MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types*

LIONEL PARREAUX, HKUST, Hong Kong, China
CHUN YIN CHAU, HKUST, Hong Kong, China

Intersection and union types are becoming more popular by the day, entering the mainstream in programming languages like TypeScript and Scala 3. Yet, no language so far has managed to combine these powerful types with principal polymorphic type inference. We present a solution to this problem in MLstruct, a language with subtyped records, equirecursive types, first-class unions and intersections, class-instance matching, and ML-style principal type inference. While MLstruct is mostly structurally typed, it contains a healthy sprinkle of nominality for classes, which gives it desirable semantics, enabling the expression of a powerful form of extensible variants that does not need row variables. Technically, we define the constructs of our language using conjunction, disjunction, and negation connectives, making sure they form a Boolean algebra, and we show that the addition of a few nonstandard subtyping rules gives us enough structure to derive a sound and complete type inference algorithm. With this work, we hope to foster the development of better type inference for present and future programming languages with expressive subtyping systems.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Polymorphism*.

Additional Key Words and Phrases: principal type inference, union and intersection types, structural typing

ACM Reference Format:
Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (October 2022), 30 pages. https://doi.org/10.1145/3563304

1 INTRODUCTION

Programming languages with ML-style type inference have traditionally avoided subtyping because of the complexities it brings over a simple unification-based treatment of type constraints. But Dolan and Mycroft [2017] recently showed with MLsub that an *algebraic* account of subtyping resolved many of these difficulties and enabled the inference of precise types that more accurately reflect the flow of expressions in programs. Unfortunately, among other limitations, MLsub does not support union and intersection types, which are emerging as important building blocks in the design of structurally-typed programming languages like TypeScript, Flow, Scala 3, and others.

We close this gap with MLstruct, showing that MLsub-style type inference can be generalized to include well-behaved forms of union and intersection types as well as pattern matching on single-inheritance class hierarchies. As a first example, consider the following definitions:

```
class Some[A]: { value: A }
class None: {}
```

```
def flatMap f opt = case opt of
  Some → f opt.value,
  None → None{}
```

*This is version 6.1 of the paper; get the latest extended version at https://lptk.github.io/mlstruct-paper.

Authors' addresses: Lionel Parreaux, parreaux@ust.hk, HKUST, Hong Kong, China; Chun Yin Chau, cychauab@connect.ust.hk, HKUST, Hong Kong, China.

2475-1421/2022/10-ART141
https://doi.org/10.1145/3563304

The type inferred by our system for `flatMap` is:

$$\text{flatMap}: \ \forall \alpha, \beta.\ (\alpha \to \beta) \to (\text{Some}[\alpha] \vee \text{None}) \to (\beta \vee \text{None})$$

Interestingly, this is more general than the traditional type given to `flatMap` for `Option` types. Indeed, our `flatMap` does not require the function passed in argument to return either a `None` or a `Some` value, but allows it to return anything it wants (any β), which gets merged with the `None` value returned by the other branch (yielding type $\beta \vee \text{None}$). For example,

```
let res = flatMap (fun x → x) (Some{value = 42})
```

is given type $42 \vee \text{None}$[1] because the function may return either `42` or `None`. A value of this type can later be inspected with an instance match expression of the form:

```
case res of Int → res, None → 0
```

which is inferred to be of type $42 \vee 0$, a subtype of Nat. This is not the most general version of `flatMap` either. We can also make the function open-ended, accepting either a `Some` value or *anything else*, instead of just `Some` or `None`, by using a default case (denoted by the underscore '_'):

```
def flatMap2 f opt = case opt of Some → f opt.value, _ → opt
```

This `flatMap2` version has the following type inferred, where $\vee$ and $\wedge$ have the usual precedence:

$$\text{flatMap2}: \ \forall \alpha, \beta.\ (\alpha \to \beta) \to (\text{Some}[\alpha] \vee \beta \wedge \neg\, \#\text{Some}) \to \beta$$

This type demonstrates a central aspect of our approach: the use of *negation types* (also called *complement* types), written $\neg\tau$, which allows us to find principal type solutions in tricky typing situations. Here, type #Some is the *nominal tag* of class Some. A nominal tag represents the *identity* of a class, disregarding the values of its fields and type parameters: if a value v has type #Some, this means v is an instance of Some, while if v has type $\neg$#Some, this means it is not. To showcase different usages of this definition, consider the following calls along with their inferred types:[2]

```
ex1 =  flatMap2 (fun x → x + 1) 42                          :  Int
ex2 =  flatMap2 (fun x → Some{value = x}) (Some{value = 12}) :  Some[12]
ex3 =  flatMap2 (fun x → Some{value = x}) 42                 :  Some[⊥] ∨ 42
```

It is easy to see that instantiating β to Int and Some[12] respectively allows `ex1` and `ex2` to type check. In `ex3`, both types Some[γ] and 42 flow into the result, for some type inference variable γ, but γ is never constrained and only occurs positively so it can be simplified, yielding Some[$\bot$] $\vee$ 42. We can convert `ex3` to `42` through a **`case`** expression using the `impossible` helper function:[3]

```
def impossible x = case x of {}                    :  ⊥ → ⊥
case ex3 of Int → ex3, Some → impossible ex3.value :  42
```

One may naively think that the following type could fit `flatMap2` as well:

$$\text{flatMap2\_wrong}: \ \forall \alpha, \beta, \gamma.\ (\alpha \to \beta) \to (\text{Some}[\alpha] \vee \gamma) \to (\beta \vee \gamma)$$

but this type does not work. To see why, consider what happens if we instantiate the type variables to $\alpha = \text{Int}$, $\beta = \text{Int}$, and $\gamma = \text{Some}[\text{Bool}]$. This yields the type:

$$\text{flatMap2\_wrong}': \ (\text{Int} \to \text{Int}) \to (\text{Some}[\text{Int}] \vee \text{Some}[\text{Bool}]) \to (\text{Int} \vee \text{Some}[\text{Bool}])$$

which would allow the call `flatMap2 (fun x → x + 1) (Some{value = false})` because Some[Bool] $\leqslant$ Some[Int] $\vee$ Some[Bool]. This expression, however, would crash with a runtime type mismatch! Indeed, the shape of the `Some` argument matches the first branch of `flatMap2`'s **`case`** expression,

[1] MLstruct supports singleton types for constant literals, e.g., 42 is both a value and a type, with $42 : 42 \leqslant \text{Nat} \leqslant \text{Int}$.
[2] Notice that only `ex3` features a union of two distinct type constructors 'Some[$\bot$] $\vee$ 42' because in `ex1` and `ex2` only one concrete type constructor statically flows into the result of the expression (42 and Some, respectively).
[3] One may expect Some[$\bot$] $\equiv \bot$, but this does not hold in MLstruct, as it would prevent effective principal type inference.

and therefore `false` is passed to our argument function, which tries to add `1` to it as though it was an integer... So we *do* need the negation that appears in the correct type of `flatMap2`, as it prevents passing in arguments that are also of the `Some` shape, but with the wrong type arguments.

Finally, let us push the generality of our function further yet, to demonstrate the flexibility of the system. Consider this last twist on `flatMap` for optional values, which we will call `mapSome`:

```
def mapSome f opt = case opt of Some → f opt, _ → opt
```

The difference with the previous function is that this one does not unwrap the `Some` value received in argument, but simply passes it unchanged to its function argument. Its inferred type is:

$$\text{mapSome} : \forall \alpha, \beta.\ (\alpha \to \beta) \to (\alpha \wedge \#\text{Some} \vee \beta \wedge \neg \#\text{Some}) \to \beta$$

This type shows that it does not matter what specific subtype of Some we have in the first branch: as long as the argument has type α ***when it is*** a Some instance, then α is the type the argument function should take, without loss of generality. This demonstrates that our type system can tease apart different flows of values based on the nominal identities of individual matched classes.

As an example of the additional flexibility afforded by this new function, consider the following:

```
class SomeAnd[A, P]: Some[A] ∧ { payload: P }
let arg = if ⟨arbitrary condition⟩ then SomeAnd{value = 42, payload = 23}
                                  else None{}
in mapSome (fun x → x.value + x.payload) arg
```

of inferred type Int ∨ None. Here, we define a new subclass of `Some` containing an additional `payload` field, and we use this class instead of `Some`, allowing the `payload` field to be used from within the function argument we pass to `mapSome`. This is not expressible in OCaml polymorphic variants [Garrigue 2001] and related systems [Ohori 1995]. More powerful systems with row variables [Pottier 2003; Rémy 1994] would still fail here because of their use of *unification*: `mapSome` merges its `opt` parameter with the result, so these systems would yield a unification error at the `mapSome` call site, because the argument function returns an integer instead of a value of the same type as the input: subtyping makes MLstruct more flexible than existing systems based on row variable.

MLscript is a new programming language developed at the Hong Kong University of Science and Technology[4] featuring first-class unions, intersections, negations, and ML-style type inference, among other features. For simplicity, this paper focuses on a *core subset* of MLscript referred to as ***MLstruct***, containing only the features relevant to principal type inference in a Boolean algebra of structural types, used in all examples above. An MLstruct implementation is provided as an artifact [Parreaux et al. 2022] and available at github.com/hkust-taco/mlstruct, with a web demonstration at hkust-taco.github.io/mlstruct. The specific contributions we make are the following:

- We present MLstruct (Section 2), which subsumes both the original ML type system and the newer MLsub [Dolan 2017], extending the latter with simple class hierarchies and class-instance matching based on union, intersection, and negation type connectives.
- We describe our approach to type inference based on the Boolean-algebraic properties of MLstruct's types (Section 3). To the best of our knowledge, MLstruct is the first language to support principal polymorphic type inference with union and intersection types. Moreover, it ***does not rely on backtracking*** and yields types that are amenable to simplification.
- We formalize the declarative semantics of MLstruct in the $\lambda^{\neg}$ calculus (Section 4), making sure to establish the Boolean-algebraic properties of its subtyping lattice (Section 4.4.3). We state the standard soundness properties of *progress* and *preservation*, whose complete proofs are given in the extended version of this paper [Parreaux and Chau 2022].

[4]The GitHub repository of the full MLscript language is available at https://github.com/hkust-taco/mlscript.

- We formally describe our type inference algorithm (Section 5). We state its *soundness* and *completeness* theorems. Again, the proofs can be found in the paper's extended version.

2 PRESENTATION OF MLSTRUCT

MLstruct *subsumes* Dolan's MLsub, the previous state of the art in type inference with subtyping, which itself subsumes traditional ML typing: all ML terms are typeable in MLsub and all MLsub terms are typeable in MLstruct. We now present the features and subtyping discipline of MLstruct.

2.1 Overview of MLscript Features

An MLstruct program is made of top-level statements followed by an expression, the program's body. A statements can be either a type declaration (class or type alias) or a top-level function definition, written `def f = t` or `rec def f = t` when `f` is recursive. MLstruct infers polymorphic types for `def` bindings, allowing them to be used at different type instantiations in the program.

2.1.1 Polymorphism. Polymorphic types include a set of type variables with bounds, such as $\forall(\alpha \leqslant \mathsf{Int}).\, \mathsf{List}[\alpha] \to \mathsf{List}[\alpha]$. The bounds of polymorphic types are allowed to be cyclic, which can be interpreted as indirectly describing recursive types. For example, $\forall(\alpha \leqslant \top \to \alpha).\, \alpha$ is the principal type scheme of `rec def f = fun a → f` which accepts any argument and returns itself.

2.1.2 Classes, inheritance, and type aliases. Because object orientation is not the topic of this paper, which focuses on functional-style use cases, the basic OO constructs of MLstruct presented here are intentionally bare-bone. Classes are declared with the following syntax:

```
class C[A, B, ...]: D[S, T, ...] ∧ { x: X, y: Y, ... }
```

where `A`, `B`, etc. are type parameters, `S`, `T`, `X`, `Y`, etc. are arbitrary types and `D` is the parent class of `C`, which can be left out if the class has no parents. Along with a *type* constructor $\mathsf{C}[A, B, \ldots]$, such a declaration also introduces a *data* constructor C of type:

$$\mathsf{C} : \forall\, \beta_1, \beta_2, \ldots, (\alpha_1 \leqslant \tau_1), (\alpha_2 \leqslant \tau_2), \ldots.\, \{\, x_1 : \alpha_1,\, x_2 : \alpha_2,\, \ldots \,\} \to \mathsf{C}[\beta_1, \beta_2 \ldots] \wedge \{\, x_1 : \alpha_1,\, x_2 : \alpha_2,\, \ldots \,\}$$

where x_i are all the fields declared by $\mathsf{C}[\beta_1, \beta_2, \ldots]$ or by any of its ancestors in the inheritance hierarchy, and τ_i are the corresponding types – if a field is declared in several classes of the hierarchy, we take the intersection of all the declared types for that field. To retain as precise typing as possible, we let the types of the fields taken in parameters to be arbitrary *subtypes* α_i of the declared τ_i, so we can refine the result type $\mathsf{C}[\beta_1, \beta_2 \ldots] \wedge \{\, x_1 : \alpha_1,\, x_2 : \alpha_2,\, \ldots \,\}$ to retain these precise types. For instance, assuming `class C: { x: Int }`, term `C { x = 1 }` is given the precise type $\mathsf{C} \wedge \{\, x : 1 \,\}$.

Classes are restricted to single-inheritance hierarchies. Like in the work of Muehlboeck and Tate [2018], this has the nice property that it allows union types to be refined by reducing types like $(C_0 \vee \tau) \wedge C_1$ to $C_0 \wedge C_1 \vee \tau \wedge C_1$ by distributivity and to just $\tau \wedge C_1$ when C_0 and C_1 are unrelated ($C_0 \wedge C_1 \equiv \bot$). But MLstruct can easily be extended to support traits, which are not subject to this restriction, by slightly adapting the definition of type normal forms (our artifact [Parreaux et al. 2022] implements this). Thanks to their use of negation types (described in Section 4.3), the typing rules for pattern matching do not even have to change, and traits can also be pattern-matched.

2.1.3 Shadowing. Non-recursive `def`s use shadowing semantics,[5] so they can simulate the more traditional field initialization and overriding semantics of traditional class constructors. For instance:

```
class Person: {name: Str, age: Nat, isMajor: Bool}
def Person n a = Person{name = capitalize n, age = a, isMajor = a >= 18}
```

in which the `def`, of inferred type $\mathsf{Person}_1 : \forall(\alpha \leqslant \mathsf{Nat}).\, \mathsf{Str} \to \alpha \to \mathsf{Person} \wedge \{\, \mathsf{age} : \alpha \,\}$, *shadows* the bare constructor of the `Person` class (of type $\mathsf{Person}_0 : \forall(\alpha \leqslant \mathsf{Str}), (\beta \leqslant \mathsf{Nat}), (\gamma \leqslant$

[5] Type names, on the other hand, live in a different namespace and are not subject to shadowing.

Bool). $\{\,\mathsf{name} : \alpha,\ \mathsf{age} : \beta,\ \mathsf{isMajor} : \gamma\,\} \to \mathsf{Person} \wedge \{\,\mathsf{name} : \alpha,\ \mathsf{age} : \beta,\ \mathsf{isMajor} : \gamma\,\}$), forcing users of the class to go through it as the official `Person` constructor. Function `capitalize` returns a Str, so no 'name' refinement is needed ($\mathsf{Person} \wedge \{\,\mathsf{age} : \alpha,\ \mathsf{name} : \mathsf{Str}\,\} \equiv \mathsf{Person} \wedge \{\,\mathsf{age} : \alpha\,\}$).

2.1.4 Nominality. Classes are not equivalent to their bodies. Indeed, they include a notion of "nominal identity", which means that while a class type is a *subtype* of its body, it is not a *supertype* of it. So unlike TypeScript, it is not possible to use a record `{x = 1}` as an instance of a class declared as `class C: {x: Int}`. To obtain a `C`, one must use its constructor, as in `C{x = 1}`. This nominality property is a central part of our type system and is much demanded by users in practice.[6] It comes at no loss of generality, as type synonyms can be used if nominality is not wanted.

2.1.5 Type aliases. Arbitrary types can be given names using the syntax `type X[A, B, ...] = T`. Type aliases and classes can refer to each other freely and can be mutually recursive.

2.1.6 Guardedness check. Classes and type aliases are checked to ensure they do not inherit or refer to themselves immediately without going through a "concrete" type constructor first (i.e., a function or record type). For instance, the recursive occurrence of `A` in `type A[X] = Id[A[X]] ∨ Int` where `type Id[Y] = Y` is unguarded and thus illegal, but `type A[X] = { x: A[X] } ∨ Int` is fine.

2.1.7 Class-instance matching. As presented in the introduction, one can match values against class patterns in a form of primitive pattern matching. Consider the following definitions:

```
class Cons[A]: Some[A] ∧ { tail: List[A] }   type List[A] = Cons[A] ∨ None
rec def mapList f ls = case ls of
  Cons → Cons{value = f ls.value, tail = mapList f ls.tail},
  None → None{}
```

of inferred type:[7] $\mathsf{mapList} : \forall \alpha, \beta.\ (\alpha \to \beta) \to \gamma \to \delta$ **where** $\gamma \leqslant (\mathsf{Cons}[\alpha] \wedge \{\,\mathsf{tail} : \gamma\,\} \vee \mathsf{None})$, $\delta \geqslant (\mathsf{Cons}[\beta] \wedge \{\,\mathsf{tail} : \delta\,\} \vee \mathsf{None})$

We define a `List` type using `None` as the "nil" list and whose `Cons` constructor *extends* `Some` (from the introduction). A list in this encoding can be passed to any function that expects an option in input — if the list is a `Cons` instance, it is also a `Some` instance, and the `value` field representing the head of the list will be used as the `value` wrapped by the option. This example demonstrates that structural typing lets us mix and match as well as refine different constructors in a flexible way.

2.2 Constructing the Lattice of Types

The algebraic subtyping philosophy of type system design is to begin with the subtyping of data types (records, functions, etc.) and to define the order connectives to fit this subtyping order, rather than to follow set-theoretic intuitions. We follow this philosophy and aim to design our subtyping order to tackle the following design constraints:

(A) The order connectives $\wedge$, $\vee$, and $\neg$ should induce a Boolean algebra, so that we can manipulate types using well-known and intuitive Boolean-algebraic reasoning techniques.
(B) Nominal tags and their negations specifically should admit an intuitive set-theoretic understanding, in the sense that for any class C, type $\#C$ should denote all instances of C while type $\neg\#C$ should correspondingly denote all instances that are *not* derived from class C.[8]
(C) The resulting system should admit principal types as well as an *effective* polymorphic type inference strategy, where "effective" means that it should not rely on backtracking.

