\frac{\Omega \vDash e : \tau_l \qquad \Omega \vDash Q \qquad \Omega \vDash \tau_l <: \tau_r \qquad \mathrm{A} \# \tau_l}{\Omega \vDash e : \texttt{ALL[A}\ Q\texttt{]}\tau_r}$$

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

Theorem 8.26. Model subtyping sequence last

$$\frac{\Omega \vDash \Delta \ (\tau_l <: \tau_r)}{\Omega \vDash \tau_l <: \tau_r}$$

Proof:
TODO: ...

Theorem 8.27. Model subtyping sequence reduction

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

Proof:
TODO: ...

Theorem 8.28. Model subtyping sequence prefix

$$\frac{\Omega \vDash \Delta' \qquad \Delta \preceq \Delta'}{\Omega \vDash \Delta}$$

Proof:
TODO: ...

Theorem 8.29. Model subtyping sequence uncat

$$\frac{\Omega \vDash \Delta \sqcup \Delta'}{\Omega \vDash \Delta}$$

Proof:
TODO: ...

Theorem 8.30. concatenation prefix

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

Proof:
TODO: ...

Theorem 8.31. Model subtyping unsub left

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

Proof:
TODO: ...

Theorem 8.32. Model subtyping unsub right

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

Proof:
TODO: ...

Theorem 8.33. Model subtyping something

$$\frac{\Omega \vDash \Delta \qquad M, \Delta \vdash \mathrm{T} <:^{\star} \alpha}{\alpha /\!| \, (\mathrm{T}) \in \Omega}$$

Proof:
TODO: ...

Theorem 8.34. Model typing record introduction

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

Proof:
TODO: ...

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

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

Proof:
TODO: ...

Theorem 8.36. Model typing intersection elimination

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

Proof:
TODO: ...

Theorem 8.37. Model typing reflexivity

$$\frac{}{\Omega \vDash \tau <: \tau}$$

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

Theorem 8.38. (Model typing subsumption)

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

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

Theorem 8.39. (Model typing implication elimination)

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

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

Theorem 8.40. Model typing reduced implication elimination

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

assume $\Vdash e_1\,[\Sigma]$
. **let** Σ **s.t.** $\Omega, \Sigma \vDash \Gamma$ by theorem 8.50

. $\Vdash e_1[\Sigma]$ by theorem 8.51
. **induct on** $\Vdash e_1[\Sigma]$
. **case** $e_1[\Sigma] = v_1$
. **wrt** v_1
. . $\Omega, \Gamma \vDash (F\$p{=}{>}e)(e_1) : \tau_r$ by theorem 8.41
. **case** $e_1[\Sigma] \rightsquigarrow e_1' \quad \Vdash e_1'$
. **hypo** $\Vdash e_1' \implies \Omega, \Gamma \vDash (F\$p{=}{>}e)(e_1') : \tau_r$
. **wrt** e_1'
. . $\Omega, \Gamma \vDash (F\$p{=}{>}e)(e_1') : \tau_r$ by application
. . $(F\$p{=}{>}e)[\Sigma](e_1[\Sigma]) \rightsquigarrow (F\$p{=}{>}e)[\Sigma](e_1'[\Sigma])$ by definition
. . $\forall x.\ x \notin \mathbf{FV}(e_1')$ by theorem 8.48
. . $e_1' = e_1'[\Sigma]$ by by theorem 8.49
. . $(F\$p{=}{>}e)[\Sigma](e_1[\Sigma]) \rightsquigarrow (F\$p{=}{>}e)[\Sigma](e_1'[\Sigma])$ by substitution
. . $((F\$p{=}{>}e)(e_1))[\Sigma] \rightsquigarrow ((F\$p{=}{>}e)(e_1'))[\Sigma]$ by definition
. . $\Sigma \sqcup \epsilon = \Sigma$ by definition
. . $((F\$p{=}{>}e)(e_1))[\Sigma] \rightsquigarrow ((F\$p{=}{>}e)(e_1'))[\Sigma \sqcup \epsilon]$ by substitution
. . $\Gamma \sqcup \epsilon = \Gamma$ by definition
. . $\Omega, \epsilon \vDash \epsilon$ by definition
. . $\Omega, \Gamma \sqcup \epsilon \vDash (F\$p{=}{>}e)(e_1')$ by substitution
. . $\Omega, \Gamma \vDash (F\$p{=}{>}e)(e_1) : \tau_r$ by definition
. $\Omega, \Gamma \vDash (F\$p{=}{>}e)(e_1) : \tau_r$ by induction
□ by implication

