A Lightweight Type-and-Effect System for Invalidation Safety

Tracking Permanent and Temporary Invalidation with Constraint-Based Subtype Inference

CUNYUAN GAO, HKUST, Hong Kong, China
LIONEL PARREAUX, HKUST, Hong Kong, China

In many programming paradigms, some program entities are only valid within delimited regions of the program, such as resources that might be automatically deallocated at the end of specific scopes. Outside their live scopes, the corresponding entities are no longer valid – they are *permanently invalidated.* Sometimes, even within the live scope of a resource, the use of that resource must become *temporarily invalid*, such as when iterating over a mutable collection, as mutating the collection during iteration might lead to undefined behavior. However, high-level general-purpose programming languages rarely allow this information to be reflected on the type level. Most previously proposed solutions to this problem have relied on restricting either the aliasing or the capture of variables, which can reduce the expressiveness of the language. In this paper, we propose a higher-rank polymorphic type-and-effect system to statically track the permanent and temporary invalidation of sensitive values and resources, without any aliasing or capture restrictions. We use Boolean-algebraic types (unions, intersections, and negations) to precisely model the side effects of program terms and guarantee they are invalidation-safe. Moreover, we present a complete and practical type inference algorithm, whereby programmers only need to annotate the types of higher-rank and polymorphically-recursive functions. Our system, nicknamed InvalML, has a wide range of applications where tracking invalidation is needed, including stack-based and region-based memory management, iterator invalidation, data-race-free concurrency, mutable state encapsulation, type-safe exception and effect handlers, and even scope-safe metaprogramming.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → *Functional languages.*

Additional Key Words and Phrases: type inference, higher-rank polymorphism, effect systems

ACM Reference Format:
Cunyuan Gao and Lionel Parreaux. 2025. A Lightweight Type-and-Effect System for Invalidation Safety: Tracking Permanent and Temporary Invalidation with Constraint-Based Subtype Inference. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 366 (October 2025), 78 pages. https://doi.org/10.1145/3763144

1 Introduction

Motivating example. Consider the following Java program:

```
var xs = new java.util.ArrayList<String>();
xs.add("1"); xs.add("2"); xs.add("3");
for (var e : xs) { System.out.println(e); xs.clear(); }
```

This innocuous-looking code snippet creates a list of strings and then prints them in a `for` loop, which is syntax sugar for using an iterator in Java. However, the code also does something wrong: it tries to *clear* the mutable list *while still iterating* over it. This code crashes at runtime:

```
Exception in thread "main" java.util.ConcurrentModificationException
```

Authors' Contact Information: Cunyuan Gao, cgaoan@connect.ust.hk, HKUST, Hong Kong, China; Lionel Parreaux, parreaux@ust.hk, HKUST, Hong Kong, China.

ACM 2475-1421/2025/10-ART366
https://doi.org/10.1145/3763144

The error occurs because in Java, the semantics of iterating on mutable collections while mutating them is not well-defined. Java is nice enough to detect this error at runtime through sanity checks. In a language like C++, similar code would likely cause undefined behavior, which could lead to silent memory corruption and security vulnerabilities [97]. This problem is known as *iterator invalidation*, and it is a common source of bugs in imperative languages.

It would be better if the error could be caught at compile time, avoiding the need for runtime checks and the associated performance overhead. This would give us the best of both worlds: the safety of a high-level language and the performance of a low-level one. The root problem to address is that unrestricted uses of a mutable collection should be *invalid* while the collection is being iterated over. To address this, we need to track mutation and invalidation in the type system, giving us a way of preventing the mutation of certain values while they are being used – or "borrowed" – by other parts of the code.

Previous work. There is already a rich literature dedicated to this general idea, including:

Linearity and borrowing. A popular approach, originally advocated by Wadler [92] and henceforth by many others (e.g., [27, 54, 77]), is the use of *linear type systems* to prevent the sharing of mutable values. Linearity on its own severely restricts the allowed programming styles. *Borrowing* is the standard way of alleviating this restriction: by temporarily making the linear resource immutable, we can allow its aliasing, provided that we can bound the lifetime of these aliases, so that they all become unreachable or unusable when the resource becomes mutable again [54, 61, 77, 100]. We refer to this problem as ***temporary invalidation***. Rust [37, 54, 100] is a prominent example of a language that uses linearity and borrowing to track temporary invalidation.