[6] The lack of nominal typing for classes has been a major pain point in TypeScript. The issue requesting it, created in 2014 and still not resolved, has accumulated more than 500 "thumbs up". See: https://github.com/Microsoft/Typescript/issues/202.
[7] The `where` keyword is used to visually separate the specification of type variable bounds, making them more readable.
[8] By contrast, we have no specific requirements on the meaning of negated function and record types, which are uninhabited.

2.2.1 Lattice types. *Top*, written $\top$, is the type of all values, a supertype of every other type. Its dual *bottom*, written $\bot$, is the type of no values, a subtype of every other type. For every τ, we have $\bot \leqslant \tau \leqslant \top$. Intersection $\wedge$ and union $\vee$ types are the respective *meet* and *join* operators in the subtyping lattice. It is worth discussing possible treatments one can give these connectives:

(1) We can axiomatize them as denoting the intersection $\cup$ and union $\cap$ of the sets of values that their operands denote, which is the approach taken by semantic subtyping.
(2) We can axiomatize them as *greatest lower bound* (GLB) and *least upper bound* (LUB) operators, usually written $\sqcap$ and $\sqcup$, whose meaning is given by following the structure of a preexisting lattice of simple types (types without order connectives). In this interpretation, we can *calculate* the results of these operators when their operands are concretely known.
(3) Finally, we can view $\wedge$ and $\vee$ as type constructors in their own right, with dedicated subtyping derivation rules. Then unions and intersections are not "computed away" but instead represent proper constructed types, which may or may not be equivalent to existing simple types.

2.2.2 Subtyping. We base our approach primarily on (3) but we do include a number of subtyping rules whose goal is to make the order connectives behave like (2) in some specific cases:

- We posit $\#C_1 \wedge \#C_2 \leqslant \bot$ whenever classes C_1 and C_2 are unrelated.[9] This makes sense because there are no values that can be instances of both classes at the same time, due to single inheritance. We obviously also have $\#C_1 \wedge \#C_2 \geqslant \bot$, meaning the two sides are equivalent (they subtype each other), which we write $\#C_1 \wedge \#C_2 \equiv \bot$. On the other hand, $\#C \leqslant \#D$ for all C, D where C inherits from D; so when $\#C_1$ and $\#C_2$ *are* related then either $\#C_1 \wedge \#C_2 \equiv \#C_1$ or $\#C_1 \wedge \#C_2 \equiv \#C_2$. Overall, we can always "reduce" intersections of nominal class tags to a single non-intersection type, making $\wedge$ behave like a GLB operator in the class inheritance sublattice, made of nominal tags, $\top$, $\bot$, and $\vee$, evocative of (2).
- We also posit the nonstandard rule $(\tau_1 \to \tau_2) \wedge (\tau_3 \to \tau_4) \leqslant (\tau_1 \vee \tau_3) \to (\tau_2 \wedge \tau_4)$. The other direction holds by function parameter contravariance and result covariance, so again the two sides are made equivalent. $\wedge$ behaves like a GLB operator on function types in a lattice which does not contain subtyping-based overloaded functions types, such as those of Dolan [2017]; Pottier [1998b]. This rule is illogical from the set-theoretic point of view: a function that can be viewed as returning a τ_2 when given a τ_1 and returning a τ_4 when given a τ_3 cannot be viewed as *always* returning a $\tau_2 \wedge \tau_4$. For instance, consider $\lambda x.\, x$, typeable both as $\mathsf{Int} \to \mathsf{Int}$ and as $\mathsf{Bool} \to \mathsf{Bool}$. According to both classical intersection type systems and the semantic subtyping interpretation, this term could be assigned type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$. But *we posited* that this type is equivalent to $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \wedge \mathsf{Bool})$. Thankfully, in MLstruct $\lambda x.\, x$ cannot be assigned such an intersection type; instead, its most general type is $\forall \alpha.\, \alpha \to \alpha$, which does subsume both $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$, but not $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$. This explains why intersection types cannot be used to encode overloading in MLstruct.[10]
- For record intersections, we have the standard rule that $\{ x : \tau \} \wedge \{ x : \pi \} \leqslant \{ x : \tau \wedge \pi \}$, making the two sides equivalent since the other direction holds by depth subtyping. Intersections of distinct record fields, on the other hand, do not reduce and stay as they are — in fact, multi-field record types are encoded, in MLstruct, as intersections of individual single-field record types, following Reynolds [1997]. For instance, assuming $x \neq y$, then $\{ x : \tau_1,\ y : \tau_2 \}$ is *not* a core form but merely *syntax sugar* for $\{ x : \tau_1 \} \wedge \{ y : \tau_2 \}$.
- We apply similar treatments to various forms of unions: First, $(\tau_1 \to \tau_2) \vee (\tau_3 \to \tau_4) \equiv (\tau_1 \wedge \tau_3) \to (\tau_2 \vee \tau_4)$, the dual of the function intersection treatment mentioned above.

[9] This class intersection annihilation rule is not novel; for example, Ceylon has a similar one [Muehlboeck and Tate 2018].
[10] Other forms of overloading, such as type classes and *constructor overloading* (see Section 6), are still possible.

Second, we recognize that $\{ x : \tau \} \vee \{ y : \pi \}$ and $\{ x : \tau \} \vee (\pi_1 \to \pi_2)$, where $x \neq y$, cannot be meaningfully used in a program, as the language has no feature allowing to tease these two components apart, so we identify these types with $\top$, the top type. This is done by adding $\top \leqslant \{ x : \tau \} \vee \{ y : \pi \}$ and $\top \leqslant \{ x : \tau \} \vee (\pi_1 \to \pi_2)$ as subtyping derivation rules.

The full specification of our subtyping theory is presented later, in Section 4 (Figure 4).

2.2.3 Soundness. The soundness of subtyping disciplines was traditionally studied by finding semantic models corresponding to types and subtyping, where types are typically understood as predicates on the denotations of λ terms (obtained from some λ model) and where subtyping is understood as inclusion between the corresponding sets of denotations. In this paper, we take a much more straightforward approach: all we require from the subtyping relation is that it be *consistent*, in the sense that it correctly relate types constructed from the same constructors and that it not relate unrelated type constructors. For instance, $\tau_1 \to \tau_2 \leqslant \pi_1 \to \pi_2$ should hold *if and only if* $\pi_1 \leqslant \tau_1$ and $\tau_2 \leqslant \pi_2$, and $\{ x : \mathsf{Int} \} \leqslant \#C$ should *not* be derivable. This turns out to be a sufficient condition for the usual soundness properties of *progress* and *preservation* to hold in our language. Consistency is more subtle than it may first appear. We cannot identify, e.g., $\#C \vee \{ x : \tau \}$ with $\top$ even though the components of this type cannot be teased apart through instance matching, as doing so is incompatible with distributivity. Notice the conjunctive normal form of $\pi = \#C \wedge \{ x : \tau \} \vee \#D \wedge \{ y : \tau' \}$ is $\pi \equiv (\#C \vee \#D) \wedge (\#C \vee \{ y : \tau' \}) \wedge (\{ x : \tau \} \vee \#D) \wedge (\{ x : \tau \} \vee \{ y : \tau' \})$. We can make $\{ x : \tau \} \vee \{ y : \tau' \}$ equivalent to $\top$ when $x \neq y$ because that still leaves $\pi \equiv (\#C \vee \#D) \wedge (\#C \vee \{ y : \tau' \}) \wedge (\{ x : \tau \} \vee \#D)$, which is equivalent to the original π by distributivity and simplification. But making $\#C \vee \{ y : \tau' \}$ and $\{ x : \tau \} \vee \#D$ equivalent to $\top$ would make $\pi \equiv \#C \vee \#D$, losing all information related to the fields, and breaking pattern matching!

2.2.4 Negation types. Finally, we can add Boolean-algebraic negation to our subtyping lattice. However, its interpretation is considerably constrained by the Boolean structure and by the rules already presented in Section 2.2.2. In some languages, the values of a negation type $\neg\tau$ are intuitively understood as *all* values that are *not* of the negated type τ, but in MLstruct, this intuition only holds for nominal tags.[11] Negations can express interesting patterns, such as safe division, as seen below, where '`e : T`' is used to ascribe a type `T` to an expression `e`:

```
def div n m = n / (m : Int ∧ ¬0)                    def f x = div x 2
div: Int → (Int ∧ ¬0) → Int                         f: Int → Int

def g (x: Int) = div 100 x    ⇐ error: found Int, expected Int ∧ ¬0

def div_opt n m = case m of 0 → None{}, _ → Some{value = div n m}
div_opt: Int → Int → None ∨ Some[Int]
```

Here, '`case m of ...`' is actually a shorthand for the core form '`case m = m of ...`' which shadows the outer `m` with a local variable `m` that is assigned a more refined type in each `case` branch.

As we saw in the introduction, $\neg$ also allows for the sound typing of class-instance matching with default cases. Moreover, together with $\top$, $\bot$, $\wedge$, and $\vee$, our type structure forms a Boolean lattice, whose algebraic properties are essential to enabling principal type inference (see Section 3.2.1).

2.2.5 Structural decomposition. We reduce complex object types to simpler elementary parts, which can be handled in a uniform way. Similarly to type aliases, which can always be replaced by their bodies, we can replace class types by their fields intersected with the corresponding nominal tags. For example, $\mathsf{Cons}[\tau]$ as defined in Section 2.1.7 reduces to $\#\mathsf{Cons} \wedge \{ \mathsf{value} : \tau, \mathsf{tail} : \mathsf{List}[\tau] \}$. Recall that class tags like #Cons represent the *nominal identities* of classes. They are related with other class tags by a subtyping relationship that follows the inheritance hierarchy. For instance,

[11] For other constructs, such as functions and records, negations assume a purely algebraic role. For instance, we have relationships like $\neg\{ x : \tau \} \leqslant \pi_1 \to \pi_2$ due to $\{ x : \tau \} \vee \pi_1 \to \pi_2$ being identified with $\top$ (see also Section 4.4.4).

given **class** $C[\alpha] : D[\alpha \vee 2] \wedge \{ x : 0 \vee \alpha \}$ and **class** $D[\beta] : \{ x : \beta,\ y : \mathsf{Int} \}$, then we have $\#C \leqslant \#D$. Moreover, the refined class type $C[1] \wedge \{ y : \mathsf{Nat} \}$ *reduces to* the equivalent $\#C \wedge \{ x : 0 \vee 1 \} \wedge \{ x : 1 \vee 2,\ y : \mathsf{Int} \} \wedge \{ y : \mathsf{Nat} \}$, which reduces further to $\#C \wedge \{ x : 1,\ y : \mathsf{Nat} \}$.

Decomposing class types into more elementary types makes MLstruct's approach fundamentally structural, while retaining the right amount of nominality to allow for runtime instance matching based on class identity. It also means that there is no primitive notion of nominal type constructor *variance* in MLstruct: the covariance and contravariance of type parameters simply arise from the way class and alias types desugar into basic structural components.

2.3 Limitations

MLstruct comes with some limitations. We already mentioned in Section 2.2.2 that intersections cannot be used to type overloading. Here we explain several other significant limitations.

2.3.1 Regular structural types. We restrict the shapes of MLstruct data types to be *regular trees* to make the problem of deciding whether one subsumes another decidable: concretely, occurrences of a class or alias type transitively reachable through the body of that type must have the same shape as the type's head declaration. For instance, the following are disallowed:

```
class C[A]: {x: C[Int]}   class C[A]: C[{x: List[A]}]   class C[A]: {x: C[C[A]]}
```

We conjecture that allowing such definitions would give our types the expressive power of context-free grammars, for which language inclusion is undecidable, making subtyping undecidable.[12] To replace illegal non-regular class fields, one can use either top-level functions or *methods*. The latter solve this problem by having their types known in advance and not participating in structural subtype checking. Methods are implemented in MLstruct but not presented in this paper.

2.3.2 Simplified treatment of unions. MLstruct keeps the expressiveness of unions in check by identifying $\{ x : \tau_1 \} \vee \{ y : \tau_2 \}$ $(x \neq y)$ and $\{ x : \tau_1 \} \vee (\tau_2 \to \tau_3)$ with $\top$, as described in Section 2.2.2. To make unions of different fields useful, one needs to "tag" the different cases with class types, as in $C_1 \wedge \{ x : \tau_1 \} \vee C_2 \wedge \{ y : \tau_2 \}$, allowing us to separately handle these cases through instance matching '**case** v **of** $C_1 \to \ldots\ v.x\ \ldots, C_2 \to \ldots\ v.y\ \ldots$', whereas this is not necessary in, e.g., TypeScript.

A direct consequence of this restriction is that in MLstruct, there is no difference between $\{ x : \mathsf{Int},\ y : \mathsf{Int} \} \vee \{ x : \mathsf{Str},\ y : \mathsf{Str} \}$ and $\{ x : \mathsf{Int} \vee \mathsf{Str},\ y : \mathsf{Int} \vee \mathsf{Str} \}$ (still assuming $x \neq y$). Indeed, remember that $\{ x : \tau_1,\ y : \tau_2 \}$ is syntax sugar for $\{ x : \tau_1 \} \wedge \{ y : \tau_2 \}$ and by distributivity of unions over intersections, we can take $\{ x : \mathsf{Int},\ y : \mathsf{Int} \} \vee \{ x : \mathsf{Str},\ y : \mathsf{Str} \}$ to:

$$(\{ x : \mathsf{Int} \} \vee \{ x : \mathsf{Str} \}) \wedge (\{ x : \mathsf{Int} \} \vee \{ y : \mathsf{Str} \}) \wedge (\{ y : \mathsf{Int} \} \vee \{ x : \mathsf{Str} \}) \wedge (\{ y : \mathsf{Int} \} \vee \{ y : \mathsf{Str} \})$$

and since $\{ x : \tau_1 \} \vee \{ y : \tau_2 \}$ is identified with $\top$, as explained in Section 2.2.2, this reduces to:

$$(\{ x : \mathsf{Int} \} \vee \{ x : \mathsf{Str} \}) \wedge (\{ y : \mathsf{Int} \} \vee \{ y : \mathsf{Str} \})$$

which reduces by field merging to $\{ x : \mathsf{Int} \vee \mathsf{Str} \} \wedge \{ y : \mathsf{Int} \vee \mathsf{Str} \}$, i.e., $\{ x : \mathsf{Int} \vee \mathsf{Str},\ y : \mathsf{Int} \vee \mathsf{Str} \}$.

Another consequence is that, e.g., $\mathsf{List}[\mathsf{Int}] \vee \mathsf{List}[\mathsf{Str}]$ is identified with $\mathsf{List}[\mathsf{Int} \vee \mathsf{Str}]$. Again, to distinguish between these two, one should prefer the use of class-tagged unions or, equivalently, proper sum types such as $\mathsf{Either}[\mathsf{List}[\mathsf{Int}], \mathsf{List}[\mathsf{Str}]]$, defined in terms of Left and Right classes.

2.3.3 Fewer relationships. Unlike in semantic subtyping approaches, but like in most practical programming languages, we do not have $\{x : \bot\} \leqslant \bot$. This would in fact lead to unsoundness in MLstruct: consider $\pi = (\{ x : \mathsf{Some}[\mathsf{Int}],\ y : \tau_1 \} \vee \{ x : \mathsf{None},\ y : \tau_2 \}) \wedge \{ x : \mathsf{None} \}$; we would have $\pi \equiv \{ x : \bot,\ y : \tau_1 \} \vee \{ x : \mathsf{None},\ y : \tau_2 \} \equiv \{ x : \mathsf{None},\ y : \tau_2 \}$ by distributivity and *also* $\pi \equiv \{ x : \bot \vee \mathsf{None},\ y : \tau_1 \vee \tau_2 \}$ by using (2.3.2) before distributing, but $\tau_1 \not\equiv \tau_1 \vee \tau_2$ in general.

[12] TypeScript does allow such definitions, meaning its type checker would necessarily be either unsound or incomplete.

3 INFERRING PRINCIPAL TYPES FOR MLSTRUCT

We now informally describe our general approach to principal type inference in MLstruct.

3.1 Basic Type Inference Idea

We base the core of our type inference algorithm on a simple formulation of MLsub type inference we formulated in previous work [Parreaux 2020]. The constraint solver attaches a set of *lower* and *upper* bounds to each type variable, and maintain the transitive closure of these constraints, i.e., it makes sure that at all times the union of all lower bounds of a variable remains a *subtype* of the intersection of all its upper bounds. This means that when registering a new constraint of the form $\alpha \leqslant \tau$, we not only have to add τ to the upper bounds of α, but also to constrain $\mathit{lowerBounds}(\alpha) \leqslant \tau$ in turn. One has to be particularly careful to maintain a "cache" of subtyping relationships currently being constrained, as the graphs formed by type variable bounds may contain cycles. Because types are regular, there is always a point, in a cyclic constraint, where we end up checking a constraint we are already in the process of checking (it is in the cache), in which case we can assume that the constraint holds and terminate. Constraints of the general form $\tau_1 \leqslant \tau_2$ are handled by *losslessly* decomposing them into smaller constraints, until we arrive at constraints on type variables, which is made possible by the algebraic subtyping rules. The losslessness of this approach is needed to ensure that we only infer *principal* types. In other words, when decomposing a constraint, we must produce a set of smaller constraints that is *equivalent* to the original constraint. For example, we can decompose the constraint $\tau_1 \vee (\tau_2 \to \tau_3) \leqslant \tau_4 \to \tau_5$ into the equivalent set of constraints: $\tau_1 \leqslant \tau_4 \to \tau_5$; $\tau_4 \leqslant \tau_2$; and $\tau_3 \leqslant \tau_5$. If we arrive at a constraint between two incompatible type constructors, such as $\tau_1 \to \tau_2 \leqslant \{ x : \tau_3 \}$, an error is reported.

3.2 Solving Constraints with Unions and Intersections

By contrast with MLsub, MLstruct supports union and intersections types in a *first-class* capacity, meaning that one can use these types in both positive *and* negative positions. This is particularly important to type check instance matching, which requires unions in negative positions, and class types, which require intersections in positive positions (both illegal in MLsub).

The main problem that arises in this setting is: ***How to resolve constraints with the shapes*** $\tau_1 \leqslant \tau_2 \vee \tau_3$ ***and*** $\tau_1 \wedge \tau_2 \leqslant \tau_3$ ***?*** Such constraints cannot be easily decomposed into simpler constraints without losing information — which would prevent us from achieving complete type inference — and without having to perform backtracking — which would quickly become intractable, even in non-pathological cases, and would yield a set of possible types instead of a single principal type. When faced with such constraints, we distinguish two cases: (1) there is a type variable among τ_1, τ_2, and τ_3; and (2) conversely, none of these types are type variables.

