Extrinsic Relational Type Inference

ACM Reference Format:
. 2024. Extrinsic Relational Type Inference. 1, 1 (March 2024), 13 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.

For various reasons that aren't completely clear, intrinsically typed languages have lost favor, and untyped or *extrinsically typed* languages, such as Javascript/Typescript and Python, have surged 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 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.

Author's address:

XXXX-XXXX/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 these limitations, 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

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)

We illustrate the syntax and semantics of programs and types with the example program shown in Fig. 1.

Path typing. Consider the function `trivial` that completes an English phrase:

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

$$\Delta \cdot \Gamma \vdash \texttt{trivial} : \texttt{(?hello -> ?world) \& (?good -> ?morning)}$$

Path selection. Suppose the function `trivial` is applied to a literal value `#hello`. The system can discard the irrelevant clauses and infer the singleton type `?world`.

$$\Delta \cdot \Gamma \vdash \texttt{trivial \#hello} : \texttt{?world}$$

Relational typing. Consider the function `repeat` that takes a natural number and returns a list of that length. Without specifying any requirements besides the function definition, the system

```
let trivial =
    path #hello => #world
    path #good => #morning in

let repeat = path x => fix(path self =>
    path #zero => #nil
    path #succ n => #cons (x, self n)) in

let fromList = fix(path self =>
    path #nil => ...
    path #cons (x, xs) => ...) in

let fromNat = path x n => fromList (repeat x n)

let fromUno = path (@uno = content) => ... in
let fromDos = path (@dos = content) => ... in
let fromBoth = path x => (fromUno x, fromDos, x)

let lessOrEq = fix(path self =>
    path (#zero, _) => #true
    path (#succ x, #succ y) => self (x, y)
    path (#succ _, #zero) => #false) in

let max = (path (x, y) =>
    if lessOrEq (x, y) then y else x) in

...
```

Fig. 1. Example program

can infer the property that the resulting list has the length of the input number.

$$\frac{\begin{array}{r} nat = \texttt{induc[N] ?zero | ?succ N} \\ nat\_list\ \alpha = \left(\begin{array}{l}\texttt{induc[NL]} \\ \quad \texttt{?zero} * \texttt{?nil} \mid \\ \quad \{\texttt{?succ N} * \texttt{?cons (}\alpha * \texttt{L) with N} * \texttt{L <: NL}\}\end{array}\right)\end{array}}{\Delta \cdot \Gamma \vdash \texttt{repeat} : \texttt{[X]X -> [F<:\{N -> L with N} * \texttt{L <:}\ nat\_list\ \texttt{X\}]F}}$$

Relational selection. Suppose the function `repeat` is applied to the hardcoded number two, represented as `(#succ #succ #zero)`. The system can infer the result to be a singleton type representing a single list, much like how Prolog evaluates logic programs.

$$\Delta \cdot \Gamma \vdash \texttt{repeat () (\#succ \#succ \#zero)} : \texttt{?cons (unit} * \texttt{?cons (unit} * \texttt{?nil))}$$

Path selection and relational selection demonstrate that the declarative type language is expressive enough to perform evaluation. However, this simply reproduces the effect of the dynamic semantics, albeit in a declarative style. For types to be useful in practice, they need to offer ways to express properties with incomplete information. The next examples illustrate how the system can compose abstract properties to infer useful properties, which are not reproducible by dynamic semantics.