Type-and-effect systems. In another line of work, type systems have been used to track the *effects* of program expressions [32, 48]. A historical use of this approach was to guarantee the safety of region-based memory management, where objects are allocated inside growable regions [1, 85, 90, 91]. When a region goes out of scope, it is deallocated altogether, and references to the objects that were allocated withing that region must no longer be used. We refer to this problem as ***permanent invalidation***. By associating the uses of a region with an effect that mentions the region, one can enforce its correct usage statically. Such effect systems typically work well with type inference, being quite similar to a form of static analysis. Recent work by Elsman [24] in ReML showed that this approach can be taken quite far: by adding *disjointness constraints* [7, 15] (called *disjunction constraints* by Elsman), it is possible to guarantee that, e.g., the mutations performed by two threads of computation will not interfere, avoiding data races.

Static capability systems. After the seminal works on type-and-effect systems, it was realized that these could be viewed as special cases of *static capability systems* [10, 17, 19, 94]. In the latter, performing certain operations, such as accessing the contents of regions, requires *capabilities* that are passed around the program but do not have a runtime representation – they are erased, similar to type parameters. By making some of these capabilities *linear*, one can connect the two schools of thought (tracking linearity vs. tracking effects) inside a very powerful system, which can express programming patterns like borrowing and forms of flow-sensitive reasoning [19, 29, 75].

Goals of this work. In this paper, we seek a *lightweight* type system for tracking permanent and temporary invalidation. By "lightweight", we mean that the system should be *easy to use* and should *not get in the way* of the programmer, even if that translates into slightly less powerful programming primitives. Concretely, we want our system to be amenable to practical type inference with minimal type annotations and to avoid the use of disciplines that constrain the aliasing of values or capabilities, such as linear types, uniqueness, or ownership disciplines.

Our approach. To achieve these goals, we propose to continue the *pure* type-and-effect systems line of work that was most lately embodied by Elsman's ReML language. While ReML is excellent at what it does, it does not fit our requirements: its use of unification makes it subject to *effect poisoning*, where spurious effects are inferred due to a lack of directionality in the type system (see App. A.3); it lacks higher-rank polymorphism to abstract over invalidation-aware interfaces, making it unable to express the iterator invalidation example above; and it does not fully support modular disjointness reasoning (see §6). While addressing these shortcomings, we adopt several techniques that are already well-known, and in that sense, we stand on the shoulders of giants:

- It is well-known that subtyping can be used to avoid the infamous problem of effect poisoning [86] (see also App. A.3). This is a real problem in practice, in that it easily gets in the way of programmers, preventing code from optimization [52] or leading to surprising type annotations [11], so subtyping has to figure prominently in our design.
- We embrace *Boolean-algebraic subtyping* in particular [68], which supports union, intersection, and negation type connectives. We will see that this allows us to express precise effect specifications, including disjointness constraints, without needing additional machinery.
- Higher-rank polymorphism is a well-known and powerful abstraction mechanism. Here, it allows us to describe complex invalidation-aware interfaces. Beyond iterator invalidation, these include data-race-free concurrency (§2.4), type-safe exception and effect handlers (§5.2), stack-based and region-based memory management (§5.3), and even scope-safe metaprogramming (§5.4). This is a core feature making InvalML reusable.
- Higher-rank polymorphism and polymorphic recursion, which we need, are not amenable to full type inference (for which they are both undecidable in general [36, 41]). Thankfully, there is a standard way to deal with this problem: *bidirectional typing* [22] allows one to type check higher-rank and polymorphically-recursive functions by simply providing their type signatures, while using constraint-based type inference to deal with everything else [72].
- We use *polymorphism levels* to keep track of the degree of polymorphism in types [28, 42, 66, 76, 79], notably preventing lower-level type variables from being instantiated to types that refer to higher-level type variables.
- We use lexically-scoped static regions along with region handles [11, 17, 24, 32, 48, 85]. Region handles are used to perform allocations in a corresponding static region and are first-class values that can be passed around and stored in data structures.