3.2.1 Negation types. We use negation types to reformulate constraints involving type variables into forms that allow us to make progress, relying on the Boolean-algebraic properties of negation. A constraint such as $\tau_1 \leqslant \tau_2 \vee \alpha$ can be rewritten to $\tau_1 \wedge \neg\tau_2 \leqslant \alpha$ by turning the "positive" τ_2 on the right into a "negative" on the left, as these are equivalent in a Boolean algebra.[13] Therefore, it is sufficient *and* necessary to constrain α to be a supertype of $\tau_1 \wedge \neg\tau_2$ to solve the constraint at hand. Similarly, we can solve $\alpha \wedge \tau_1 \leqslant \tau_2$ by constraining α to be a subtype of $\tau_2 \vee \neg\tau_1$.[14] When both

[13] Aiken and Wimmers [1993] used a similar trick, albeit in a more specific *set-theoretic* interpretation of unions/intersections.

[14] If it were not for pattern matching, we could avoid negation types by adopting a more complicated representation of type variable bounds that internalizes the same information. That is, instead of $\alpha \leqslant \tau$ and $\alpha \geqslant \tau$ for a given type variable α, we would have bounds of the form $\alpha \wedge \pi \leqslant \tau$ and $\alpha \vee \pi \geqslant \tau$, representing $\alpha \leqslant \tau \vee \neg\pi$ and $\alpha \geqslant \tau \wedge \neg\pi$ respectively. But reducing several upper/lower bounds into a single bound, which previously worked by simply intersecting/taking the union of them, would now be impossible without generalizing bounds further. Type simplification would also become difficult.

transformations are possible, one may pick one or the other equivalently. The correctness of these transformations is formally demonstrated in in the *Swapping* theorem of the extended version of this paper [Parreaux and Chau 2022]. This approach provides a solution to case (1), but in a way it only pushes the problem around, delaying the inevitable apparition of case (2).

3.2.2 Normalization of constraints. To solve problem (2), we *normalize* constraints until they are in the shape "$\tau_{\text{con}} \leqslant \tau_{\text{dis}}$", where (using a horizontal overline to denote 0 to n repetitions):

- τ_{con} represents $\top$, $\bot$, or the intersection of any non-empty subset of $\{\, \#C,\ \tau_1 \to \tau_2,\ \{\, \overline{x : \tau} \,\} \,\}$.
- τ_{dis} represents types of the form $\top$, $\bot$, $(\tau_1 \to \tau_2)\ \overline{\vee\ \#C}$, $\{\, x : \tau \,\}\ \overline{\vee\ \#C}$, or $\#C\ \overline{\vee\ \#C'}$.

Let us consider a few examples. First, given a constraint like $(\tau_1 \vee \tau_2) \wedge \tau_3 \leqslant \tau_4$, we can distribute the intersection over the union thanks to the rules of Boolean algebras (see Section 4.4.3), which results in $(\tau_1 \wedge \tau_3) \vee (\tau_2 \wedge \tau_3) \leqslant \tau_4$, allowing us to solve $\tau_1 \wedge \tau_3 \leqslant \tau_4$ and $\tau_2 \wedge \tau_3 \leqslant \tau_4$ independently. Second, given a constraint like $\tau_1 \leqslant \{\, x : \tau_2 \,\} \vee \tau_3 \to \tau_4$, we simply use the fact that $\{\, x : \tau_2 \,\} \vee \tau_3 \to \tau_4 \equiv \top$ (as explained in Section 2.2.2) to reduce the constraint to $\tau_1 \leqslant \top$, a tautology. Third, with constraints containing intersected nominal class tags on the left, we can compute their greatest lower bound based on our knowledge of the single-inheritance class hierarchy. We eventually end up with constraints of the shape "$\tau_{\text{con}} \leqslant \tau_{\text{dis}}$" and there always exists a $\tau_i \in \tau_{\text{con}}$ and $\tau'_j \in \tau_{\text{dis}}$ such that we can reduce the constraint to an *equivalent* constraint $\tau_i \leqslant \tau'_j$. Notice that if two related nominal tags appears on each side, it is always safe to pick that comparison, as doing so does not entail any additional constraints. If there are no such related nominal tags, the only other choice is to find a type in the right-hand side to match a corresponding type in the left-hand side, and the syntax of these normal forms prevents there being more than one possible choice. All in all, our Boolean algebra of types equipped with various algebraic simplification laws ensures that we have a lossless way of resolving the complex constraints that arise from union and intersection types, enabling principal type inference.

The constraint solving algorithm described in Section 5.3 and implemented in the artifact uses the ideas explored above but puts the entire constraint into a *normal form*, instead of normalizing constraints on the fly. This helps to efficiently guarantee termination by maintaining a cache of currently-processed subtyping relationships in normal forms, which is straightforward to query.

3.3 Subsumption Checking

Subsumption checking, denoted by $\leqslant^{\forall}$, is important to check that definitions conform to given signatures. Contrary to MLsub, which syntactically separates positive from negative types (the *polarity restriction*), and therefore requires different algorithms for constraint solving and subsumption checking, in MLstruct we can immediately *reuse* the constraint solving algorithm for subsumption checking, without requiring much changes to the type system. To implement $\forall \Xi_1.\ \tau_1 \leqslant^{\forall} \forall \Xi_2.\ \tau_2$, we instantiate all the type variables in Ξ_1, with their bounds, to fresh type variables, and we turn all the variables in Ξ_2 into *rigid* variables (so-called "skolems"). The latter can be done by turning these type variables into fresh *flexible* nominal tags and by inlining their bounds, expressing them in terms of unions, intersections, and recursive types. Since there is no polarity restrictions in our system, the resulting types can be compared directly using the normal constraint solving algorithm.

Flexible nominal tags $\#F$ are just like nominal class tags $\#C$, except that they can coexist with unrelated tags without reducing to $\bot$. For example, while $\#C_1 \wedge \#C_2$ is equivalent to $\bot$ in MLstruct when C_1 and C_2 are unrelated, $\#F \wedge \#C_2$ is not.[15] Flexible nominal tags are also the feature used to encode the nominal tags of *traits*, necessary to implement mixin traits as described in Section 2.1.2.

For lack of space, we do not formally describe subsumption checking in this paper.

[15] This requires extending the syntax of normal forms in a straightforward way to $\tau'_{\text{con}} ::= \tau_{\text{con}}\ \overline{\wedge\ \#F}$ and $\tau'_{\text{dis}} ::= \tau_{\text{dis}}\ \overline{\vee\ \#F}$.

3.4 Simplification and Presentation of Inferred Types

Type simplification and pretty-printing are important components of any practical implementation of MLsub and MLstruct. They indeed perform a lot of the heavy-lifting of type inference, massaging inferred types, which are often big and unwieldy, into neat and concise equivalent type expressions. In this section, we briefly explain how simplification is performed in MLstruct.

3.4.1 Basic simplifications. For basic simplifications, we essentially follow Parreaux [2020] — we remove polar occurrences of type variables, remove type variables "sandwiched" between identical bounds, and we perform some hash consing to simplify inferred recursive types. The simplification of unions, intersections, and negations is not fully addressed by Parreaux, since MLsub does not fully supports these features. In MLstruct, we apply standard Boolean algebra simplification techniques to simplify these types, such as putting them into disjunctive normal forms, simplifying complements, and factorizing common conjuncts. We also reduce types as they arise, based on Section 2.2.2.

3.4.2 Bound inlining. Many types can be represented equivalently using either bounded quantification or inlined intersection and union types, so we often have to choose between them. For instance, $\forall(\alpha \leqslant \mathsf{Int})\cdot(\beta \geqslant \mathsf{Int}).\ \alpha \to \alpha \to \beta$ is much better expressed as the equivalent $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. But whether $(\alpha \wedge \mathsf{Int}) \to (\alpha \wedge \mathsf{Int}) \to \alpha$ is better than the equivalent $\forall(\alpha \leqslant \mathsf{Int}).\ \alpha \to \alpha \to \alpha$ may depend on personal preferences. As a general rule of thumb, we only inline bounds when doing so would not duplicate them and when they are not cyclic (i.e., we do not inline recursive bounds).

3.5 Implementation

MLstruct is implemented in ~5000 lines of Scala code, including advanced type simplification algorithms and error reporting infrastructure.[16] We have an extensive tests suite consisting of more than 4000 lines of well-typed *and* ill-typed MLstruct expressions, for which we automatically check the output of the type simplifier and error reporting for regressions. Running this test suite in parallel takes ~2s on a 2020 iMac with a 3.8 GHz 8-Core Intel Core i7 and 32 GB 2667 MHz DDR4.

4 FORMAL SEMANTICS OF MLSTRUCT

In this section, we introduce $\lambda^{\neg}$, a formal calculus which reflects the core features of MLstruct.

4.1 Syntax

The syntax of $\lambda^{\neg}$ is presented in Figure 1. We use the notation $\overline{E_i}^{i}$ to denote a repetition of $i = 0$ to n occurrences of a syntax form E, and we use the shorthand $\overline{E}$ when i is not needed for disambiguation.

4.1.1 Core syntax. The core syntax of $\lambda^{\neg}$ follows the MLstruct source language presented previously quite closely, though it introduces a syntactic novelty: the *mode* $\diamond$ or $\circ$ of a syntactic form is used to deduplicate sentences that refer to unions and intersections as well as top and bottom, which are respective duals and can therefore often be treated symmetrically. For instance, $\top^{\diamond}$ is to be understood as either $\top^{\cdot}$ when $\diamond = \cdot$, i.e., $\top$, or as $\top^{\circlearrowright}$ when $\diamond = \circlearrowright$, i.e., $\bot$. A similar idea was developed independently by d. S. Oliveira et al. [2020] to cut down on boilerplate and repetition in formalizing subtyping systems.

Parametric polymorphism in $\lambda^{\neg}$ is attached solely to top-level '**def**' bindings, whose semantics, as in languages like Scala, is to re-evaluate their right-hand side every time they are referred to in the program. In contrast, local let bindings are desugared to immediately-applied lambdas, and are treated monomorphically. *Let polymorphism* is orthogonal to the features presented in this paper,

[16] This does not include about 1200 additional lines of code to generate JavaScript (the tests are run through NodeJS).

Core syntax

Type	τ, π	$::= \tau \to \tau \mid \{x : \tau\} \mid A[\overline{\tau}] \mid C[\overline{\tau}] \mid \#C \mid \alpha \mid \top^\diamond \mid \tau \vee^\diamond \tau \mid \neg\tau$
Mode	$\diamond, \circ$	$::= \cdot \mid \circlearrowright$
Polymorphic type	σ	$::= \forall \Xi. \tau$
Term	s, t	$::= x, y, z \mid t : \tau \mid \lambda x.\, t \mid t\ t \mid t.x \mid C\,\{\overline{x = t}\} \mid \textbf{case}\ x = t\ \textbf{of}\ M$
Case branches	M	$::= \epsilon \mid \_ \to t \mid C \to t, M$
Value	v, w	$::= \lambda x.\, t \mid C\,\{\overline{x = v}\}$
Program	P	$::= t \mid \textbf{def}\ x = t; P$
Top-level declaration	d	$::= \textbf{class}\ C[\overline{\alpha}] : \tau \mid \textbf{type}\ A[\overline{\alpha}] = \tau$

Contexts

Declarations context	$\mathcal{D}$	$::= \epsilon \mid \mathcal{D} \cdot d$
Typing context	Γ	$::= \epsilon \mid \Gamma \cdot (x : \tau) \mid \Gamma \cdot (x : \sigma)$
Subtyping context	Σ	$::= \Xi \mid \Sigma \cdot (\tau \leqslant \tau) \mid \Sigma \cdot \triangleright(\tau \leqslant \tau)$
Constraining context	Ξ	$::= \epsilon \mid \Xi \cdot (\alpha \leqslant \tau) \mid \Xi \cdot (\tau \leqslant \alpha) \mid \Xi \cdot \textbf{\textit{err}}$

Fig. 1. Syntax of types, terms, and contexts.

and can be handled by using a level-based algorithm [Parreaux 2020] on top of the core algorithm we describe here, as well as a value restriction if the language is meant to incorporate mutation.

In $\lambda^\neg$, **def** bindings are never recursive. This simplification is made without loss of generality, as recursion can be recovered using a Z fixed point combinator, typeable in MLsub [Dolan 2017] and thus also in $\lambda^\neg$. This combinator is defined as $t_Z = \lambda f.\, t'_Z\ t'_Z$ where $t'_Z = \lambda x.\, f\ (\lambda v.\, x\ x\ v)$. One can easily verify that t_Z can be typed as $((\alpha \to \beta) \to ((\alpha \to \beta) \wedge \gamma)) \to \gamma$.

To keep the formalism on point, we only present class object types, and ignore uninteresting primitive and built-in types like Int and Bool, which can be encoded as classes. Note that singleton types like 1, 2, and **true**, as we use them in the introduction, are easily encoded as subclasses 1_C, 2_C, and $\textbf{true}_C$ of the corresponding built-in types.

Finally, the syntax of pattern matching '**case** $x = t$ **of** ...' includes a variable binding because the rules for typing it will *refine* the type of that variable in the different branches. We do not use '**case** x **of** ...' as the core form in order to allow for simple substitution of variables with terms.

4.1.2 Contexts. We use four kinds of contexts. Declarations contexts $\mathcal{D}$ hold the type declarations of the program. Throughout this paper, **we assume an ambient declarations context** (i.e., our formal developments are implicitly parameterized by $\mathcal{D}$). Typing contexts Γ bind both *monomorphic* and *polymorphic* types, the latter corresponding to '**def**' bindings. Subtyping contexts Σ record assumptions about subtyping relationships, with some of these assumptions potentially hidden behind a $\triangleright$ (explained in Section 4.4.1). Finally, *polymorphic* or *constraining* contexts Ξ contain bounds/constraints on type variables and possibly errors ($\textbf{\textit{err}} \in \Xi$) encountered during type inference. The typing rules will ensure that in a polymorphic type $\forall \Xi.\, \tau$, context Ξ is *consistent*, which implies $\textbf{\textit{err}} \notin \Xi$. Note that Σ contexts are *rooted* in Ξ contexts because subtyping judgments require the former but are invoked from typing judgments, which use the latter for polymorphism.

4.1.3 Shorthands. Throughout this paper, we make use of the following notations and shorthands:

$$R ::= \{\overline{x = v}\} \qquad N ::= A \mid C \qquad H ::= \tau \leqslant \tau \qquad N \equiv N[\epsilon] \qquad C \to t \equiv C \to t, \epsilon$$

$$\{\overline{x : \tau_x}^{x \in S}, y : \tau_y\} \equiv \{\overline{x : \tau_x}^{x \in S}\} \wedge \{y : \tau_y\} \quad (y \notin S) \qquad \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \equiv (\lambda x.\, t_2)\ t_1$$

$$\textbf{case}\ y\ \textbf{of}\ M \equiv \textbf{case}\ x = y\ \textbf{of}\ [y \mapsto x]M \quad (x \notin FV(M))$$

$$E[\square] ::= \square\ t \mid v\ \square \mid \square.x \mid C\,\{\overline{x = v},\ y = \square,\ \overline{z = t}\} \mid \textbf{case}\ x = \square\ \textbf{of}\ M$$

E-Ctx	$E[t] \rightsquigarrow E[t']$	if $t \rightsquigarrow t'$
E-Def	$\textbf{def}\ x = t\,;\ P \rightsquigarrow [x \mapsto t]P$	
E-App	$(\lambda x.\ t)\ v \rightsquigarrow [x \mapsto v]t$	
E-Asc	$t : \tau \rightsquigarrow t$	
E-Proj	$v_1.x \rightsquigarrow v_2$	if $\{\, x = v_2 \,\} \in v_1$
E-CaseCls1	$\textbf{case}\ x = C_1\ R\ \textbf{of}\ C_2 \to t,\ M \rightsquigarrow [x \mapsto C_1\ R]t$	if $C_2 \in \mathcal{S}(\#C_1)$
E-CaseCls2	$\textbf{case}\ x = C_1\ R\ \textbf{of}\ C_2 \to t,\ M \rightsquigarrow \textbf{case}\ x = v\ \textbf{of}\ M$	if $C_2 \notin \mathcal{S}(\#C_1)$
E-CaseWld	$\textbf{case}\ x = v\ \textbf{of}\ \_ \to t \rightsquigarrow [x \mapsto v]t$	

Fig. 2. Small-step evaluation rules.

4.2 Evaluation Rules

The small-step reduction semantics of $\lambda^{\neg}$ is shown in Figure 2. The relation $P \rightsquigarrow P'$ reads "program P evaluates to program P' in one step." Note that P here may refer to a simple term t.

We write $\{\, x = v_2 \,\} \in v_1$ to say that v_1 is a value of the form '$C\,\{\,\overline{z = w},\ x = v_2\,\}$' or of the form '$C\,\{\,\overline{z = w},\ y = v_2'\,\}$' where $y \neq x$ and $\{\, x = v_2 \,\} \in C\,\{\,\overline{z = w}\,\}$. Class instances are constructed via the $C\ R$ introduction form, where R is a record of the fields of the instance. Instance matching works by inspecting the runtime instance of a scrutinee value, in order to determine which corresponding branch to evaluate. This is done through the *superclasses* function $\mathcal{S}(\tau)$. We define the *superclasses* $\mathcal{S}(\tau)$ of a type τ as the set of classes transitively inherited by type τ, assuming τ is a class type or the expansion of a class type. Note that a term of the shape '$\textbf{case}\ x = v\ \textbf{of}\ \epsilon$' is stuck.

4.3 Declarative Typing Rules

Program-typing judgments $\Xi, \Gamma \vdash^{\star} P : \tau$ are used to type programs while *term*-typing judgments $\Xi, \Gamma \vdash t : \tau$ are used to type **def** right-hand sides and program bodies. The latter judgement is read "under type variable bounds Ξ and in context Γ, term t has type τ." We present only the rules for the latter judgment in Figure 3, as they are the more interesting ones, and relegate the auxiliary *program-typing* ($\Xi, \Gamma \vdash^{\star} P : \tau$), *consistency* ($\Sigma$ ***cons.***) and *subtyping entailment* ($\Sigma \vdash \sigma \leqslant^{\forall} \sigma$ and $\Sigma \models \Sigma$) rules to the extended version of this paper. The consistency judgment is used to make sure we type `def`s and program bodies under *valid* (i.e., consistent) bounds only.[17]

Rule T-Obj features a few technicalities deserving of careful explanations. First, notice that its result type is an intersection of the nominal class tag $\#C$ with a record type of all the fields passed in the instantiation. Importantly, these fields may have any types, including ones not compatible with the field declarations in C or its parents. This simplifies the meta theory (especially type inference) and is done without loss of generality: indeed, we can *desugar* '`C {x = t, ...}`' instantiations in MLstruct into a type-ascribed instantiation '$C\{\, x = t,\ \ldots \,\} : C[\overline{\alpha}]$' in $\lambda^{\neg}$,[18] where all $\overline{\alpha}$ are fresh, which will ensure that the provided fields satisfy their declared types in C.