Factoring. Suppose the function `fromList`, which expects a list, is applied to a list that's related to a natural number, as would be the result of `(repeat x n)`, illustrated in the body of `fromNat`. The system must verify that the type of `(repeat x n)` is a subtype of the parameter type of `(fromList)`. The argument type of `(repeat x n)` is a type projected from a relation with an abstract natural number, since its arguments are not hardcoded. This abstract type information cannot be handled by dynamic semantics. Despite these complexities, the the system is able to ensure that these abstract inferred types can safely be composed, by factoring $(nat\_list\ \alpha)$ into the weaker pair $(nat\ *\ list\ \alpha)$. Once factored out, the list type of the argument type's relation can be projected and unified with the parameter's list type.

$$\frac{list\ \alpha = \texttt{induc[L] ?nil | ?cons } (\alpha\ *\ \texttt{L}) \qquad \Delta \vdash nat\_list\ \texttt{X} \sqsubseteq nat\ *\ list\ \texttt{X} \qquad \Delta \vdash list\ \texttt{X} \sqsubseteq list\ \texttt{Y}}{\Delta \vdash \{\texttt{L with N} * \texttt{L <:}\ nat\_list\ \texttt{X}\} \sqsubseteq list\ \texttt{Y}}$$

Inductive subtyping. Consider applying a function that expects a natural number to an argument whose type is an even natural number. In order to verify that this application is allowed, the system must verify that an even number is a subtype of a natural number. The system can soundly verify this subtyping by relying on an induction hypothesis. The induction hypothesis allows weakening the inductive component of the even type to the natural number type as it unrolls.

$$\frac{\dfrac{even = \texttt{induc[N] ?zero | ?succ ?succ N} \qquad \Delta \vdash \texttt{(?zero | ?succ ?succ } nat\texttt{)} \sqsubseteq nat}{\Delta, even\ \texttt{<:}\ nat \vdash \texttt{(?zero | ?succ ?succ } even\texttt{)} \sqsubseteq nat}}{\Delta \vdash even \sqsubseteq nat}$$

Relational subtyping. Due to the precise relational types that the system infers, it may also be necessary to verify that a relation subtypes another relation, such as a list with an even length subtyping an list with a natural number length. This situation is similar to inductive subtyping of simple types, but it is complicated by relational constraints in the inductive relations, which must

be added to the context.

$$\frac{\begin{array}{l} even\_list\ \alpha = \left(\begin{array}{l}\texttt{induc[EL]}\\ \quad \texttt{?zero * ?nil |}\\ \quad \texttt{\{?succ ?succ E * ?cons ?cons (}\alpha\ \texttt{* L) with E * L <: EL\}}\end{array}\right) \\ \Delta, \texttt{E * L <:}\ nat\_list\ \texttt{unit} \vdash \left(\begin{array}{l}\texttt{?zero * ?nil |}\\ \texttt{?succ ?succ E *}\\ \quad \texttt{?cons (unit * ?cons (unit * L))}\end{array}\right) \sqsubseteq nat\_list \end{array}}{\left(\begin{array}{l}\Delta, even\_list\ \texttt{unit} <: nat\_list\ \texttt{unit},\\ \texttt{E * L <:}\ even\_list\ \texttt{unit}\end{array}\right) \vdash \left(\begin{array}{l}\texttt{?zero * ?nil |}\\ \texttt{?succ ?succ E *}\\ \quad \texttt{?cons (unit * ?cons (unit * L))}\end{array}\right) \sqsubseteq nat\_list}$$

$$\Delta \vdash (even\_list\ \texttt{unit}) \sqsubseteq (nat\_list\ \texttt{unit})$$

Refinement. Consider the function `fromBoth` that calls two functions on some variable of unknown value or type. The same variable is used as an argument to two separate functions that have different parameter types. The type of the variable can be refined by intersecting both parameter types that it must satisfy.

$$\Delta \cdot \Gamma \vdash \texttt{fromBoth} : \texttt{(uno : X) \& (dos : Y) -> ...}$$

Path sensitivity. Consider the function `max` that chooses the maximum of two natural numbers. The function must satisfy the property that the result is greater or equal to each of the inputs. The system can infer this property by relying on multiple type inference mechanisms, including relational typing, path selection, refinement, and a special form of refinement that refines a type by specializing relations it belongs to. The *if-then-else* expression is merely sugar for applying a function with a *true* path and *false* path to a boolean expression. Note that the variables are used in both the condition and the bodies of the if-then-else expression. To infer a precise type for the if-then-else expression, the types of the variables need to be refined according to each path's expected type, but without leaking the refinement outside of that path. That is, the refinements must be local or path sensitive. Moreover, there must be enough paths of the applied function to handle all the possible values of the argument.