Theorem 8.41. Model typing fully reduced implication elimination

$$\frac{\begin{array}{c}\Omega, \Gamma \vDash (F\$p{=}{>}e) : \tau_l \text{->} \tau_r \\ F = \epsilon \vee \Omega, \Gamma \vDash F(e_1) : \tau_r \qquad \Omega \vDash e_1 : \tau_l \qquad \Omega, \Sigma \vDash \Gamma \qquad e_1[\Sigma] = v_1\end{array}}{\Omega, \Gamma \vDash (F\$p{=}{>}e)(e_1) : \tau_r}$$

Proof:
assume $F = \epsilon \vee \Omega, \Gamma \vDash F : \tau_l \text{->} \tau_r$ **TODO: add more assumptions**
. $\Omega \vDash e_1[\Sigma] : \tau_l$ by theorem 8.45
. $\Omega \vDash v_1 : \tau_l$ by substitution
. **invert on** $F = \epsilon \vee \Omega, \Gamma \vDash F : \tau_l \text{->} \tau_r$
. **case** $F = \epsilon$
. . $\Omega, \Gamma \vDash F\$p{=}{>}e : \tau_l \text{->} \tau_r$ by theorem **??**
. . $\Omega, \Gamma \vDash \$p{=}{>}e : \tau_l \text{->} \tau_r$ by substitution
. . **let** Σ' **s.t.** $p \equiv v_1 \dashv \Sigma'$ by theorem 8.46
. . **for** e'
. . . $\neg\epsilon[\Sigma](v_1) \rightsquigarrow e'$ by definition
. . . $\neg F[\Sigma](v_1) \rightsquigarrow e'$ by substitution
. . $\forall e'.\ \neg F[\Sigma](v_1) \rightsquigarrow e'$ by generalization
. . $(F[\Sigma]\$p{=}{>}e[\Sigma \backslash \mathbf{FV}(p)])(v_1) \rightsquigarrow e[\Sigma \backslash \mathbf{FV}(p)][\Sigma]$ by definition
. . $\forall x.\ x \in \mathbf{FV}(p) \iff x \in \mathbf{dom}(\Sigma')$ by theorem 8.42
. . $\Sigma \backslash \mathbf{FV}(p) = \Sigma \backslash \mathbf{dom}(\Sigma')$ by theorem 8.43
. . $e[\Sigma \backslash \mathbf{dom}(\Sigma')][\Sigma'] = e[\Sigma \sqcup \Sigma']$ by theorem 8.44
. . $(F[\Sigma]\$p{=}{>}e[\Sigma \backslash \mathbf{FV}(p)])(v_1) \rightsquigarrow e[\Sigma \backslash \mathbf{dom}(\Sigma')][\Sigma]$ by substitution
. . $(F[\Sigma]\$p{=}{>}e[\Sigma \backslash \mathbf{FV}(p)])(v_1) \rightsquigarrow e[\Sigma \sqcup \Sigma']$ by substitution
. . $(F\$p{=}{>}e)[\Sigma](v_1) \rightsquigarrow e[\Sigma \sqcup \Sigma']$ by definition