T-Obj also requires C to be "final" using the C ***final*** judgment (formally defined in the extended version of the paper). This means that C is not extended by any other classes in $\mathcal{D}$. It ensures that, at runtime, for every class pattern D, pattern-matching scrutinees are always instances of a class D' that is either a subclass of D (meaning $\#D' \leqslant \#D$) or an unrelated class (meaning $\#D' \leqslant \neg\#D$). Without this property, type preservation would technically not hold. Indeed, consider the program:

[17] Indeed, under inconsistent bounds, ill-typed terms become typeable. For example, we have $(\mathrm{Int} \leqslant \mathrm{Int} \to \mathrm{Int}) \vdash 1\ 1 : \mathrm{Int}$.
[18] The alternative desugaring '$\textbf{let}\ tmp = C\{\, x = t,\ \ldots \,\}\ \textbf{in let}\ \_ = tmp : C[\overline{\alpha}]\ \textbf{in}\ tmp$' is nicer because it allows the user to retain refined field types (as described in Section 2.1.2) as well as any new fields that were not declared in C or its parents.

$\boxed{\Xi, \Gamma \vdash t : \tau}$

$$\textsc{T-Subs}\ \frac{\Xi, \Gamma \vdash t : \tau_1 \quad \Xi \vdash \tau_1 \leqslant \tau_2}{\Xi, \Gamma \vdash t : \tau_2} \qquad \textsc{T-Obj}\ \frac{\overline{\Xi, \Gamma \vdash t : \tau} \quad C\ \textbf{\textit{final}}}{\Xi, \Gamma \vdash C\ \{\overline{x = t}\} : \#C \wedge \{\overline{x : \tau}\}}$$

$$\textsc{T-Proj}\ \frac{\Xi, \Gamma \vdash t : \{x : \tau\}}{\Xi, \Gamma \vdash t.x : \tau} \qquad \textsc{T-Var1}\ \frac{\Gamma(x) = \tau}{\Xi, \Gamma \vdash x : \tau} \qquad \textsc{T-Var2}\ \frac{\Gamma(x) = \sigma \quad \Xi \vdash \sigma \leqslant^{\forall} \forall \epsilon. \tau}{\Xi, \Gamma \vdash x : \tau} \qquad \textsc{T-Abs}\ \frac{\Xi, \Gamma{\cdot}(x : \tau_1) \vdash t : \tau_2}{\Xi, \Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}$$

$$\textsc{T-App}\ \frac{\Xi, \Gamma \vdash t_0 : \tau_1 \to \tau_2 \quad \Xi, \Gamma \vdash t_1 : \tau_1}{\Xi, \Gamma \vdash t_0\ t_1 : \tau_2} \qquad \textsc{T-Asc}\ \frac{\Xi, \Gamma \vdash t : \tau}{\Xi, \Gamma \vdash (t : \tau) : \tau}$$

$$\textsc{T-Case1}\ \frac{\Xi, \Gamma \vdash t_1 : \bot}{\Xi, \Gamma \vdash \textbf{case}\ x = t_1\ \textbf{of}\ \epsilon : \bot} \qquad \textsc{T-Case2}\ \frac{\Xi, \Gamma \vdash t_1 : \tau_1 \wedge \#C \quad \Xi, \Gamma{\cdot}(x : \tau_1) \vdash t_2 : \tau}{\Xi, \Gamma \vdash \textbf{case}\ x = t_1\ \textbf{of}\ \_ \to t_2 : \tau}$$

$$\textsc{T-Case3}\ \frac{\Xi, \Gamma \vdash t_1 : \#C \wedge \tau_1 \vee \neg\#C \wedge \tau_2 \quad \Xi, \Gamma{\cdot}(x : \tau_1) \vdash t_2 : \tau \quad \Xi, \Gamma{\cdot}(x : \tau_2) \vdash \textbf{case}\ x = x\ \textbf{of}\ M : \tau}{\Xi, \Gamma \vdash \textbf{case}\ x = t_1\ \textbf{of}\ C \to t_2, M : \tau}$$

Fig. 3. Term typing rules. The full set of typing rules is available in the paper's extended version.

```
class C1   class C2: C1   class C3
case x = C1{} of C2 → C3{}, _ →x
```

This program can be given type $\neg C_2$ since $C_1 \leqslant C_2 \vee \neg C_2 \equiv \top$ (in T-Case3, we pick $\tau_2 = \neg C_2$), but it reduces to $C_1\{\}$, which does *not* have type $\neg C_2$ because C_1 and C_2 are *not* unrelated classes.

This finality requirement is merely a technicality of $\lambda^{\neg}$ and it does not exist in MLstruct, where non-final classes can be instantiated. This can be understood as each MLstruct class C implicitly defining a final version C^F of itself, which is used upon instantiation. So the MLstruct program above would actually denote the following desugared $\lambda^{\neg}$ program:

```
class C1   class C1^F: C1   class C2: C1   class C3   class C3^F: C3
case x = C1^F{} : C1 of C2 → C3^F{} : C3, _ →x
```

The refined program above now evaluates to $C_1^F\{\}$, of type C_1^F, which *is* a subtype of $\neg C_2$.

In T-Subs, we use the current constraining context Ξ as a subtyping context Σ when invoking the subtyping judgement $\Xi \vdash \tau_1 \leqslant \tau_2$ (presented in the next subsection), which is possible since the syntax of constraining contexts is a special case of the syntax of subtyping contexts.

Rule T-Var2 uses the *entailment* judgment $\Xi \vdash \sigma \leqslant^{\forall} \forall \epsilon. \tau$ defined in appendix to *instantiate* the polymorphic type found in the context.

The typing of instance matching is split over three rules. Rule T-Case1 specifies that no scrutinee can be matched by a **case** expression with no branches, which is expressed by assigning type $\bot$ (the type inhabited by no value) to the scrutinee.

Rule T-Case2 handles **case** expressions with a single, default case, which is equivalent to a let binding, where the body t_2 of the default case is typed within a typing context extended with the case-bound variable x and the type of the scrutinee. This rule requires the scrutinee to have a class type $\#C$; this is to prevent functions from being matched, because that would technically break preservation in a similar way as described above (since we *do not* have $\pi_1 \to \pi_2 \leqslant \neg\#D$).

T-Case3 is the more interesting instance matching rule. We first assume that the scrutinee t_1 has some type τ_1 in order to type the first **case** branch, and then assume t_1 has type τ_2 to type the rest

of the instance matching (by reconstructing a smaller **case** expression binding a new variable x which shadows the old variable occurring in M). Then, we make sure that the scrutinee t_1 can be typed at $\#C \wedge \tau_1 \vee \neg\#C \wedge \tau_2$, which ensures that if t_1 is an instance of C, then it is also of type τ_1, and if not, then it is of type τ_2. In this rule, τ_1 can be picked to be anything, so assuming $\Gamma\cdot(x : \tau_1)$ to type t_2 is sufficient, and there is no need to assume $\Gamma\cdot(x : \tau_1 \wedge \#C)$. If the t_2 branch needs τ_1 to be a subtype of $\#C$, we can always pick $\tau_1 = \tau_1' \wedge \#C$. Notice that the required type for t_1 still has the same shape $\#C \wedge \tau_1 \vee \neg\#C \wedge \tau_2 \equiv \#C \wedge (\#C \wedge \tau_1') \vee \neg\#C \wedge \tau_2 \equiv \#C \wedge \tau_1' \vee \neg\#C \wedge \tau_2$.

4.4 Declarative Subtyping Rules

The declarative subtyping rules are presented in Figure 4. Remember that the mode syntax $\diamond$ is used to factor in dual formulations. For instance, $\tau \leqslant^{\diamond} \top^{\diamond}$ is to be understood as either $\tau \leqslant \top$ when $\diamond = \cdot$, i.e., $\tau \leqslant \top$, or as $\tau \leqslant^{\curvearrowright} \top^{\curvearrowright}$ when $\diamond = \curvearrowright$, i.e., $\tau \geqslant \bot$, also written $\bot \leqslant \tau$. The purpose of rule S-Weaken is solely to make rules which need no context slightly more concise to state. In this paper, we usually treat applications of S-Weaken implicitly.

4.4.1 Subtyping recursive types. A consequence of our syntactic account of subtyping is that we do *not* define types as some fixed point over a generative relation, as done in, e.g., [Dolan 2017; Pierce 2002]. Instead, we have to account for the fact that we manipulate *finite* syntactic type trees, in which recursive types have to be manually unfolded to derive things about them. This is the purpose of the S-Exp rules, which substitute a possibly-recursive type with its body to expose one layer of its underlying definition. As remarked by Amadio and Cardelli [1993, §3.2], to subtype recursive

$\boxed{\Sigma \vdash \tau \leqslant \tau}$ $\boxed{\tau \leqslant \tau}$ $\quad \triangleleft\Xi = \Xi \quad \triangleleft(\Sigma \cdot H) = \triangleleft\Sigma \cdot H \quad \triangleleft(\Sigma \cdot \triangleright H) = \triangleleft\Sigma \cdot H$

$$\textsc{S-Refl}\ \frac{}{\tau \leqslant \tau} \qquad \textsc{S-ToB}\diamond\ \frac{}{\tau \leqslant^{\diamond} \top^{\diamond}} \qquad \textsc{S-Compl}\diamond\ \frac{}{\tau \vee^{\diamond} \neg\tau \geqslant^{\diamond} \top^{\diamond}} \qquad \textsc{S-NegInv}\ \frac{\Sigma \vdash \tau_1 \leqslant \tau_2}{\Sigma \vdash \neg\tau_2 \leqslant \neg\tau_1}$$

$$\textsc{S-AndOr11}\diamond\ \frac{}{\tau_1 \vee^{\diamond} \tau_2 \geqslant^{\diamond} \tau_1} \qquad \textsc{S-AndOr12}\diamond\ \frac{}{\tau_1 \vee^{\diamond} \tau_2 \geqslant^{\diamond} \tau_2}$$

$$\textsc{S-AndOr2}\diamond\ \frac{\Sigma \vdash \tau \geqslant^{\diamond} \tau_1 \quad \Sigma \vdash \tau \geqslant^{\diamond} \tau_2}{\Sigma \vdash \tau \geqslant^{\diamond} \tau_1 \vee^{\diamond} \tau_2} \qquad \textsc{S-Distrib}\diamond\ \frac{}{\tau \wedge^{\diamond} (\tau_1 \vee^{\diamond} \tau_2) \leqslant^{\diamond} (\tau \wedge^{\diamond} \tau_1) \vee^{\diamond} (\tau \wedge^{\diamond} \tau_2)}$$

$$\textsc{S-Trans}\ \frac{\Sigma \vdash \tau_0 \leqslant \tau_1 \quad \Sigma \vdash \tau_1 \leqslant \tau_2}{\Sigma \vdash \tau_0 \leqslant \tau_2}$$

$$\textsc{S-Weaken}\ \frac{H}{\Sigma \vdash H} \qquad \textsc{S-Assum}\ \frac{}{\Sigma \cdot \triangleright H \vdash H} \qquad \textsc{S-Hyp}\ \frac{H \in \Sigma}{\Sigma \vdash H} \qquad \textsc{S-ClsSub}\ \frac{C_2 \in \mathcal{S}(\#C_1)}{\#C_1 \leqslant \#C_2} \qquad \textsc{S-ClsBot}\ \frac{C_1 \notin \mathcal{S}(\#C_2) \quad C_2 \notin \mathcal{S}(\#C_1)}{\#C_1 \wedge \#C_2 \leqslant \bot}$$

$$\textsc{S-FunDepth}\ \frac{\triangleleft\Sigma \vdash \tau_0 \leqslant \tau_1 \quad \triangleleft\Sigma \vdash \tau_2 \leqslant \tau_3}{\Sigma \vdash \tau_1 \to \tau_2 \leqslant \tau_0 \to \tau_3} \qquad \textsc{S-FunMrg}\diamond\ \frac{}{(\tau_1 \vee^{\diamond} \tau_3) \to (\tau_2 \wedge^{\diamond} \tau_4) \geqslant^{\diamond} \tau_1 \to \tau_2 \wedge^{\diamond} \tau_3 \to \tau_4} \qquad \textsc{S-Exp}\diamond\ \frac{\tau\ \textbf{\textit{exp.}}\ \tau'}{\tau \geqslant^{\diamond} \tau'}$$

$$\textsc{S-RcdDepth}\ \frac{\triangleleft\Sigma \vdash \tau_1 \leqslant \tau_2}{\Sigma \vdash \{x : \tau_1\} \leqslant \{x : \tau_2\}} \qquad \textsc{S-RcdMrg}\diamond\ \frac{}{\{x : \tau_1 \vee^{\diamond} \tau_2\} \leqslant^{\diamond} \{x : \tau_1\} \vee^{\diamond} \{x : \tau_2\}} \qquad \textsc{S-RcdTop}\ \frac{\tau \in \{\{y^{\neq x} : \tau_2\}, \tau_2 \to \tau_3\}}{\top \leqslant \{x : \tau_1\} \vee \tau}$$

$\boxed{\tau\ \textbf{\textit{exp.}}\ \tau}$

$$\textsc{S-AlsExp}\ \frac{(\textbf{type}\ A[\overline{\alpha_i}^{i \in S}] = \tau) \in \mathcal{D}}{A[\overline{\tau_i}^{i \in S}]\ \textbf{\textit{exp.}}\ [\overline{\alpha_i \mapsto \tau_i}^{i \in S}]\tau} \qquad \textsc{S-ClsExp}\ \frac{(\textbf{class}\ C[\overline{\alpha_i}^{i \in S}] : \tau) \in \mathcal{D}}{C[\overline{\tau_i}^{i \in S}]\ \textbf{\textit{exp.}}\ \#C \wedge [\overline{\alpha_i \mapsto \tau_i}^{i \in S}]\tau}$$

Fig. 4. Declarative subtyping rules.

types, it is not enough to simply allow unfolding them a certain number of times. Moreover, in our system, recursive types may *arise* from cyclic type variable constraints (which is important for type inference), and thus not be attached to any explicit recursive binders. Thus, we cannot simply follow Castagna [2012, §1.3.4] in admitting a μ rule, which would still be insufficient.

4.4.2 Subtyping hypotheses. We make use of the Σ environment to store subtyping hypotheses via S-Assum, to be leveraged later using the S-Hyp rule. We should be careful not to allow the use of a hypothesis right after assuming it, which would obviously make the system unsound (as it could derive any subtyping). In the specification of their constraint solving algorithm, Hosoya et al. [2005] use two distinct judgments $\vdash$ and $\vdash'$ to distinguish from places where the hypotheses can or cannot be used. We take a different, but related approach. Our S-Assum subtyping rule resembles the Löb rule described by Appel et al. [2007], which uses the "later" modality $\triangleright$ in order to delay the applicability of hypotheses — by placing this symbol in front of the hypothesis being assumed, we prevent its immediate usage by S-Hyp. We eliminate $\triangleright$ when passing through a function or record constructor: the dual $\triangleleft$ symbol is used to remove all $\triangleright$ from the set of hypotheses, making them available for use by S-Hyp. These precautions reflect the "guardedness" restrictions used by Dolan [2017] on recursive types, which prevents usages of α that are not guarded by $\rightarrow$ or $\{ \ldots \}$ in a recursive type $\mu\alpha.\, \tau$. Such productivity restriction is also implemented by our guardedness check, preventing the definition of types such as **type** $A = A$ and **type** $A = \neg A$ (Section 2.1.6).[19]

4.4.3 A Boolean algebra. The subtyping preorder in $\lambda^{\neg}$ gives rise to a Boolean *lattice* or *algebra* when taking the equivalence relation '$\tau_1 \equiv \tau_2$' to be the relation induced by '$\tau_1 \leqslant \tau_2$ *and* $\tau_2 \leqslant \tau_1$'. To see why, let us inspect the standard way of defining Boolean algebras, which is as the set of *complemented distributive lattices*. We can define a lattice equivalently as either:

- An algebra $\langle L, \wedge, \vee \rangle$ such that $\wedge$ and $\vee$ are idempotent, commutative, associative, and satisfy the absorption law, i.e., $\tau \wedge (\tau \vee \pi) \equiv \tau \vee (\tau \wedge \pi) \equiv \tau$. Then $\tau_1 \leqslant \tau_2$ is taken to mean $\tau_1 \equiv \tau_1 \wedge \tau_2$ or (equivalently) $\tau_1 \vee \tau_2 \equiv \tau_2$.
- A partially-ordered set $\langle L, \leqslant \rangle$ (i.e., $\leqslant$ is reflexive, transitive, and antisymmetric) where every two elements τ_1 and τ_2 have a least upper bound $\tau_1 \vee \tau_2$ (supremum) and a greatest lower bound $\tau_1 \wedge \tau_2$ (infimum). That is, $\forall\, \pi \leqslant \tau_1, \tau_2.\ \pi \leqslant \tau_1 \wedge \tau_2$ and $\forall\, \pi \geqslant \tau_1, \tau_2.\ \pi \geqslant \tau_1 \vee \tau_2$.

The latter is most straightforward to show: we have reflexivity by S-Refl, transitivity by S-Trans, antisymmetry by definition of $\equiv$, and the supremum and infimum properties are given directly by S-AndOr2$\cdot$ and S-AndOr2$\supset$ respectively.

Moreover, to be a Boolean algebra, our lattice needs to be:

- a *complemented* lattice, which is
  - bounded: $\top$ and $\bot$ are respective *least* and *greatest* elements (S-ToB$\diamond$);
  - such that every τ has a complement $\neg\tau$ where $\tau \vee \neg\tau \equiv \top$ and $\tau \wedge \neg\tau \equiv \bot$ (S-Compl$\diamond$);[20]
- a *distributive* lattice, meaning that $\tau \wedge^{\diamond} (\tau_1 \vee^{\diamond} \tau_2) \equiv (\tau \wedge^{\diamond} \tau_1) \vee^{\diamond} (\tau \wedge^{\diamond} \tau_2)$ for $\diamond \in \{ \supset, \cdot \}$.