TODO: make sure inductive type (LED) is explained clearly

TODO: max type inference is wrong; should be an intersection

$$\frac{\begin{array}{l} leq\_decide = \left(\begin{array}{l}\texttt{induc[LED]}\\ \quad \texttt{\{?zero * \_ * ?true\} |}\\ \quad \texttt{\{?succ X * ?succ Y * B with X * Y * B <: LED\} |}\\ \quad \texttt{\{?succ \_ * ?zero * ?false\}}\end{array}\right) \\ max\_spec = \left(\begin{array}{l}\texttt{X * Y -> Y \& \{Z with (X * Z * ?true) <:}\ leq\_decide\texttt{\} |}\\ \texttt{X * Y -> X \& \{Z with (Z * Y * ?false) <:}\ leq\_decide\texttt{\}}\end{array}\right) \end{array}}{\Delta \cdot \Gamma \vdash \texttt{max} : max\_spec}$$

TODO: more motivating and elucidating examples

3 LANGUAGE

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. 2. 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`

τ ::=	▷ *type*
α	▷ *variable*
`@`	▷ *unit*
`~`l τ	▷ *tag*
l`:`τ	▷ *field*
τ`->`τ	▷ *implication*
τ`|`τ	▷ *union*
τ`&`τ	▷ *intersection*
τ`\`τ	▷ *difference*
`EXI [`$\bar{\alpha}$ Q`]` τ	▷ *existential / indexed union*
`ALL [`α`<:`τ`]` τ	▷ *universal / indexed intersection*
`LFP` α τ	▷ *induction*
Q ::=	▷ *qualification*
`;`	▷ *empty*
Q `;` τ`<:`τ	▷ *extend subtyping*

Fig. 2. Syntax

3.1 Typing

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.

3.2 Subtyping

The subtyping is given in Fig. 3. 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.

4 ANALYSIS

The analysis consists of two main parts. The top level is type inference, which corresponds to typing and generates a type for an expression. When type inference encounters constraints that its types must adhere to, it calls unification to solve these constraints. Note that since the types are expressive enough to represent constraints, an alternative approach of generating constraints and solving them in separate stages could also be designed using the same structures. Additional structures for the analysis are given in Fig. ??. Inference generates a solution set T, which contains triples, each with a type variable set, a subtyping environment, and a type. Unification generates a solution set C, which contains subtyping environments.

4.1 Type inference

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.

4.2 Subtype unification

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 EXPERIMENTS

TODO: develop 12 tree/list experiments

6 RELATED WORK

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

In my opinion, better terms would be "model-based" vs "proof-based". A model-based system can be more complete but requires proving absence of inhabitation of the types. That is, the semantics of subtyping would depend on the semantics of inhabitation of types. This is related to SAT solving, proof synthesis, model checking, and SMT. If a type is model-based, not proof-based, then a proof must be found. Since the "proof" is not part of the model-based subtyping statement, there is an infinite space to search to prove that the subtyping statement holds. This leads to a more complete system, but a more difficult system to decide.