. . $(F\$p{=>}e)[\Sigma](e_1[\Sigma]) \rightsquigarrow e[\Sigma \sqcup \Sigma']$ by substitution
. . $(F\$p{=>}e)[\Sigma](e_1[\Sigma]) = ((F\$p{=>}e)(e_1))[\Sigma]$ by definition
. . $((F\$p{=>}e)(e_1))[\Sigma] \rightsquigarrow e[\Sigma \sqcup \Sigma']$ by substitution
. . $\Omega, \Gamma \vDash (F\$p{=>}e)(e_1) : \tau_r$ by definition
. **case** $\Omega, \Gamma \vDash F(e_1) : \tau_r$
. . **let** e' **s.t.** $(F(e_1))[\Sigma] \rightsquigarrow e' \wedge \Omega, \Gamma \vDash e' : \tau_r$ by theorem **??**
. . $((F\$p{=>}e)(e_1))[\Sigma] \rightsquigarrow e'$ by definition
. . $\Omega, \Gamma \vDash (F\$p{=>}e)(e_1) : \tau_r$ by definition
. $\Omega, \Gamma \vDash (F\$p{=>}e)(e_1) : \tau_r$ by inversion
□ by implication

Theorem 8.42. (Pattern matching consistency)

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

Proof:
TODO: ...
□

Theorem 8.43. (Consistency diffing)

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

Proof:
TODO: ...
□

Theorem 8.44. (Concatenation Substitution)

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

Proof:
TODO: ...
□

Theorem 8.45. (Model typing valuation)

$$\frac{\Omega, \Gamma \vDash e : \tau \qquad \Omega, \Sigma \vDash \Gamma}{\Omega \vDash e[\Sigma] : \tau}$$

Proof:
assume $\Omega, \Gamma \vDash e : \tau \quad \Omega, \Sigma \vDash \Gamma$
TODO: ...
□

Theorem 8.46. (Model typing pattern matching)

$$\frac{\Omega, \Gamma \vDash \$p{=>}e : \tau_l \text{->} \tau_r \qquad \Omega \vDash v : \tau_l}{\exists \Sigma.\ p \equiv v \dashv \Sigma}$$

Proof:
assume $\Omega, \Gamma \vDash \$p{=>}e : \tau_l \text{->} \tau_r \quad \Omega \vDash v : \tau_l$
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)

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

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

Theorem 8.50. (Model typing assignability)

$$\frac{\Omega, \Gamma \vDash e : \tau}{\exists \Sigma.\ \Omega, \Sigma \vDash \Gamma}$$

Proof:
TODO: ...

Theorem 8.51. (Model typing soundness)

$$\frac{\Omega, \Gamma \vDash e : \tau}{\forall \Sigma.\ \Omega, \Sigma \vDash \Gamma \implies \Vdash e[\Sigma]}$$

Proof:
TODO: redo using universal/implication
assume $\Omega, \Sigma \vDash \Gamma \quad \Omega, \Gamma \vDash e : \tau$

. **case** $e = @$
. . **let** v **s.t.** $@ = v$
. . $e[\Sigma] = v$
. . $\Vdash e[\Sigma]$
. **case** $\Omega, \Gamma \vdash e' : \tau' \quad e = <l>e' \quad \tau = <l>\tau'$
. . $\Vdash e'$ by induction hypothesis
. . **case** $e'[\Sigma] = v$
. . . **let** v' **s.t.** $<l>v = v'$
. . . $<l>e'[\Sigma] = v'$
. . . $(<l>e')[\Sigma] = v'$
. . . $e[\Sigma] = v'$
. . . $\Vdash e[\Sigma]$
. . **case** $e'[\Sigma] \rightsquigarrow e'' \quad \Vdash e''$
. . . $<l>e'[\Sigma] \rightsquigarrow <l>e''$
. . . $\Vdash <l>e''$
. . . $\Vdash <l>e'[\Sigma]$
. . . $\Vdash (<l>e')[\Sigma]$
. . . $\Vdash e[\Sigma]$
. . $\Vdash e[\Sigma]$ by cases on $\Vdash e'$
. **TODO: remaining introduction cases**
. **case** $x : \tau \in \Gamma \quad x/v \in \Sigma \quad e = x$
. . $x[\Sigma] = v$
. . $e[\Sigma] = v$
. . $\Vdash e[\Sigma]$
. **case** $e[\Sigma] \rightsquigarrow e' \quad \Omega, \Gamma \vdash e' : \tau$
. . $\Vdash e'[\Sigma]$ by induction hypothesis
. . $\Vdash e[\Sigma]$
. $\Vdash e[\Sigma]$ by induction on $\Omega, \Gamma \vdash e : \tau$