The first direction $\leqslant^{\diamond}$ of distributivity is given directly by S-Distrib. The other direction $\geqslant^{\diamond}$ is admissible: since $\tau_1 \vee^{\diamond} \tau_2 \geqslant^{\diamond} \tau_1$ (S-AndOr11$\diamond$) and $\tau_1 \vee^{\diamond} \tau_2 \geqslant^{\diamond} \tau_2$ (S-AndOr12$\diamond$), we can easily derive $\tau \wedge^{\diamond} (\tau_1 \vee^{\diamond} \tau_2) \geqslant^{\diamond} \tau \wedge^{\diamond} \tau_1$ and $\tau \wedge^{\diamond} (\tau_1 \vee^{\diamond} \tau_2) \geqslant^{\diamond} \tau \wedge^{\diamond} \tau_2$, and by (S-AndOr2$\diamond$) we conclude that $\tau \wedge^{\diamond} (\tau_1 \vee^{\diamond} \tau_2) \geqslant^{\diamond} (\tau \wedge^{\diamond} \tau_1) \vee^{\diamond} (\tau \wedge^{\diamond} \tau_2)$.

A useful property of Boolean algebras is that the usual De Morgan's laws hold, which will allow us to massage constrains into normal forms during type inference.

[19] Perhaps counter-intuitively, it is *not* a problem to infer types like '$\forall(\alpha \leqslant \alpha).\ \tau$' and '$\forall(\alpha \leqslant \neg\alpha).\ \tau$' because such "funny" cyclic bounds, unlike unproductive recursive types, do not actually allow concluding incorrect subtyping relationships.
[20] We can also show that our lattice is *uniquely* complemented, i.e., $\neg\tau_1 \equiv \neg\tau_2$ implies $\tau_1 \equiv \tau_2$.

4.4.4 Algebraic rules. We call S-FunMrg and S-RcdTop *algebraic* subtyping rules because they do not follow from a set-theoretic interpretation of order connectives ($\wedge$, $\vee$, $\neg$). Their role is to make record and function types lattice homomorphisms,[21] which is required to make type inference complete — we will see that it allows the existence of well-behaved normal forms. Though one can still think of types as sets of values, as in the *semantic subtyping* approach, in $\lambda^\neg$ the sets of values of $\tau_1 \wedge \tau_2$ is *not* the intersection of the sets of values of τ_1 and τ_2 (unless τ_1 and τ_2 are nominal tags or records), and similarly for unions and complements. These algebraic rules are sound in $\lambda^\neg$ because of the careful use we make of unions and intersections, e.g., not using intersections to encode overloading. Notably, S-RcdTop implies surprising relationships like $\neg(\tau_1 \to \tau_2) \leqslant \{ x : \pi \}$ and $\neg\{ x : \pi \} \leqslant \{ y : \pi \}$ ($x \neq y$), exemplifying that negation in $\lambda^\neg$ is essentially *algebraic*.

4.5 Soundness of the Declarative Type System

We now state the main soundness theorems for $\lambda^\neg$'s type system, proven in the extended version of this paper. In the following, $\vdash^\star$ is used as the syntax for *program*-typing judgments.

Theorem 4.1 (Progress). *If $\vdash^\star P : \tau$ and P is not a value, then $\vdash P \rightsquigarrow P'$ for some P'.*

Theorem 4.2 (Preservation). *If $\vdash^\star P : \tau$ and $\vdash P \rightsquigarrow P'$, then $\vdash^\star P' : \tau$.*

5 PRINCIPAL TYPE INFERENCE FOR $\lambda^\neg$

We now formally describe the type inference algorithm which was presented in Section 3.

5.1 Type Inference Rules

Our type inference rules are presented in Figure 5. The judgments $\Gamma \Vdash^\star P : \tau \Rightarrow \Xi$ and $\Xi, \Gamma \Vdash t : \tau \Rightarrow \Xi$ are similar to their declarative typing counterparts, except that they are *algorithmic* and produce constraining contexts Ξ containing inferred type variables bounds.

We give the following formal meaning to premises of the form 'α ***fresh***', and in the rest of this paper, we implicitly only consider well-formed derivations:

Definition 5.1 (Well-formed derivations). A type inference or constraining derivation is said to be *well-formed* if, for every α, the 'α ***fresh***' premise appears at most once in the entire derivation and, if it does, α does not occur in any user-specified type (i.e., on the right of ascription trees '$t : \tau$').

The *program*-typing inference rules I-Body and I-Def mirror their declarative counterparts. In I-Def, notice how the output context corresponding to the definition's body is the one used to quantify the corresponding type in the typing context.

The main difference between type inference rules and declarative typing rules is that in the former, we immediately produce a type for each subexpression irrelevant of its context, using type variables for local unknowns, and we then use a *constraining* judgement $\Sigma \vdash \tau \ll \pi \Rightarrow \Xi$ (explained in the next subsection) to make sure that the inferred type τ conforms to the expected type π in this context. So whenever we need to guess a type (such as the type of a lambda's parameter in I-Abs), we simply introduce a fresh type variable. As an example, in I-Proj, we infer an unconstrained type τ for the field projection's prefix t, and then make sure that this is a subtype of a record type by constraining $\Xi_0 \vdash \tau \ll \{ x : \alpha \} \Rightarrow \Xi_1$ — where Ξ_1 is the output context containing the type variable bounds necessary to make this relationship hold. Rules I-App, I-Asc, I-Case1, I-Case2, and I-Case3 all work according to the same principles, threading the set of constraining contexts currently inferred through the next type inference steps, which is necessary to make sure that all inferred type variable bounds are consistent with each other. Rule I-Var2 *refreshes* all the variables

[21] A lattice homomorphism f is such that $f(\tau \vee \pi) \equiv f(\tau) \vee f(\pi)$ and $f(\tau \wedge \pi) \equiv f(\tau) \wedge f(\pi)$. Function types are lattice homomorphisms in their parameters in the sense that $f(\tau) = (\neg\tau) \to \pi$ is a lattice homomorphism.

$\boxed{\Gamma \Vdash^\star P : \tau \Rightarrow \Xi}$

$$\textsc{I-Body}\ \frac{\Gamma \Vdash t : \tau \Rightarrow \Xi}{\Gamma \Vdash^\star t : \tau \Rightarrow \Xi} \qquad \textsc{I-Def}\ \frac{\Gamma \Vdash t : \tau \Rightarrow \Xi \quad \Gamma{\cdot}(x : \forall \Xi.\ \tau) \Vdash^\star P : \pi \Rightarrow \Xi'}{\Gamma \Vdash^\star \mathbf{def}\ x = t\,;\, P : \pi \Rightarrow \Xi'}$$

$\boxed{\Xi, \Gamma \Vdash t : \tau \Rightarrow \Xi}$

$$\textsc{I-Proj}\ \frac{\Xi_0, \Gamma \Vdash t : \tau \Rightarrow \Xi_1 \quad \alpha\ \textit{fresh} \quad \Xi_0{\cdot}\Xi_1 \vdash \tau \ll \{\, x : \alpha \,\} \Rightarrow \Xi_2}{\Xi_0, \Gamma \Vdash t.x : \alpha \Rightarrow \Xi_1{\cdot}\Xi_2}$$

$$\textsc{I-Obj}\ \frac{\Xi_0, \Gamma \Vdash t_1 : \tau_1 \Rightarrow \Xi_1 \quad \Xi_0{\cdot}\Xi_1, \Gamma \Vdash t_2 : \tau_2 \Rightarrow \Xi_2 \quad \ldots \quad \Xi_0{\cdot}\Xi_1{\cdot}\ldots{\cdot}\Xi_{n-1}, \Gamma \Vdash t_n : \tau_n \Rightarrow \Xi_n \quad C\ \textit{final}}{\Xi_0, \Gamma \Vdash C\,\{\, x_1 = t_1;\ x_2 = t_2;\ \ldots;\ x_n = t_n \,\} : \#C \wedge \{\, x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots;\ x_n : \tau_n \,\} \Rightarrow \Xi_1{\cdot}\ldots{\cdot}\Xi_n}$$

$$\textsc{I-Var1}\ \frac{\Gamma(x) = \tau}{\Xi, \Gamma \Vdash x : \tau \Rightarrow \epsilon} \qquad \textsc{I-Var2}\ \frac{\Gamma(x) = \forall \Xi_1.\ \tau_1 \quad TV(\forall \Xi_1.\ \tau_1) = S \quad \overline{\gamma_\alpha\ \textit{fresh}}^{\alpha \in S}}{\Xi_0, \Gamma \Vdash x : [\overline{\alpha \mapsto \gamma_\alpha}^{\alpha \in S}]\tau_1 \Rightarrow [\overline{\alpha \mapsto \gamma_\alpha}^{\alpha \in S}]\Xi_1}$$

$$\textsc{I-Abs}\ \frac{\alpha\ \textit{fresh} \quad \Xi_0, \Gamma{\cdot}(x : \alpha) \Vdash t : \tau \Rightarrow \Xi_1}{\Xi_0, \Gamma \Vdash \lambda x.\ t : \alpha \to \tau \Rightarrow \Xi_1} \qquad \textsc{I-App}\ \frac{\Xi_0, \Gamma \Vdash t_1 : \tau_1, \Rightarrow \Xi_1 \quad \Xi_0{\cdot}\Xi_1, \Gamma \Vdash t_2 : \tau_2 \Rightarrow \Xi_2 \quad \alpha\ \textit{fresh} \quad \Xi_0{\cdot}\Xi_1{\cdot}\Xi_2 \vdash \tau_1 \ll \tau_2 \to \alpha \Rightarrow \Xi_3}{\Gamma, \Xi_0 \Vdash t_1\ t_2 : \alpha \Rightarrow \Xi_1{\cdot}\Xi_2{\cdot}\Xi_3}$$

$$\textsc{I-Asc}\ \frac{\Xi_0, \Gamma \Vdash t : \tau_1 \Rightarrow \Xi_1 \quad \Xi_0{\cdot}\Xi_1 \vdash \tau_1 \ll \tau_2 \Rightarrow \Xi_2}{\Xi_0, \Gamma \Vdash (t : \tau_2) : \tau_2 \Rightarrow \Xi_1{\cdot}\Xi_2} \qquad \textsc{I-Case1}\ \frac{\Xi_0, \Gamma \Vdash t_1 : \tau_1 \Rightarrow \Xi_1 \quad \Xi_0{\cdot}\Xi_1 \vdash \tau_1 \ll \bot \Rightarrow \Xi_2}{\Xi_0, \Gamma \Vdash \mathbf{case}\ x = t_1\ \mathbf{of}\ \epsilon : \bot \Rightarrow \Xi_1{\cdot}\Xi_2}$$

$$\textsc{I-Case2}\ \frac{\Xi_0, \Gamma \Vdash t_1 : \tau_1 \Rightarrow \Xi_1 \quad \Xi_0{\cdot}\Xi_1 \vdash \tau_1 \ll \#C \Rightarrow \Xi_2 \quad \Xi_0{\cdot}\Xi_1{\cdot}\Xi_2, \Gamma{\cdot}(x : \tau_1) \Vdash t_2 : \tau \Rightarrow \Xi_3}{\Xi_0, \Gamma \Vdash \mathbf{case}\ x = t_1\ \mathbf{of}\ \_ \to t_2 : \tau \Rightarrow \Xi_1{\cdot}\Xi_2{\cdot}\Xi_3}$$

$$\textsc{I-Case3}\ \frac{\begin{array}{c}\Xi_0, \Gamma \Vdash t_1 : \tau_1 \Rightarrow \Xi_1 \quad \alpha\ \textit{fresh} \quad \Xi_0{\cdot}\Xi_1, \Gamma{\cdot}(x : \alpha) \Vdash t_2 : \tau_2 \Rightarrow \Xi_2 \\ \beta\ \textit{fresh} \quad \Xi_0{\cdot}\Xi_1{\cdot}\Xi_2, \Gamma{\cdot}(x : \beta) \Vdash \mathbf{case}\ x = x\ \mathbf{of}\ M : \tau_3 \Rightarrow \Xi_3 \quad \Xi_0{\cdot}\Xi_1{\cdot}\Xi_2{\cdot}\Xi_3 \vdash \tau_1 \ll \#C \wedge \alpha \vee \neg\#C \wedge \beta \Rightarrow \Xi_4\end{array}}{\Xi_0, \Gamma \Vdash \mathbf{case}\ x = t_1\ \mathbf{of}\ C \to t_2,\ M : \tau_2 \vee \tau_3 \Rightarrow \Xi_1{\cdot}\Xi_2{\cdot}\Xi_3{\cdot}\Xi_4}$$

Fig. 5. Algorithmic type inference rules.

of a type $\forall \Xi.\ \tau$ obtained from the typing context, which includes both variables that occur in the constraining context Ξ as well as those that occur in the underlying type τ, even when some of the latter may not be mentioned in Ξ; indeed, in $\lambda^\neg$ all type variables are implicitly quantified.

5.2 Reduced Disjunctive Normal Forms

To facilitate constraint solving, it is useful to massage types into a normal form which we call RDNF, for *reduced disjunctive normal form*. This normal form is similar to a classical disjunctive normal form (DNF) except that we reduce all "incompatible" intersections and unions to $\bot$ and $\top$ respectively. Here, *incompatible* means that the type holds no useful information, either because it is inhabited by no value or because it cannot be used meaningfully, as explained in Section 2.2.2.

The syntax of RDNF is given below. It is indexed by a level n and there are two possible levels: level-0 RDNF, written D^0 does not contain any occurrence of class or alias types at the top level (they will have been expanded); whereas level-1 RDNF, written D^1, allows them. **Notation:** we will often write D as a shorthand for D^1 (and similarly for the other indexed syntax forms).

$$\mathsf{D}^n ::= \bot \mid \mathsf{C}^n \mid \mathsf{D}^n \vee \mathsf{C}^n \qquad \mathsf{C}^n ::= \mathsf{I}^n \wedge \neg\mathsf{U}^n \mid \mathsf{C}^n \wedge \alpha \mid \mathsf{C}^n \wedge \neg\alpha$$
$$\mathsf{I}^1 ::= \mathsf{I}^0 \mid \mathsf{I}^1 \wedge N[\overline{\mathsf{D}^1}] \qquad \mathsf{I}^0 ::= \mathcal{I}^{\mathcal{N}}[\mathcal{N}] \mid \mathcal{I}^{\rightarrow}[\mathcal{F}] \mid \mathcal{I}^{\{\}}[\mathcal{R}]$$
$$\mathsf{U}^1 ::= \mathsf{U}^0 \mid \mathsf{U}^1 \vee N[\overline{\mathsf{D}^1}] \qquad \mathsf{U}^0 ::= \bot \mid \mathsf{D}^1 \to \mathsf{D}^1 \mid \{x : \mathsf{D}^1\} \mid \mathsf{U}^0 \vee \#C$$

where the $\mathcal{I}^{\cdot}$ contexts stand for combinations of nominal tags $\mathcal{N}$, functions $\mathcal{F}$, and records $\mathcal{R}$:

$$\mathcal{I}^{\mathcal{N}}[\square] ::= \square \wedge \mathcal{F} \wedge \mathcal{R} \qquad \mathcal{N} ::= \top \mid \#C \qquad \mathcal{I}[\square] ::= \mathcal{I}^{\mathcal{N}}[\square] \mid \mathcal{I}^{\rightarrow}[\square] \mid \mathcal{I}^{\{\}}[\square]$$
$$\mathcal{I}^{\rightarrow}[\square] ::= \mathcal{N} \wedge \square \wedge \mathcal{R} \qquad \mathcal{F} ::= \top \mid \mathsf{D}^1 \to \mathsf{D}^1 \qquad \top^3 ::= \top \wedge \top \wedge \top$$
$$\mathcal{I}^{\{\}}[\square] ::= \mathcal{N} \wedge \mathcal{F} \wedge \square \qquad \mathcal{R} ::= \top \mid \{\overline{x : \mathsf{D}^1}\}$$

As an example, '$\mathsf{D}_1 = \#C \wedge \top \wedge \{x : \top\} \wedge C[\mathsf{Int}, \mathsf{Bool}] \wedge A[\mathsf{Str}] \wedge \neg\bot \wedge \neg\alpha$' is a valid level-1 RDNF, but not a valid level-0 one because $C[\mathsf{Int}, \mathsf{Bool}]$ and $A[\mathsf{Str}]$ occur at the top level and are not expanded, while '$\mathsf{D}_2^n = \top \wedge \top \wedge \{x : C[\mathsf{Int}, \mathsf{Bool}]\} \wedge \neg\bot$' is well-defined for both $n \in \{0, 1\}$.

5.2.1 Algorithm. Figures 6 and 7 give an algorithm to convert types τ to level-n RDNFs, written $\mathsf{dnf}^n(\tau)$. The task is essentially straightforward, if relatively tedious. Essentially, dnf^n pushes negations in using DeMorgan laws, distributes intersections over unions, and at the same time ensures that all constructed conjunctions are de-duplicated and as reduced as possible, so that for instance intersections of unrelated classes are reduced to $\bot$ and function and record types are merged with themselves. We write $(\neg)\tau$ as a shorthand for either τ or $\neg\tau$ (used uniformly in a rule) and make use of auxiliary functions $\mathsf{union}^n(\mathsf{D}^n, \mathsf{D}^n)$ and $\mathsf{inter}^n(\mathsf{D}^n, \mathsf{D}^n)$, which rely on the following context definitions $S^+[\cdot]$ and $S^-[\cdot]$, used to "dig into" the various shapes of C^n syntaxes:

$$S^+[\square] ::= \mathcal{I}[\square] \mid S^+[\square] \wedge \alpha \mid S^+[\square] \wedge \neg\alpha \mid S^+[\square] \wedge \neg\mathsf{U} \mid S^+[\square] \wedge N[\overline{\mathsf{D}^1}]$$
$$S^-[\square] ::= S^-[\square] \wedge \alpha \mid S^-[\square] \wedge \neg\alpha \mid \mathsf{I} \wedge \neg S^\neg[\square]$$
$$S^\neg[\square] ::= \square \mid S^\neg[\square] \vee N[\overline{\mathsf{D}^1}] \mid S^\neg[\square] \vee \#C \mid \mathsf{U} \vee \square$$

For example, we can decompose $\mathsf{C}^n = \mathsf{I}^n \wedge \neg((\mathsf{D}_1^n \to \mathsf{D}_2^n) \vee \#C) \wedge \alpha$ as $\mathsf{C}^n = S^-[\mathsf{D}_1^n \to \mathsf{D}_2^n]$ where $S^-[\square] = \mathsf{I}^n \wedge \neg(\square \vee \#C) \wedge \alpha$.