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

$\boxed{\langle \Delta, \Omega \rangle \rightsquigarrow \langle \Delta, \Omega \rangle \vdash \tau \sqsubseteq \tau}$

$$\textsc{ExtendStrongExis}\ \frac{\wedge_i(\alpha_i \notin FTV(\Delta_i)) \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_h, \Omega_h \rangle \vdash \tau_{qs} \sqsubseteq \tau_{qw} \qquad \langle \Delta_h, \Omega_h \cup \{\alpha_i\}_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash (\texttt{[|}\ Q\ \texttt{]}\ \tau_s) \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash (\texttt{[|}\ \overline{\alpha_i}^i\ Q\ \texttt{;}\ \tau_{qs}\texttt{<:}\tau_{qw}\ \texttt{]}\ \tau_s) \sqsubseteq \tau_w}$$

$$\textsc{EmptyStrongExis}\ \frac{\wedge_i(\alpha_i \notin FTV(\Delta_i)) \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_h, \Omega_h \rangle \vdash \tau_s \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash (\texttt{[|}\ \overline{\alpha_i}^i\ \texttt{.]}\ \tau_s) \sqsubseteq \tau_w}$$

$$\textsc{ExtendWeakExis}\ \frac{\wedge_i(\alpha_i \notin FTV(\Delta_i)) \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_h, \Omega_h \rangle \vdash \tau_s \sqsubseteq \tau_w \qquad \langle \Delta_h, \Omega_h \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_{qs} \sqsubseteq (\texttt{[|}\ Q\ \texttt{]}\ \tau_{qw})}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq (\texttt{[|}\ \overline{\alpha_i}^i\ Q\ \texttt{;}\ \tau_{qs}\texttt{<:}\tau_{qw}\ \texttt{]}\ \tau_w)}$$

$$\textsc{EmptyWeakExis}\ \frac{\wedge_i(\alpha_i \notin FTV(\Delta_i)) \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq (\texttt{[|}\ \overline{\alpha_i}^i\ \texttt{.]}\ \tau_w)}$$

$$\textsc{StrictStrongVar}\ \frac{\alpha \in \Omega_i \qquad condenseWeakest(\Delta, \alpha) = \tau_s \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \alpha \sqsubseteq \tau_w}$$

$$\textsc{LenientStrongVar}\ \frac{\alpha \notin \Omega_i \qquad extractStrongest(\Delta_i, \alpha) = \tau_s \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_h, \Omega_h \rangle \vdash \tau_s \sqsubseteq \tau_w \qquad \Delta_h \subseteq \Delta_o \qquad \Omega_h \subseteq \Omega_o \qquad \alpha\texttt{<:}\tau_w \in \Omega_o}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \alpha \sqsubseteq \tau_w}$$

$$\textsc{StrongUnion}\ \frac{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_h, \Omega_h \rangle \vdash \tau_1 \sqsubseteq \tau_w \qquad \langle \Delta_h, \Omega_h \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_2 \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_1 \,|\, \tau_2 \sqsubseteq \tau_w}$$

$$\textsc{LeftWeakUnion}\ \frac{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_1}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_1 \,|\, \tau_2}$$

$$\textsc{RightWeakUnion}\ \frac{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_2}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_1 \,|\, \tau_2}$$

$$\textsc{Equiv}\ \frac{\vdash \tau_s \equiv \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_w}$$

$$\textsc{FactorStrongInduc}\ \frac{factorLeast(\texttt{induc}\ \alpha\ \tau_s) = \tau_f \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_f \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash (\texttt{induc}\ \alpha\ \tau_s) \sqsubseteq \tau_w}$$

$$\textsc{KInducStrongInduc}\ \frac{\alpha \notin FTV(\Delta_i) \qquad subst(\{\alpha \mapsto \tau_w\}, \tau_s) = \tau_s' \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s' \sqsubseteq \tau_w}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash (\texttt{induc}\ \alpha\ \tau_s) \sqsubseteq \tau_w}$$

$$\textsc{UninhabWeakDiff}\ \frac{\neg inhabitable(\tau_s)}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_c \backslash \tau_n}$$

$$\textsc{InhabWeakDiff}\ \frac{inhabitable(\tau_s) \qquad \langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_c}{\langle \Delta_i, \Omega_i \rangle \rightsquigarrow \langle \Delta_o, \Omega_o \rangle \vdash \tau_s \sqsubseteq \tau_c \backslash \tau_n}$$