□

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 $\Omega, \Gamma \vdash e : \tau$ 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 8.35. $\boxed{\Omega, \Gamma \vDash e : \tau}$ Model typing

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

$$\frac{\texttt{\$}l\texttt{=>}v \in G \qquad \Omega, \Gamma \vDash v : \tau \qquad \forall v'.\ \texttt{\$}l\texttt{=>}v' \in G \implies v' = v}{\Omega, \Gamma \vDash G : l\texttt{->}\tau}$$

$$\frac{\forall e.\ \Omega, \Gamma \vDash e : \tau_l \implies \Omega, \Gamma \vDash e : \tau_p \wedge (\forall \tau_n\ \tau.\ \Omega, \Gamma \vDash F : \tau_n\texttt{->}\tau \implies \neg(\Omega, \Gamma \vDash e : \tau_n)) \qquad \Omega, \Gamma \sqcup \Gamma' \vDash p \Diamond \tau_p \qquad \Omega, \Gamma \sqcup \Gamma' \vDash e : \tau_r}{\Omega, \Gamma \vDash F\texttt{*}p\texttt{=>}e : \tau_l\texttt{->}\tau_r}$$

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

$$\frac{\Omega, \Gamma \vDash e : \tau_l \qquad \Omega, \Gamma \vDash e : \tau_r}{\Omega, \Gamma \vDash e : \tau_l \& \tau_r} \qquad \frac{\Omega, \Gamma \vDash e : \tau_l \qquad \neg\ (\Omega, \Gamma \vDash e : \tau_r)}{\Omega, \Gamma \vDash e : \tau_l \backslash \tau_r}$$

$$\frac{\Omega \sqcup \Omega' \vDash Q \qquad \Omega \sqcup \Omega' \vDash e : \tau}{\Omega \vDash e : \texttt{EXI[A}\ Q\texttt{]}\tau} \qquad \frac{\forall \Omega'.\ \Omega \sqcup \Omega' \vDash Q \implies \Omega \sqcup \Omega', \Gamma \vDash e : \tau}{\Omega, \Gamma \vDash e : \texttt{ALL[A}\ Q\texttt{]}\tau}$$

$$\frac{\Omega\ \alpha/\texttt{LFP[}\alpha\texttt{]}\tau \vDash e : \tau}{\Omega, \Gamma \vDash e : \texttt{LFP[}\alpha\texttt{]}\tau} \qquad \frac{x : \tau \in \Gamma}{\Omega, \Gamma \vDash x : \tau}$$

$$\frac{\Omega, \Sigma \vDash \Gamma \qquad e[\Sigma] \rightsquigarrow e'[\Sigma'] \qquad \Omega, \Sigma' \vDash \Gamma' \qquad \Omega, \Gamma' \vDash e' : \tau}{\Omega, \Gamma \vDash e : \tau}$$

Definition 8.36. $\boxed{\Omega, \Sigma \vDash \Gamma}$

$$\frac{}{\Omega, \Sigma \vDash \epsilon} \qquad \frac{\Omega, \Sigma \vDash \Gamma \qquad \Omega \vDash v : \tau}{\Omega, \Sigma\ x/v \vDash \Gamma\ x : \tau}$$

Definition 8.37. $\boxed{\Omega \vDash \tau <: \tau}$

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

Definition 8.38. $\boxed{\Omega \vDash Q}$

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