The algorithm is well-defined on well-formed types τ ***wf***, assuming a well-formed declarations context $\mathcal{D}$ ***wf***. These notions of well-formedness are defined formally in the extended paper version.

Lemma 5.2 (Well-Defined dnf). *If $\mathcal{D}$ **wf**, τ **wf**, and $n \in \{0, 1\}$, then $\mathsf{dnf}^n(\tau) = \mathsf{D}^n$ for some D^n.*

Lemma 5.3 (Correctness of dnf). *For all τ, $n \in \{0, 1\}$, and $\mathsf{D}^n = \mathsf{dnf}^n(\tau)$, we have $\tau \equiv \mathsf{D}^n$.*

5.3 Type Constraining Rules

The type constraining rules are defined in Figure 8. They are defined for *any* pairs of types and input subtyping contexts, returning an output context containing ***err*** in case the constraining fails. We need ***err*** cases to distinguish an infinite loop in the algorithm from a subtype constraining error, i.e., we want to justify that we have a proper algorithm and not just a semi-algorithm.

In top-level constraining judgments, of the form $\Sigma \vdash \tau \ll \tau \Rightarrow \Xi$, we check whether a subtyping relationship is currently in the assumptions; if not, we extend the set of assumptions with the current constraint (guarded by a $\triangleright$) and call the nested constraining rules with the two sides τ_1 and τ_2 merged into a single $\mathsf{dnf}^0(\tau_1 \wedge \neg\tau_2)$ normal form.[22] Nested constraining judgments have syntax $\Sigma \vdash \mathsf{D}^0 \Rightarrow \Xi$; they implicitly solve the constraint $\mathsf{D}^0 \leqslant \bot$. We can do this because for all τ_1 and τ_2, the subtyping relationship $\Sigma \vdash \tau_1 \leqslant \tau_2$ is formally equivalent to $\Sigma \vdash \tau_1 \wedge \neg\tau_2 \leqslant \bot$. This technique was inspired by Pearce [2013], who also puts constraints into this form to solve subtyping problems involving unions, intersections, and negations. Our constraining rules are deterministic except for C-Var1 and C-Var2. By convention, we always pick C-Var1 in case both can be applied.

[22] The real implementation is a little smarter and does not always put the entire constraint into DNF to avoid needless work in common cases. It also uses a mutable cache to reuse previous computations and avoid exponential blowups [Pierce 2002].

$$\boxed{\mathsf{dnf}^n(\tau)} : \mathsf{D}^n$$

$$\mathsf{dnf}^n(\top) = \mathsf{dnf}^n(\neg\bot) = \top^3 \wedge \neg\bot \tag{1}$$
$$\mathsf{dnf}^n(\bot) = \mathsf{dnf}^n(\neg\top) = \bot \tag{2}$$
$$\mathsf{dnf}^n(\alpha) = \top^3 \wedge \neg\bot \wedge \alpha \tag{3}$$
$$\mathsf{dnf}^n(\#C) = \#C \wedge \top \wedge \top \wedge \neg\bot \tag{4}$$
$$\mathsf{dnf}^n(\tau_1 \to \tau_2) = \top \wedge \mathsf{dnf}^1(\tau_1) \to \mathsf{dnf}^1(\tau_2) \wedge \top \wedge \neg\bot \tag{5}$$
$$\mathsf{dnf}^n(\{\, x : \tau \,\}) = \{\, x : \mathsf{dnf}^1(\tau) \,\} \wedge \top \wedge \top \wedge \neg\bot \tag{6}$$
$$\mathsf{dnf}^0(N[\overline{\tau}]) = \mathsf{dnf}^0(\tau') \qquad \textit{when} \quad N[\overline{\tau}] \ \textbf{\textit{exp.}}\ \tau' \tag{7}$$
$$\mathsf{dnf}^1(N[\overline{\tau}]) = \top^3 \wedge N[\,\overline{\mathsf{dnf}^1(\tau)}\,] \wedge \neg\bot \tag{8}$$
$$\mathsf{dnf}^n(\tau_1 \wedge \tau_2) = \mathsf{inter}(\mathsf{dnf}^n(\tau_1),\ \mathsf{dnf}^n(\tau_2)) \tag{9}$$
$$\mathsf{dnf}^n(\tau_1 \vee \tau_2) = \mathsf{union}(\mathsf{dnf}^n(\tau_1),\ \mathsf{dnf}^n(\tau_2)) \tag{10}$$
$$\mathsf{dnf}^n(\neg\alpha) = \top^3 \wedge \neg\bot \wedge \neg\alpha \tag{11}$$
$$\mathsf{dnf}^n(\neg\#C) = \top^3 \wedge \neg(\bot \vee \#C) \tag{12}$$
$$\mathsf{dnf}^n(\neg\{\, x : \tau \,\}) = \top^3 \wedge \neg\{\, x : \mathsf{dnf}^1(\tau) \,\} \tag{13}$$
$$\mathsf{dnf}^n(\neg(\tau_1 \to \tau_2)) = \top^3 \wedge \neg(\mathsf{dnf}^1(\tau_1) \to \mathsf{dnf}^1(\tau_2)) \tag{14}$$
$$\mathsf{dnf}^0(\neg N[\overline{\tau}]) = \mathsf{dnf}^0(\neg\tau') \qquad \textit{when} \quad N[\overline{\tau}] \ \textbf{\textit{exp.}}\ \tau' \tag{15}$$
$$\mathsf{dnf}^1(\neg N[\overline{\tau}]) = \top^3 \wedge \neg(\bot \vee N[\,\overline{\mathsf{dnf}^1(\tau)}\,]) \tag{16}$$
$$\mathsf{dnf}^n(\neg(\tau_1 \wedge \tau_2)) = \mathsf{union}(\mathsf{dnf}^n(\neg\tau_1),\ \mathsf{dnf}^n(\neg\tau_2)) \tag{17}$$
$$\mathsf{dnf}^n(\neg(\tau_1 \vee \tau_2)) = \mathsf{inter}(\mathsf{dnf}^n(\neg\tau_1),\ \mathsf{dnf}^n(\neg\tau_2)) \tag{18}$$

$$\boxed{\mathsf{union}(\mathsf{D}^n,\ \mathsf{D}^n)} : \mathsf{D}^n$$

$$\mathsf{union}(\mathsf{D}^n,\ \bot) = \mathsf{D}^n \tag{19}$$
$$\mathsf{union}(\mathsf{D}^n,\ \mathsf{C}^n) = \begin{cases} \mathsf{D}^n & \textit{when}\ \mathsf{C}^n \in \mathsf{D}^n \\ \mathsf{D}^n \vee \mathsf{C}^n & \textit{otherwise} \end{cases} \tag{20}$$
$$\mathsf{union}(\mathsf{D}^n_1,\ \mathsf{D}^n_2 \vee \mathsf{C}^n) = \mathsf{union}(\mathsf{union}(\mathsf{D}^n_1,\ \mathsf{C}^n),\ \mathsf{D}^n_2) \tag{21}$$

Fig. 6. Normal form construction algorithm.

Definition 5.4 (Upper and lower bounds). We use the following definitions of lower and upper bounds $lb_\Xi(\alpha)$ and $ub_\Xi(\alpha)$ of a type variable α inside a constraining context Ξ:

$$\boxed{lb_\Xi(\alpha)} : \tau$$

$$lb_{\Xi\cdot\textbf{\textit{err}}}(\alpha) = lb_{\Xi\cdot\triangleright H}(\alpha) = lb_\Xi(\alpha)$$
$$lb_{\Xi\cdot(\tau\leqslant\alpha)}(\alpha) = \tau \vee lb_\Xi(\alpha)$$
$$lb_{\Xi\cdot(\tau\leqslant\beta)}(\alpha) = lb_\Xi(\alpha) \qquad (\alpha \neq \beta)$$
$$lb_{\Xi\cdot(\beta\leqslant\tau)}(\alpha) = lb_\Xi(\alpha)$$
$$lb_\epsilon(\alpha) = \bot$$

$$\boxed{ub_\Xi(\alpha)} : \tau$$

$$ub_{\Xi\cdot\textbf{\textit{err}}}(\alpha) = ub_{\Xi\cdot\triangleright H}(\alpha) = ub_\Xi(\alpha)$$
$$ub_{\Xi\cdot(\tau\leqslant\beta)}(\alpha) = ub_\Xi(\alpha)$$
$$ub_{\Xi\cdot(\alpha\leqslant\tau)}(\alpha) = \tau \wedge ub_\Xi(\alpha)$$
$$ub_{\Xi\cdot(\beta\leqslant\tau)}(\alpha) = ub_\Xi(\alpha) \qquad (\alpha \neq \beta)$$
$$ub_\epsilon(\alpha) = \top$$

Notice how the C-Var1/2 rules solve tricky constraints involving type variables by moving the rest of a type expression to the other side of the inequality, relying on negation types and on the properties of Boolean algebras. Moreover, C-Var1/2 look up the existing bounds of the type variable being constrained and perform a recursive call to ensure that the new bound is consistent with these existing ones. This is required to ensure we only produce consistent output contexts, and it explains why we have to thread constraining contexts throughout all type inference derivations. As part of this recursive call, we extend the subtyping assumptions context with the bound being recorded. For example, C-Var2 recurses with context $\Sigma\cdot(\mathsf{C} \leqslant \alpha)$ instead of just Σ. This is crucial

$$\boxed{\mathsf{inter}(\mathsf{D}^n,\ \mathsf{D}^n)} : \mathsf{D}^n$$

$$\mathsf{inter}(\bot,\ \mathsf{D}^n) = \mathsf{inter}(\mathsf{D}^n,\ \bot) = \bot \tag{22}$$

$$\mathsf{inter}(\mathsf{D}_1^n \vee \mathsf{C}^n,\ \mathsf{D}_2^n) = \mathsf{union}(\mathsf{inter}(\mathsf{D}_1^n,\ \mathsf{D}_2^n),\ \mathsf{inter}(\mathsf{C}^n,\ \mathsf{D}_2^n)) \tag{23}$$

$$\mathsf{inter}(\mathsf{C}_1^n,\ \mathsf{D}^n \vee \mathsf{C}_2^n) = \mathsf{union}(\mathsf{inter}(\mathsf{C}_1^n,\ \mathsf{D}^n),\ \mathsf{inter}(\mathsf{C}_1^n,\ \mathsf{C}_2^n)) \tag{24}$$

$$\boxed{\mathsf{inter}(\mathsf{C}^n \mid \bot,\ \mathsf{C}^n \mid \mathsf{I}^n \mid \neg\mathsf{U}^n)} : \mathsf{C}^n \mid \bot$$

$$\mathsf{inter}(\bot,\ \_) = \bot \tag{25}$$

$$\mathsf{inter}(\mathsf{C}_1^n,\ \mathsf{C}_2^n \wedge (\neg)\alpha) = \begin{cases} \mathsf{inter}(\mathsf{C}_1^n,\ \mathsf{C}_2^n) & \textit{when } (\neg)\alpha \in \mathsf{C}_1^n \\ \mathsf{inter}(\mathsf{C}_1^n \wedge (\neg)\alpha,\ \mathsf{C}_2^n) & \textit{otherwise} \end{cases} \tag{26}$$

$$\mathsf{inter}(\mathsf{C}^n,\ \mathsf{I}^n \wedge \neg\mathsf{U}^n) = \mathsf{inter}(\mathsf{inter}(\mathsf{C}^n,\ \mathsf{I}^n),\ \neg\mathsf{U}^n) \tag{27}$$

$$\mathsf{inter}(\mathsf{C}^1,\ \mathsf{I}^1 \wedge N[\,\overline{\mathsf{D}^1}\,]) = \mathsf{inter}(\mathsf{inter}(\mathsf{C}^1,\ \mathsf{I}^1),\ N[\,\overline{\mathsf{D}^1}\,]) \tag{28}$$

$$\mathsf{inter}(\mathsf{C}^n,\ \mathcal{N} \wedge \mathcal{F} \wedge \mathcal{R}) = \mathsf{inter}(\mathsf{inter}(\mathsf{inter}(\mathsf{C}^n,\ \mathcal{N}),\ \mathcal{F}),\ \mathcal{R}) \tag{29}$$

$$\mathsf{inter}(\mathsf{C}^1,\ \neg(\mathsf{U}^1 \vee N[\,\overline{\mathsf{D}^1}\,])) = \mathsf{inter}(\mathsf{inter}(\mathsf{C}^1,\ \neg\mathsf{U}^1),\ \neg N[\,\overline{\mathsf{D}^1}\,]) \tag{30}$$

$$\mathsf{inter}(\mathsf{C}^n,\ \neg\bot) = \mathsf{C}^n \tag{31}$$

$$\mathsf{inter}(S^-[\mathsf{U}_1^n],\ \neg\mathsf{U}_2^n) = \top^3 \quad \textit{when} \quad (\mathsf{U}_1^n,\ \mathsf{U}_2^n) \in \left\{ \begin{array}{l} (\_ \to \_,\ \{\,x : \_\,\})\,; \\ (\{\,x : \_\,\},\ \_ \to \_)\,; \\ (\{\,x : \_\,\},\ \{\,y^{\neq x} : \_\,\}) \end{array} \right\} \tag{32}$$

$$\mathsf{inter}(S^-[\mathsf{D}_1^1 \to \mathsf{D}_2^1],\ \neg(\mathsf{D}_3^1 \to \mathsf{D}_4^1)) = S^-[\mathsf{inter}(\mathsf{D}_1^1,\ \mathsf{D}_3^1) \to \mathsf{union}(\mathsf{D}_2^1,\ \mathsf{D}_4^1)] \tag{33}$$

$$\mathsf{inter}(S^-[\{\,x : \mathsf{D}_1^1\,\}],\ \neg\{\,x : \mathsf{D}_2^1\,\}) = S^-[\{\,x : \mathsf{union}(\mathsf{D}_1^1,\ \mathsf{D}_2^1)\,\}] \tag{34}$$

$$\mathsf{inter}(S^-[\mathsf{U}_1^n],\ \neg(\mathsf{U}_2^n \vee \#C)) = \begin{cases} \mathsf{inter}(S^-[\mathsf{U}_1^n],\ \neg\mathsf{U}_2^n) & \textit{when } \#C \in \mathsf{U}_1^n \\ \mathsf{inter}(S^-[\mathsf{U}_1^n \vee \#C],\ \neg\mathsf{U}_2^n) & \textit{otherwise} \end{cases} \tag{35}$$

$$\mathsf{inter}(S^-[\bot],\ \neg\mathsf{U}^n) = S^-[\mathsf{U}^n] \tag{36}$$

$$\boxed{\mathsf{inter}(\mathsf{D}^1 \mid \mathsf{C}^1,\ (\neg)N[\,\overline{\mathsf{D}^1}\,])} : \mathsf{D}^1$$

$$\mathsf{inter}(\bot,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) = \bot \tag{37}$$

$$\mathsf{inter}(\mathsf{D}_0^1 \vee \mathsf{C}^1,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) = \mathsf{inter}(\mathsf{D}_0^1,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) \vee \mathsf{inter}(\mathsf{C}^1,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) \tag{38}$$

$$\mathsf{inter}(\mathsf{C}^1 \wedge \alpha,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) = \mathsf{inter}(\mathsf{C}^1,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) \wedge \alpha \tag{39}$$

$$\mathsf{inter}(\mathsf{C}^1 \wedge \neg\alpha,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) = \mathsf{inter}(\mathsf{C}^1,\ (\neg)N[\,\overline{\mathsf{D}^1}\,]) \wedge \neg\alpha \tag{40}$$

$$\mathsf{inter}(\mathsf{I}^1 \wedge \neg\mathsf{U}^1,\ N[\,\overline{\mathsf{D}^1}\,]) = \begin{cases} \mathsf{I}^1 \wedge \neg\mathsf{U}^1 & \textit{when } N[\,\overline{\mathsf{D}^1}\,] \in \mathsf{I}^1 \\ \mathsf{I}^1 \wedge N[\,\overline{\mathsf{D}^1}\,] \wedge \neg\mathsf{U}^1 & \textit{otherwise} \end{cases} \tag{41}$$

$$\mathsf{inter}(\mathsf{I}^1 \wedge \neg\mathsf{U}^1,\ \neg N[\,\overline{\mathsf{D}^1}\,]) = \begin{cases} \mathsf{I}^1 \wedge \neg\mathsf{U}^1 & \textit{when } N[\,\overline{\mathsf{D}^1}\,] \in \mathsf{U}^1 \\ \mathsf{I}^1 \wedge \neg(\mathsf{U}^1 \vee N[\,\overline{\mathsf{D}^1}\,]) & \textit{otherwise} \end{cases} \tag{42}$$

$$\boxed{\mathsf{inter}(\mathsf{C}^n,\ \mathcal{N} \mid \mathcal{F} \mid \mathcal{R})} : \mathsf{C}^n \mid \bot$$

$$\mathsf{inter}(\mathsf{C}^n,\ \top) = \mathsf{C}^n \tag{43}$$

$$\mathsf{inter}(S^+[\mathcal{I}^{\mathcal{N}}[\top]],\ \#C) = S^+[\mathcal{I}^{\mathcal{N}}[\#C]] \tag{44}$$

$$\mathsf{inter}(S^+[\mathcal{I}[\#C_1]],\ \#C_2) = \begin{cases} \bot & \textit{when } C_1 \notin \mathcal{S}(\#C_2) \textit{ and } C_2 \notin \mathcal{S}(\#C_1) \\ S^+[\mathcal{I}[\#C_2]] & \textit{when } C_1 \in \mathcal{S}(\#C_2) \\ S^+[\mathcal{I}[\#C_1]] & \textit{when } C_2 \in \mathcal{S}(\#C_1) \end{cases} \tag{45}$$

$$\mathsf{inter}(S^+[\mathcal{I}^{\to}[\top]],\ \mathsf{D}_1^1 \to \mathsf{D}_2^1) = S^+[\mathcal{I}^{\to}[\mathsf{D}_1^1 \to \mathsf{D}_2^1]] \tag{46}$$

$$\mathsf{inter}(S^+[\mathcal{I}[\mathsf{D}_1^1 \to \mathsf{D}_2^1]],\ \mathsf{D}_3^1 \to \mathsf{D}_4^1) = S^+[\mathcal{I}[\mathsf{union}(\mathsf{D}_1^1,\ \mathsf{D}_3^1) \to \mathsf{inter}(\mathsf{D}_2^1,\ \mathsf{D}_4^1)]] \tag{47}$$

$$\mathsf{inter}(\mathsf{C}^n,\ \{\,x : \mathsf{D}_x^1,\ \overline{y : \mathsf{D}_y^1}\,\}) = \mathsf{inter}(\mathsf{inter}(\mathsf{C}^n,\ \{\,x : \mathsf{D}_x^1\,\}),\ \{\,\overline{y : \mathsf{D}_y^1}\,\}) \tag{48}$$

$$\mathsf{inter}(S^+[\mathcal{I}^{\{\}}[\top]],\ \{\,x : \mathsf{D}^1\,\}) = S^+[\mathcal{I}^{\{\}}[\{\,x : \mathsf{D}^1\,\}]] \tag{49}$$

$$\mathsf{inter}(S^+[\mathcal{I}[\{\,\overline{x : \mathsf{D}_x^1}^{\,x \in S}\,\}]],\ \{\,y : \mathsf{D}^1\,\}) = \begin{cases} S^+[\mathcal{I}[\{\,\overline{x : \mathsf{D}_x^1}^{\,x \in S \setminus \{y\}},\ y : \mathsf{inter}(\mathsf{D}_y^1,\ \mathsf{D}^1)\,\}]] & \textit{when } y \in S \\ S^+[\mathcal{I}[\{\,\overline{x : \mathsf{D}_x^1}^{\,x \in S},\ y : \mathsf{D}^1\,\}]] & \textit{otherwise} \end{cases} \tag{50}$$

Fig. 7. Normal form construction algorithm (continued).