Main challenges. While the techniques above are individually well known, their combination has not been previously studied, and two major new challenges arise from it:

- How to deal with *type extrusion* arising from higher-rank polymorphism? Most previous approaches simply return an error when a type of a higher polymorphic level flows into a lower level. But we simply cannot do this in the presence of subtyping, as it would be overly conservative, thwarting complete type inference [20], and would also prevent perfectly legitimate programs from type checking, as we shall see later (§2.4). The original work on Boolean-algebraic subtyping [68] did not support local 'let'-polymorphism, let alone higher-rank polymorphism. While the former can be dealt with by a *lambda-lifting* trick [40, 73], the latter is fundamentally more difficult and requires what we call *subtype extrusion*. Parreaux [66], Parreaux et al. [67] previously described simple forms of subtype extrusion, but generalizing them to the Boolean-algebraic setting, where rigid type variables (a.k.a. skolems) can have multiple bounds, is nontrivial. This is explained in §2.3 and §4.4.
- How to type check the region introduction construct while accounting for the disjointness between the new region and yet-unknown outer regions? When a local region `r` is created inside a function `f`, region `r` should be understood to be disjoint from all regions currently installed on the stack. The main problem is that these *outer* regions are not statically known,

as they will vary depending on the call sites of `f`[1]. To address this, we introduce a novel mechanism: we attach a special "outer" type variable ω to polymorphic type schemes, using it to represent the dynamic extent of all regions that will be live at each of the corresponding call sites. This is explained in §2.4 and §3.2.

InvalML. We present our design as the InvalML language, which is implemented by extending the existing programming language MLscript (anonymized for review), a multi-paradigm language supporting both functional and imperative programming. A web demonstration of InvalML is available online at https://github.com/hkust-taco/invalml.

Our motivating example can be rewritten in InvalML as follows:

```
region r in  // This is used to delimit the scope of mutation
let xs = mkArrayList(r) in  // Creates a new mutable list in region r
add(xs, "1"); add(xs, "2"); add(xs, "3");
iter(xs, it ⇒ foreach(it, e ⇒ println(e); clear(xs)))
```

This program now contains a *type error*, which complains that the expression `clear(xs)` has effect `r`, which is forbidden from occurring in the corresponding scope. One way to fix this error is to call `clear(xs)` *outside* the scope of `it` (but still *inside* the scope of `r`), which was probably the original intent of the programmer who wrote the erroneous program[2]

Contributions. Our core technical contributions are the following:

- We define a *Boolean-algebraic subtyping* system in the higher-rank setting (§3)
- We present an algorithm for complete type inference without backtracking in that type system, using a novel subtype extrusion mechanism (§4).
- We propose a *lightweight effect system* built on top of this subtyping discipline, including a novel `outer`-variable mechanism to deal with statically unknown outer regions. The system is lightweight because:
  - It requires few type annotations, thanks to our complete type inference algorithm: only higher-rank and polymorphically-recursive functions need a type signature.
  - It benefits from an *economy of concepts*: the same Boolean algebra of types is used to describe the shapes of runtime values, the side effects of computations, the lifetimes of stack-allocated variables, and the scopes of program fragments, among others; and all of them can be subjected to disjointness reasoning and can be abstracted over by the same higher-rank polymorphism mechanism.
- We demonstrate the wide range of applications of our system, including:
  - Mutation-safe polymorphism (§3.2) and mutable state encapsulation (App. A.2).
  - Data-race-free structured concurrency (§2.4).
  - Permanent and temporary invalidation safety (§2.5).
  - Scope-safe metaprogramming with statically-typed and analytic quasiquotes allowing the manipulation of open code (§5.4).
  - Type-safe exceptions and effect handlers (§5.2)
  - Type-safe region-based and stack-based memory management (§5.3.2).

The appendices of this paper are available in the technical report version, which can be found online at https://lptk.github.io/invalml-paper.

[1]We are not aware of any previous work that addresses this problem. In particular, static capability-based approaches encode disjointness through linearity, which is sufficient but not always as flexible, and approaches like ReML do not typically allow expressing disjointness predicates that mix abstract and concrete regions. (Both of these are discussed in §6.)
[2]Note that our type system is *flow-insensitive*, unlike other work by, e.g., DeLine and Fähndrich [19], Foster et al. [29], Gordon [34], so it will *not* catch errors such as "calling `clear` too early", for some definition of "too early" specified by a protocol.

2 Lightweight Invalidation Tracking

In this section, we informally describe our methodology for invalidation tracking.