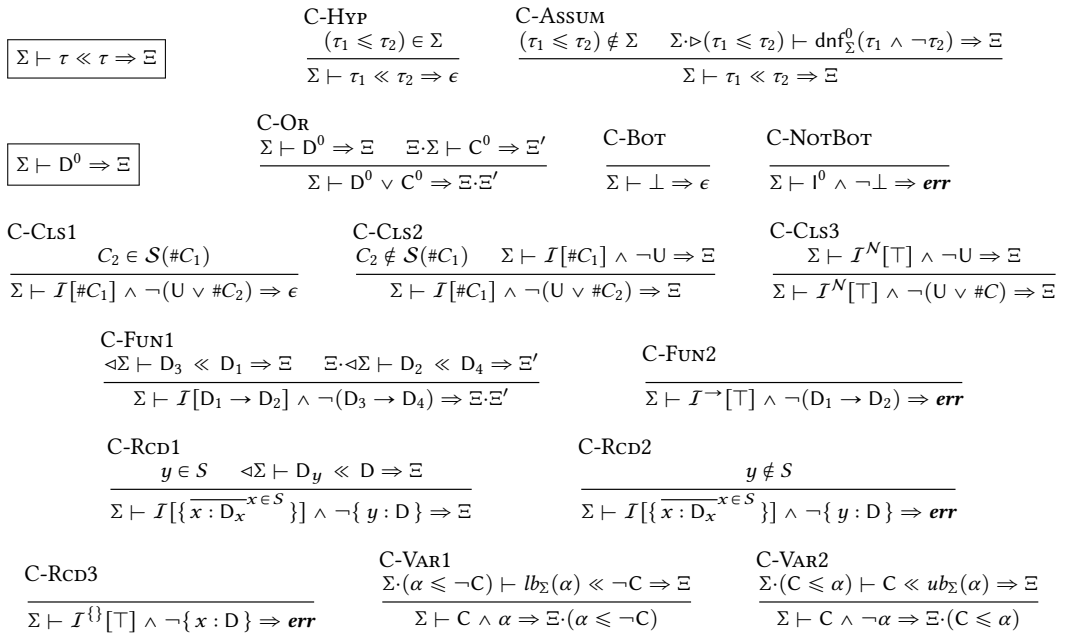

Fig. 8. Normal form constraining rules.

for two reasons: First, it is possible that new upper bounds τ_i be recorded for α as part of the recursive call. By adding C to the current lower bounds of α within the recursive call, we make sure that any such new upper bounds τ_i will be checked against C as part of the resulting $\mathit{lb}_\Sigma(\alpha) \ll \tau_i$ constraining call performed when adding bound τ_i. Second, it is quite common for type inference to result in direct type variable bound cycles, such as $\alpha \leqslant \beta$, $\beta \leqslant \alpha$, which can for instance arise from constraining $\beta \to \beta \leqslant \alpha \to \alpha$. These cycles do not lead to divergence of type inference thanks to the use of $\Sigma{\cdot}(\mathsf{C} \leqslant \alpha)$ instead of Σ in the recursive call, ensuring that any constraint resulting from a type variable bound cycle will end up being caught by C-Hyp.

The other constraining rules are fairly straightforward. The "beauty" of the RDNF is that it essentially makes constraint solving with $\lambda^\neg$ types obvious. In each case, there is always an obvious choice to make: either (1) the constraint is unsatisfiable (for example with $\top \leqslant \bot$ in C-NotBot, which yields an ***err***); or (2) the constraint needs to unwrap an irrelevant part of the type to continue (for example with $\mathsf{D}_1 \to \mathsf{D}_2 \leqslant \mathsf{U} \vee \#C$ in C-Cls3, which can be solved iff $\mathsf{D}_1 \to \mathsf{D}_2 \leqslant \mathsf{U}$ itself can be solved, because function types are unrelated to nominal class tags); or (3) we can solve the constraint in an obvious, unambiguous way (for example with $\{\overline{x : \mathsf{D}_x}^{x \in S}\} \leqslant \{ y : \mathsf{D} \}$ where $y \in S$ in C-Rcd1).

Normalizing types deeply (i.e., not solely on the outermost level) makes the termination of constraining straightforward. If we did not normalize nested types and for example merged $\{ x : \tau_1 \} \wedge \{ x : \tau_2 \}$ syntactically as $\{ x : \tau_1 \wedge \tau_2 \}$, constraining recursive types in a way that repetitively merges the same type constructors together could lead to unbounded numbers of equivalent types being constrained, such as $\{ x : \tau_1 \wedge \tau_1 \wedge \tau_1 \wedge \dots \}$, failing to terminate by C-Hyp.

Example. Consider the constraint $\tau = \{ x : \mathsf{Nat},\ y : \mathsf{Nat} \} \ll \pi = \{ x : \mathsf{Int},\ y : \top \}$. After adding the pair to the set of hypotheses, C-Assum computes the RDNF $\mathsf{dnf}^0(\tau \wedge \neg\pi) = \{ x : \mathsf{Nat},\ y : \mathsf{Nat} \} \wedge \neg\{ x : \mathsf{Int} \} \vee \{ x : \mathsf{Nat},\ y : \mathsf{Nat} \} \wedge \neg\{ y : \top \}$. Then this constrained type is decomposed into

two smaller constrained types $\{ x : \mathsf{Nat},\ y : \mathsf{Nat} \} \wedge \neg\{ x : \mathsf{Int} \}$ and $\{ x : \mathsf{Nat},\ y : \mathsf{Nat} \} \wedge \neg\{ y : \top \}$ by C-Or, and each one is solved individually by C-Rcd1, which requires constraining respectively $\mathsf{Nat} \ll \mathsf{Int}$ and $\mathsf{Nat} \ll \top$. The former yields RDNF $\#\mathsf{Nat} \wedge \neg\#\mathsf{Int}$, which is solved by C-ClsCls1, and the latter yields RDNF $\bot$, which is solved by C-Bot.

5.4 Correctness of Type Inference

We conclude this section by presenting the main correctness lemmas and theorems of type inference.

Theorem 5.5 (Soundness of type inference). *If the type inference algorithm successfully yields a type for program P, then P has this type. Formally: if $\Vdash^\star P : \tau \Rightarrow \Xi$ and $\mathbf{err} \notin \Xi$, then $\Xi \vdash^\star P : \tau$.*

Lemma 5.6 (Sufficiency of Constraining). *Successful type constraining ensures subtyping: if Σ **cons.** and $\Sigma \vdash \tau \ll \pi \Rightarrow \Xi$ and $\mathbf{err} \notin \Xi$, then $\Xi{\cdot}\Sigma$ **cons.** and $\Xi{\cdot}\Sigma \vdash \tau \leqslant \pi$.*

Theorem 5.7 (Constraining Termination). *For all $\tau, \pi, \mathcal{D}, \Sigma$ **wf**, $\Sigma \vdash \tau \ll \pi \Rightarrow \Xi$ for some Ξ.*

Theorem 5.8 (Completeness of type inference). *If a program P can be typed at type σ, then the type inference algorithm derives a type σ' such that $\sigma' \leqslant^\forall \sigma$. Formally: if $\Xi \vdash^\star P : \tau$, then $\Vdash^\star P : \tau' \Rightarrow \Xi'$ for some Ξ' and τ' where Ξ' **cons.** and $\forall \Xi'.\, \tau' \leqslant^\forall \forall \Xi.\, \tau$.*

In the following lemma, which is crucial for proving the above theorem, ρ refers to type variable substitutions and $\Xi \models \Xi'$ denotes that Ξ entails Ξ' (both defined formally in the extended version).

Lemma 5.9 (Completeness of Constraining). *If there is a substitution ρ that makes $\rho(\tau_1)$ a subtype of $\rho(\tau_2)$ in some consistent Ξ, then constraining $\tau_1 \ll \tau_2$ succeeds and only introduces type variable bounds that are entailed by Ξ (modulo ρ). Formally: if Ξ **cons.** and $\Xi \vdash \rho(\tau_1) \leqslant \rho(\tau_2)$ and $\Xi \models \rho(\Xi_0)$, then $\Xi_0 \vdash \tau_1 \ll \tau_2 \Rightarrow \Xi_1$ for some Ξ_1 so that $\mathbf{err} \notin \Xi_1$ and $\Xi \models \rho(\Xi_1)$.*

6 RELATED WORK

We now relate the different aspects of MLstruct and $\lambda^\neg$ with previous work.

Intersection type systems. Intersection types for lambda calculus were pioneered by Coppo and Dezani-Ciancaglini [1980]; Barendregt et al. [1983], after whom the "BCD" type system is named. BCD has the very powerful "T-$\wedge$-I" rule, stating: if $\Gamma \vdash t : \tau_1$ and $\Gamma \vdash t : \tau_2$, then $\Gamma \vdash t : \tau_1 \wedge \tau_2$. Such systems have the interesting property that typeability coincides with strong normalization [Ghilezan 1996], making type inference undecidable. Thankfully, we do not need something as powerful as T-$\wedge$-I — instead, we introduce intersections in less general ways (i.e., through T-Obj), and we retain decidability of type inference. Most intersection type systems, including MLstruct and $\lambda^\neg$, do admit the following standard BCD subtyping rules given by Barendregt et al.: (1) $\tau_1 \wedge \tau_2 \leqslant \tau_1$; (2) $\tau_1 \wedge \tau_2 \leqslant \tau_2$; and (3) if $\tau_1 \leqslant \tau_2$ and $\tau_1 \leqslant \tau_3$, then $\tau_1 \leqslant \tau_2 \wedge \tau_3$. Some systems use intersection types to encode a form of overloading [Pierce 1991]. However, Smith [1991] showed that ML-style type inference with such a general form of overloading and subtyping is undecidable (more specifically, finding whether inferred sets of constraints are satisfiable is) and proposed *constructor overloading*, a restricted form of overloading with more tractable properties, sufficient to encode many common functions, such as addition on different primitive types as well as vectors of those types. Constructor overloading is eminently compatible with MLstruct and MLscript. Another design decision for intersection systems is whether and how this connective should *distribute* over function types. BCD subtyping states[23] $(\tau \to \pi_1) \wedge (\tau \to \pi_2) \leqslant \tau \to (\pi_1 \wedge \pi_2)$ and Barbanera et al. [1995] also propose $(\tau_1 \to \pi) \wedge (\tau_2 \to \pi) \leqslant (\tau_1 \vee \tau_2) \to \pi$. Together, these correspond to the minimal relevant logic B+ [Dezani-Ciancaglini et al. 1998]. Approaches like that of Pottier [1998b] use a *greatest*

[23] This rule together with T-$\wedge$-I was shown unsound in the presence of imperative features by Davies and Pfenning [2000].

lower bound connective $\sqcap$ that resembles type intersection $\wedge$ but admits a more liberal rule that generalizes the previous two: $(\tau_1 \to \pi_1) \wedge (\tau_2 \to \pi_2) \leqslant (\tau_1 \vee \tau_2) \to (\pi_1 \wedge \pi_2)$, which we will refer to as *(full) function distributivity*. However, notice that in a system with primitives, full function distributivity is incompatible with T-$\wedge$-I and thus precludes intersection-based overloading.[24]

Union and intersection types in programming. Union types are almost as old as intersection types, first introduced by MacQueen et al. [1986],[25] and both have a vast (and largely overlapping) research literature, with popular applications such as refinement types [Freeman and Pfenning 1991]. These types have seen a recent resurgence, gaining a lot of traction both in academia [Alpuim et al. 2017; Binder et al. 2022; Castagna et al. 2022; Dunfield 2012; Huang and Oliveira 2021; Muehlboeck and Tate 2018; Rehman et al. 2022] and in industry,[26] with several industry-grade programming languages like TypeScript, Flow, and Scala 3 supporting them, in addition to a myriad of lesser-known research languages.

Type inference for unions and intersections. None of the previous approaches we know have proposed a satisfactory ML-style type inference algorithm for full union and intersection types. By *satisfactory*, we mean that the algorithm should infer principal polymorphic types without backtracking. Earlier approaches used heavily-restricted forms of unions and intersections. For instance, Aiken and Wimmers [1993]; Aiken et al. [1994] impose very strict restrictions on negative unions (they must be disjoint) and on positive intersections (they must not have free variables and must be "upward closed"). Trifonov and Smith [1996] go further and restrict intersections to *negative* or *input* positions (those appearing on the right of $\leqslant$ constraints) and unions types to *positive* or *output* positions (those appearing on the left). Binder et al. [2022]; Dolan [2017]; Parreaux [2020]; Pottier [1998b] all follow the same idea. In these systems, unions and intersections are not *first-class* citizens: they cannot be used freely in type annotations. Frisch et al. [2008] infer set-theoretic types (see *semantic subtyping* below) for a higher-order language with overloading but do not infer polymorphic types. Castagna et al. [2016] propose a complete polymorphic set-theoretic type inference system, but their types are not principal so their algorithm returns several solutions, which leads to the need for backtracking. It seems this should have severe scalability issues, as the number of possible types for an expression would commonly grow exponentially.[27] Petrucciani [2019] describes ways to reduce backtracking, but recognizes it as fundamentally "unavoidable."

Negation or complement types. Negation types have not been nearly as ubiquitous as unions and intersection in mainstream programming language practice and theory, except in the field of *semantic subtyping* (see below). Nevertheless, our use of negation types to make progress while solving constraints is not new — Aiken and Wimmers [1993] were the first to propose using complement types in such a way. However, their complement types are less precise than our negation types,[28] and in their system $\alpha \wedge \tau_1 \leqslant \tau_2$ and $\alpha \leqslant \tau_2 \vee \neg\tau_1$ are *not* always equivalent.

Recursive types. Recursive types in the style of MLstruct, where a recursive type is *equivalent* to its unfolding (a.k.a. *equi-recursive* types, not to be confused with *iso-recursive* types), have a

[24] For instance, term $id = \lambda x.\ x$ has both types Int $\to$ Int and Bool $\to$ Bool so by T-$\wedge$-I it would also have type (Int $\to$ Int) $\wedge$ (Bool $\to$ Bool). But by function distributivity and subsumption, this would allow typing id as (Int $\vee$ Bool) $\to$ (Int $\wedge$ Bool) and thus typing id 0 (which reduces to 0) as Int $\wedge$ Bool, breaking type preservation.

[25] Funnily, MacQueen et al. reported at the time that "*type-checking difficulties seem to make intersection and union awkward in practice; moreover it is not clear if there are any potential benefits from their use*."

[26] The first author of this paper has received emails from various people reimplementing Simple-sub [Parreaux 2020] and wanting to know how to add support for first-class union and intersection types, showing the enduring interest in these.

[27] Hindley-Milner type inference and derived systems like MLsub and MLstruct can also infer types that grow exponentially in some situations, but these mostly occur in pathological cases, and not in common human-written programs.

[28] For example, in their system $\neg(\tau \to \pi)$ is the type of all values that are not functions, *regardless* of τ and π.

long history in programming languages research [Abadi and Fiore 1996; Amadio and Cardelli 1993; Appel et al. 2007; Hosoya et al. 2005; MacQueen et al. 1986; Pierce 2002], dating as far back as Morris' thesis, where he conjectured their use under the name of *cyclic types* [Morris 1969, pp.122–124]. Recursive types with subtyping were developed in the foundational work of Amadio and Cardelli [1993] and Brandt and Henglein [1998] gave a coinductive axiomatization of such recursive types. Jim and Palsberg [1999] described a co-inductive formalization of recursive types as arbitrary infinite trees which is more general than approaches like ours, which only allows reasoning about *regular* types. Nevertheless, the algorithms they gave were unsurprisingly restricted to regular types. Gapeyev et al. [2002]; Pierce [2002] reconciled the representation as infinite regular trees with the representation as μ types, and described the standard algorithms to decide the corresponding subtyping relationship. An important aspect of practical recursive type algorithms is that one needs to maintain the cache of discovered subtyping relationships *across recursive calls* to avoid exponential blowup [Gapeyev et al. 2002]. Our implementation of MLstruct follows the same principle, as a naive implementation of $\lambda^{\neg}$ would lead to exactly the same blowup. Also refer to Section 4.4.2 for more parallels between the handling of recursive types in $\lambda^{\neg}$ and previous work.

Early approaches to subtype inference. The problem of type inference in the presence of subtyping was kick-started in the 1980s [Fuh and Mishra 1989; Mitchell 1984; Stansifer 1988] and studied extensively in the 1990s [Aiken and Wimmers 1993; Curtis 1990; Fuh and Mishra 1990; Jim and Palsberg 1999; Kozen et al. 1994; Palsberg et al. 1997; Pottier 1998a,b; Smith 1991], mostly through the lens of constraint solving on top of Hindley-Milner-style type inference [Damas and Milner 1982; Hindley 1969; Milner 1978]. These approaches often involved combinations of record, intersection, union, and recursive types, but as far as we know none proposed an effective (i.e., without backtracking) principal type inference technique for a system with all of these combined. Odersky et al. [1999] gave them a unified account by proposing a general framework called HM(X), where the 'X' stands for a constraint solver to plug into their generic system. While these approaches often claimed a form of principal type inference (also called *minimality*[29]), the *constrained types* they inferred were often large and unwieldy. Beyond inferring constraint sets and ensuring their satisfiability, the related problem of *simplification* to produce more readable and efficiently-processable types was also studied, often by leveraging the connection between regular type trees and finite-state automata [Aiken 1996; Eifrig et al. 1995; Pottier 1996, 1998b, 2001; Simonet 2003]. A major stumbling block with all of these approaches was the problem of *non-structural subtyping entailment*[30] (NSSE), which is to decide whether a given type scheme, which consists in a polymorphic type along with its constraints on type variables, subsumes another. Solving this issue is of central importance because it is needed to check implementations against user-provided interfaces and type signatures, and because it provides a foundation from which to derive sound type simplification techniques. However, to this day NSSE remains an open problem, and it is not known whether it is even decidable [Dolan 2017]. Due to these difficulties, interest in this very powerful form of subtyping all but faded in the subsequent decade, in what we interpret as a minor "subtype inference winter." Indeed, many subsequent approaches were developed in reaction to this complexity with the aim of being simpler to reason about (e.g., polymorphic variants — see below).

Algebraic subtyping. Approaches like that of Pottier [1998b] used a lattice-theoretic construction of types inspired by the connection between types and term automata. Meet $\sqcap$ and join $\sqcup$ operators resembling intersection and union types are used to compactly representing conjunctions

[29] Some authors like Aiken et al. [1994] make a distinction between a concept of *principality* which is purely syntactic (relating types by a substitution instance relationship) and *minimality* which involve a *semantic* interpretation of types.
[30] "*Non-structural*" here is by opposition to so-called *structural subtyping*, which is a more tractable but heavily restricted form of subtyping that only relates type constructors of identical arities [Palsberg et al. 1997] (precluding, e.g., $\{ x : \tau \} \leqslant \top$).

of constraints, but these are not *first-class* types, in that they are restricted to appearing respectively in negative and positive positions only. Full function distributivity (defined above, in *intersection type systems*) holds in these approaches due to the lattice structure. Pottier's system still suffered from a lack of complete entailment algorithm due to NSSE. Dolan [2017]; Dolan and Mycroft [2017] later built upon that foundation and proposed an *algebraic* construction of types which allowed breaking free of NSSE and finally enjoying a sound and complete entailment algorithm. Two magical ingredients allowed this to be possible: 1. the definition of "extensible" type semantics based on constructing types as a distributive lattice of coproducts; and 2. a different treatment of type variables than in previous work, representing them as part of the lattice of types and not as unknowns ranging over a set of ground types. In this paper, we in turn build on these foundations, although we only retain the latter innovation, somehow forgoing the "extensible" construction of types.[31] Together with our generalization of the subtyping lattice to a Boolean one by adding negations and with the additional structure we impose on types (such as reducing unions of unrelated records to $\top$), this turns out to be sufficient for allowing principal type inference and decidable entailment (though we only sketched the latter in this paper for lack of space). Ingredient 1 allowed Dolan to show the soundness of his system in a very straightforward way, relying on the property (called Proposition 12 by Dolan [2017]) that any constraint of the form $\bigwedge_i \tau_i \leqslant \bigvee_i \pi_i$ holds iff there is a k such that $\tau_k \leqslant \pi_k$ when all τ_i have distinct constructors and all π_i similarly. By contrast, we allow some intersections of unrelated type constructors to reduce to $\bot$ and some unions of them to $\top$, and we are thus not "extensible" in Dolan's terminology. This is actually desirable in the context of pattern matching, where we want to eliminate impossible cases by making the intersections of unrelated class types empty. It is also needed in order to remove the ambiguity from constraints like $(\tau_1 \to \tau_2) \wedge \{ x : \pi \} \leqslant (\tau_1' \to \tau_2') \vee \{ x : \pi' \}$ which in our system reduces to $(\tau_1 \to \tau_2) \wedge \{ x : \pi \} \leqslant \top$. The present paper also takes heavy inspiration from our earlier operationally-focused take on Dolan's type inference algorithm [Parreaux 2020]. While Dolan shirks from explicitly representing constraints, which he prefers to inline inside types on the fly as $\sqcap$ and $\sqcup$ types, we use an approach closer to the original constrained-types formulation followed by Pottier. Besides being much easier to implement, our approach has other concrete advantages, such as the ability to deal with invariance seamlessly (`class C[A]: {f: A → A}`, which is invariant in `A`, is valid in MLstruct) and a simpler treatment of cyclic type variable constraints.

Semantic subtyping and set-theoretic types. The *semantic subtyping* approaches [Castagna et al. 2022, 2016; Frisch et al. 2002, 2008; Petrucciani 2019] view types as sets of values which inhabit them and define the subtyping relationship as set inclusion, giving set-based meaning to *union*, *intersection*, and *negation* (or *complement*) connectives. This is by contrast to algebraic subtyping, which may admit subtyping rules that *violate* the set-theoretic interpretation, such as function distributivity, to ensure that the subtyping lattice has desirable algebraic properties. For more detailed discussions contrasting semantic subtyping with other approaches, we refer the reader to Parreaux [2020] and Muehlboeck and Tate [2018].

Occurrence and flow typing. Occurrence typing was originally introduced by Tobin-Hochstadt and Felleisen [2008] for Typed Scheme, and was later incorporated into TypeScript and Flow, where it is known as *flow typing*. It allows the types of variables to be locally refined based on path conditions encountered in the program. Instance-matching in MLstruct can be understood as a primitive form of occurrence typing in that it refines the types of scrutinee variables in **`case`** expressions, similarly to the approach of Rehman et al. [2022]. Occurrence typing was also recently extended to the semantic subtyping context [Castagna et al. 2021, 2022], where negation types

[31] As discussed in prior work [Parreaux 2020], we believe the argument for Dolan's notion of extensibility to be rather weak.

are first-class types. The latter work proposes a powerful type inference approach that can infer overloaded function signatures as intersections types; however, this approach does not support polymorphism and likely does not admit principal types.

Polymorphic records/variants and row polymorphism. Polymorphic *records* are structurally-typed products whose types admit the usual width and depth subtyping relationships. Their dual, polymorphic *variants*, are another useful language feature [Garrigue 1998, 2001], used to encode structural sum types. In their simplest expression, polymorphic records (resp. variants) do not support ad-hoc field extension (resp. default match cases). Previous approaches have thus extended polymorphic records and variants with *row polymorphism*, which uses a new kind of variables, named "row" variables, to record the presence and absence of fields (resp. cases) in a given type. Some approaches, like OCaml's polymorphic variants and object types, use row polymorphism exclusively to *simulate* subtype polymorphism, in order to avoid subtyping in the wider languages. However, row polymorphism and subtyping actually complement each other well, and neither is as flexible without the other [Pottier 1998b, Chapter 14.7]. There are also techniques for supporting variant and record extensibility through union, intersection, and negation types, as shown by Castagna et al. [2016], who also explain that their system resolves long-standing limitations with OCaml-style row polymorphism. In our system, we solve many (though not all) of these limitations, but we also support principal type inference. It is worth pointing out that OCaml's polymorphic variants [Garrigue 2001] and related systems based on kinds [Ohori 1995] lack support for *polymorphic extension* [Gaster and Jones 1996; White 2015], whereas MLstruct does (see `mapSome` in the introduction). As a simpler example, `def foo x dflt els = case x of { Apple → dflt | _ → els x }` would be assigned a too restrictive type in OCaml and as a consequence `foo (Banana{}) 0 (fun z → case z of { Banana → 1 })` would not type check (OCaml would complains that the function argument does not handle `Apple`). A more expressive row-polymorphic system exposing row variables to users would support this use case [Gaster and Jones 1996; Rémy 1994], but as explained in the introduction, even these have limitations compared to our subtyped unions.

7 CONCLUSION AND FUTURE WORK

In this paper, we saw that polymorphic type inference for first-class union, intersection, and negation types was possible, and that it enabled advanced instance-matching patterns yielding very precise types comparable in expressiveness to row-polymorphic variants. We saw that this type inference approach crucially relies on two crucial aspects of MLstruct's type system: 1. using the full power of Boolean algebras to normalize types and massage constraints into shapes amenable to constraint solving without backtracking; and 2. approximating some unions and intersections, most notably unions of records and intersections of functions, in order to remove potential ambiguities during constraint solving without threatening the soundness of the system.

Future Work. In the future, we intend to explore more advanced forms of polymorphism present in MLscript, such as first-class polymorphism, as well as how to remove some of the limitations of regular types, which currently prevent fully supporting object-oriented programming idioms.

Acknowledgements. We would like to sincerely thank the anonymous reviewers as well as François Pottier, Didier Rémy, Alan Mycroft, Bruno C. d. S. Oliveira, and Andong Fan for their constructive and helpful comments on earlier versions of this paper. We are particularly grateful to Stephen Dolan, who gave us some invaluable feedback and mathematical intuitions on the development of this new algebraic subtyping system.

REFERENCES

Martin Abadi and Marcelo P. Fiore. 1996. Syntactic considerations on recursive types. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 242–252. ↪ page 25

Alexander Aiken. 1996. Making set-constraint program analyses scale. In *In CP96 Workshop on Set Constraints*. ↪ page 25

Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 31–41. https://doi.org/10.1145/165180.165188 ↪ pages 9, 24, and 25

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 163–173. https://doi.org/10.1145/174675.177847 ↪ pages 24 and 25

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28. ↪ page 24

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.* 15, 4 (Sept. 1993), 575–631. https://doi.org/10.1145/155183.155231 ↪ pages 15 and 25

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 109–122. https://doi.org/10.1145/1190216.1190235 ↪ pages 16 and 25

F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (1995), 202–230. https://doi.org/10.1006/inco.1995.1086 ↪ page 23

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic* 48, 4 (1983), 931–940. https://doi.org/10.2307/2273659 ↪ page 23

David Binder, Ingo Skupin, David Läwen, and Klaus Ostermann. 2022. Structural Refinement Types. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3546196.3550163 ↪ page 24

Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338. ↪ page 25

Giuseppe Castagna. 2012. *Object-Oriented Programming A Unified Foundation*. Springer Science & Business Media. ↪ page 16

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2021. Revisiting Occurrence Typing. arXiv:1907.05590 [cs.PL] ↪ page 26

Giuseppe Castagna, Mickaël Laurent, Kim Nguyundefinedn, and Matthew Lutze. 2022. On Type-Cases, Union Elimination, and Occurrence Typing. *Proc. ACM Program. Lang.* 6, POPL, Article 13 (jan 2022), 31 pages. https://doi.org/10.1145/3498674 ↪ pages 24 and 26

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, Nara, Japan, 378–391. https://doi.org/10.1145/2951913.2951928 ↪ pages 24, 26, and 27

M. Coppo and M. Dezani-Ciancaglini. 1980. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic* 21, 4 (1980), 685 – 693. https://doi.org/10.1305/ndjfl/1093883253 ↪ page 23

Pavel Curtis. 1990. *Constrained Qualification in Polymorphic Type Analysis*. Ph.D. Dissertation. USA. UMI Order No. GAX90-26980. ↪ page 25

Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman. 2020. The Duality of Subtyping. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.29 ↪ page 11

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '82)*. Association for Computing Machinery, Albuquerque, New Mexico, 207–212. https://doi.org/10.1145/582153.582176 ↪ page 25

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/351240.351259 ↪ page 23

Van Bakel Dezani-Ciancaglini, S. Van Bakel, M. Dezani-ciancaglini, and Y. Motohama. 1998. *The Minimal Relevant Logic and the Call-by-Value Lambda Calculus*. Technical Report. ↪ page 23

Stephen Dolan. 2017. *Algebraic subtyping*. Ph.D. Dissertation. ↪ pages 3, 4, 6, 12, 15, 16, 24, 25, and 26

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 60–72. https://doi.org/10.1145/3093333.3009882 ↪ pages 1 and 26

Jana Dunfield. 2012. Elaborating Intersection and Union Types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 17–28. https://doi.org/10.1145/2364527.2364534 ↪ page 24

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995. Sound Polymorphic Type Inference for Objects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (Austin, Texas, USA) *(OOPSLA '95)*. Association for Computing Machinery, New York, NY, USA, 169–184. https://doi.org/10.1145/217838.217858 ↪ page 25

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468 event-place: Toronto, Ontario, Canada. ↪ page 24

A. Frisch, G. Castagna, and V. Benzaken. 2002. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 137–146. https://doi.org/10.1109/LICS.2002.1029823 ↪ page 26

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages. https://doi.org/10.1145/1391289.1391293 ↪ pages 24 and 26

You-Chin Fuh and Prateek Mishra. 1989. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT '89*, J. Díaz and F. Orejas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–183. ↪ page 25

You-Chin Fuh and Prateek Mishra. 1990. Type inference with subtypes. *Theoretical Computer Science* 73, 2 (1990), 155–175. https://doi.org/10.1016/0304-3975(90)90144-7 ↪ page 25

Vladimir Gapeyev, Michael Y Levin, and Benjamin C Pierce. 2002. Recursive subtyping revealed. *Journal of Functional Programming* 12, 6 (2002), 511–548. ↪ page 25

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*, Vol. 13. Baltimore, 7. ↪ page 27

Jacques Garrigue. 2001. Simple Type Inference for Structural Polymorphism.. In *APLAS*. 329–343. ↪ pages 3 and 27

Benedict R. Gaster and Mark P. Jones. 1996. A Polymorphic Type System for Extensible Records and Variants. ↪ page 27

Silvia Ghilezan. 1996. Strong Normalization and Typability with Intersection Types. *Notre Dame Journal of Formal Logic* 37, 1 (1996), 44 – 52. https://doi.org/10.1305/ndjfl/1040067315 ↪ page 23

Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. https://doi.org/10.2307/1995158 Publisher: American Mathematical Society. ↪ page 25

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.* 27, 1 (Jan. 2005), 46–90. https://doi.org/10.1145/1053468.1053470 ↪ pages 16 and 25

Xuejing Huang and Bruno C. d. S. Oliveira. 2021. Distributing Intersection and Union Types with Splits and Duality (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 89 (aug 2021), 24 pages. https://doi.org/10.1145/3473594 ↪ page 24

Trevor Jim and Jens Palsberg. 1999. Type Inference in Systems of Recursive Types With Subtyping. ↪ page 25

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1994. Efficient inference of partial types. *J. Comput. System Sci.* 49, 2 (1994), 306–324. https://doi.org/10.1016/S0022-0000(05)80051-0 ↪ page 25

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1986. An ideal model for recursive polymorphic types. *Information and Control* 71, 1 (1986), 95–130. https://doi.org/10.1016/S0019-9958(86)80019-5 ↪ pages 24 and 25

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4 ↪ page 25

John C. Mitchell. 1984. Coercion and Type Inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/800017.800529 ↪ page 25

James Hiram Morris. 1969. *Lambda-calculus models of programming languages.* Ph.D. Dissertation. Massachusetts Institute of Technology. ↪ page 25

Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 112 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276482 ↪ pages 4, 6, 24, and 26

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55. ↪ page 25

Atsushi Ohori. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.* 17, 6 (nov 1995), 844–895. https://doi.org/10.1145/218570.218572 ↪ pages 3 and 27

Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. 1997. Type inference with non-structural subtyping. *Formal Aspects of Computing* 9, 1 (Jan. 1997), 49–67. https://doi.org/10.1007/BF01212524 ↪ page 25

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. https://doi.org/10.1145/3409006 ↪ pages 9, 11, 12, 24, and 26

Lionel Parreaux and Chun Yin Chau. 2022. *MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types (Extended Version).* Technical Report. The Hong Kong University of Science and Technology. https://lptk.github.io/mlscript-paper ↪ pages 3 and 10

Lionel Parreaux, Luyu Cheng, Tony Chau, Ishan Bhanuka, Andong Fan, Malcolm Law, Ali Mahzoun, and Elise Rouillé. 2022. *MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types (Artifact).* https://doi.org/10.5281/zenodo.7121838 ↪ pages 3 and 4

David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, Berlin, Heidelberg, 335–354. https://doi.org/10.1007/978-3-642-35873-9_21 ↪ page 19

Tommaso Petrucciani. 2019. *Polymorphic set-theoretic types for functional languages.* Ph.D. Dissertation. Università di Genova; Université Sorbonne Paris Cité – Université Paris Diderot. ↪ pages 24 and 26

Benjamin C Pierce. 1991. *Programming with intersection types and bounded polymorphism.* Ph.D. Dissertation. Citeseer. ↪ page 23

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT press. ↪ pages 15, 19, and 25

François Pottier. 1996. Simplifying Subtyping Constraints. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. Association for Computing Machinery, New York, NY, USA, 122–133. https://doi.org/10.1145/232627.232642 ↪ page 25

François Pottier. 1998a. A Framework for Type Inference with Subtyping. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 228–238. https://doi.org/10.1145/289423.289448 ↪ page 25

François Pottier. 1998b. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://hal.inria.fr/inria-00073205 ↪ pages 6, 23, 24, 25, 26, and 27

François Pottier. 2001. Simplifying Subtyping Constraints: A Theory. *Information and Computation* 170, 2 (2001), 153–183. https://doi.org/10.1006/inco.2001.2963 ↪ page 25

François Pottier. 2003. A Constraint-Based Presentation and Generalization of Rows. In *IEEE Symposium on Logic In Computer Science (LICS)*. Ottawa, Canada, 331–340. http://cambium.inria.fr/~fpottier/publis/fpottier-lics03.pdf ↪ page 3

Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. 2022. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:31. https://doi.org/10.4230/LIPIcs.ECOOP.2022.25 ↪ pages 24 and 26

Didier Rémy. 1994. *Type Inference for Records in Natural Extension of ML.* MIT Press, Cambridge, MA, USA, 67–95. ↪ pages 3 and 27

John C. Reynolds. 1997. *Design of the Programming Language Forsythe.* Birkhäuser Boston, Boston, MA, 173–233. https://doi.org/10.1007/978-1-4612-4118-8_9 ↪ page 6

Vincent Simonet. 2003. Type Inference with Structural Subtyping: A Faithful Formalization of an Efficient Constraint Solver. In *Programming Languages and Systems*, Atsushi Ohori (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–302. ↪ page 25

Geoffrey Seward Smith. 1991. *Polymorphic type inference for languages with overloading and subtyping.* Ph.D. Dissertation. Cornell University. ↪ pages 23 and 25

R. Stansifer. 1988. Type Inference with Subtypes. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 88–97. https://doi.org/10.1145/73560.73568 ↪ page 25

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486 ↪ page 26

Valery Trifonov and Scott Smith. 1996. Subtyping constrained types. In *Static Analysis*, Radhia Cousot and David A. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 349–365. ↪ page 24

Leo White. 2015. Row polymorphism. https://www.cl.cam.ac.uk/teaching/1415/L28/rows.pdf ↪ page 27