2.1 Core Type System of InvalML

InvalML uses a *Boolean-algebraic* subtyping approach inspired by MLstruct [68], albeit with a much simpler soundness proof (see App. D.4). However, InvalML uses different typing rules than MLstruct. For simplicity, it omits record types, which lets us devise a slightly more straightforward normal forms and constraint-solving rules (in §4), although this omission is benign and could easily be lifted. Also, whereas MLstruct is limited to top-level polymorphism, features a bidirectional higher-rank-polymorphic type system. This is possible because MLstruct's subtyping theory supports abstract types (in the form of type variables with bounds). Like in MLstruct, we use polymorphic types of the form '$\forall V\{\Xi\}.\,\tau$', called *multi-bounded polymorphism* by Parreaux et al. [67], which consists in a set of quantified type variables $V = \alpha_1, \alpha_2, \ldots$, a set of *upper and lower bounds* Ξ on these type variables, which must be consistent (i.e., all lower bounds must be subtypes of all upper bounds of the same type variable), and a type body τ.

Boolean-algebraic subtyping uses union, intersection, and negation types. For example:

```
type ResCode = 200 ∨ 404; type Real = Num ∧ ¬NaN ∧ ¬Inf
type Inf = PositiveInfinity ∨ NegativeInfinity
```

In the above example, $\vee$ is the union type constructor. The values of type `ResCode` are `200` and `404`, which are associated with corresponding singleton literal types written identically. $\wedge$ and $\neg$ are the intersection and negation type constructors. An intersection type satisfies both sides of the $\wedge$ sides simultaneously. Negation type $\neg\, T$ is the Boolean-algebraic negation of the subtyping relation with $\wedge$ and $\vee$ as conjunction and disjunction and $\top$ and $\bot$ as top and bottom, respectively. Intuitively, one can think of $\neg\, T$ as the type of all values that do not have type T, although this is technically only true in some situations, most importantly when T is concrete type constructor.

Historically, programming languages with ML-style type inference (e.g., OCaml and Haskell) avoided introducing implicit subtyping due to its complexity. *Algebraic subtyping* was proposed in MLsub as to effectively deal with this complexity [20, 21]. However, in the original formulation of algebraic subtyping, unions could occur only in positive (i.e., in *output*) positions and intersections could occur only in negative (i.e., *input*) positions. This restriction did not reduce the expressiveness of MLsub (which only had records, functions, and let bindings), but to allow more advanced features, such as tag-based pattern matching, effect sets, and disjointness constraints, one needs unrestricted unions and intersections. MLstruct removed this restriction:

```
let foo(x) = if x is { 0 then true, x then x }
// foo: (0 ∨ 'a ∧ ¬0) → (true ∨ 'a)
```

Function `foo` takes a parameter that can be either 0 or any other non-zero value of type `'a`, i.e., `(0 ∨ 'a ∧ ¬0)`; it maps the former to **`true`** and the latter to itself, i.e., `(true ∨ 'a)`.

As we shall see, principal type inference is achieved in this context notably by rewriting ambiguous constraints like $\tau \leq \alpha \vee \sigma$ to bounds on type variables that are equivalent to the original constraints (according to the rules of Boolean algebras) – here, $\tau \wedge \neg\sigma \leq \alpha$.

In this paper, we extend *Boolean-algebraic subtyping* to *higher-rank polymorphism*, a major feature that was not supported in previous work. We use bidirectional typing together with constraint-based type inference to support higher-rank polymorphism while requiring relatively few type annotations. Although this basic approach is standard [72], adapting it to our setting requires a new level-based subtyping-aware type extrusion mechanism (see §4.4). This allows InvalML to support complete type inference without backtracking, whereby only higher-rank and polymorphically-recursive definitions need to be annotated, which is standard.

2.2 A Lightweight Type-and-Effect System

All approaches that aim to track effects statically without requiring the likes of monad transformers need a way of combining various effects $\varphi_1, \varphi_2, \ldots$ into a single type $\varphi_1 \vee \varphi_2 \vee \ldots$ and they need such combinations to possibly be *partially abstract* in order to allow polymorphism and modularity. This implies the use of extensible effect types with effect variables, as in $\varphi_1 \vee \varphi_2 \vee \alpha$. In previous work, this was typically done through *row polymorphism*, a form of polymorphism allowing exactly this kind of extensibility. However, traditional unification-based row polymorphism has severe limitations which are exacerbated in the context of effect systems [52] and notably suffers from the well-known poisoning problem [98]. ReML [24] adopts Tofte and Birkedal [88]'s region labels that are similar to a simple form of row polymorphism. Row polymorphism was originally preferred over subtyping to model extensible records [78, 80, 96], but more recent work showed that subtype inference could be made more tractable by adopting an *algebraic* approach [20, 21, 66, 74], leading Parreaux and Chau [68] to propose a new *Boolean-algebraic subtyping* system as powerful as the row-polymorphic system of Rémy [80] but without requiring special row variables and without the pitfalls of unification. In that system, a combination of effects φ_1 and α (where α is an abstract type or type variable) would simply use the *union type constructor* $\varphi_1 \vee \alpha$.

To illustrate our effect system, consider the `mapi` function below, which maps elements in a given list with their index:[3]

```
fun mapi[A, B, E](xs: List[A])(f: (Int, A) →{E} B){E}: List[B] =
  region r in let index = r.ref -1 in List.map(xs)(x ⇒
    index := !index + 1; f(!index, x))
```

where function type syntax `S →{E} T`, formally written $S \xrightarrow{E} T$, stands for a function that takes arguments of type `S`, has effects `E`, and returns values of type `T`. Regions are first-class values that are assigned static scopes through the **region** construct and that are used to track scope safety statically. The `x ⇒ ...` lambda captures the mutable reference `index`, passing it to `List.map`. InvalML prevents the leakage of mutable state: it ensures that all mutations associated with the region `r` will be unobservable outside of the scope of `mapi`; this is discussed in more detail in App. A.2.

2.3 Region Typing and Type Extrusion

When the user writes **region** `r` **in** `e`, where `e` is some expression or block, we introduce a new *type variable* α used to track the scope of the region. This variable is *rigid*, meaning that it cannot be constrained or instantiated by any other type; in type inference jargon, this is referred to as a *skolem* variable. This is like an abstract type about which nothing is known — or *almost* nothing, as we shall see next. Note that the **region** `r` form, followed by the rest `R` of the current block, as in **region** `r; R`, is desugared into the previous form: **region** `r` **in** `R`. When creating a new reference in a region `r`, as done through the '`r.`**ref** `value`' form, we assign to the result the type $\mathsf{Ref}[\tau, \textbf{out}\,\alpha]$, where τ is the type of values stored in the reference, which is some supertype of the type of `value`. The keyword **out** indicates reference types are covariant on the region types. Suppose two references $r_1 : \mathsf{Ref}[\tau, \textbf{out}\,\alpha]$ and $r_2 : \mathsf{Ref}[\tau, \textbf{out}\,\beta]$. A function that needs to consume both r_1 and r_2 can be declared with an argument type $\mathsf{Ref}[\tau, \textbf{out}\,\alpha \vee \beta]$. Since $\alpha \leq \alpha \vee \beta$ and $\beta \leq \alpha \vee \beta$, one can upcast the types of r_1 and r_2 accordingly. We introduce subtyping formally in §3.2.

When a value that depends on region `r` leaves the scope of `r`, it is *extruded*, meaning that its type is stripped of all references to `r`. When `r` occurs in positive positions, it is replaced by its known upper bound (which is $\top$, the *top* type of the subtyping lattice, if `r` is the outermost region); when `r` occurs in negative positions, it is replaced by $\bot$, the *bottom* type. Moreover, we ensure that the

[3]Since this definition does not use higher-rank polymorphism or polymorphic recursion, all type and effect annotations here could be removed, and the principal type would be inferred by InvalML.

entry-point of the program (its “main” function) can only be typed with effect $\bot$, which denotes the absence of effects and expresses that all effects of the program are properly handled/encapsulated. This ensures that any leaked region reference *cannot* be used outside of its scope. As an example, the expression `(region r in r.ref 0) := 1`, which tries to update a reference after it has left its defining region, can only be assigned effect $\top$, ensuring that it cannot be used in any meaningful way, as that effect would propagate all the way to the entry point of the program and fail its type checking.[4] Similar leakage can result from function captures, as exemplified below:

```
region r in let f = (x ⇒ r.ref (x + 1)) in f
```

This region expression returns a lambda function that captures the region instance `r`, yielding a location referring to an integer. The allocation behavior is reflected on the effect type of `f`, thus $f : \mathsf{Int} \xrightarrow{\alpha} \mathsf{Ref}[\mathsf{Int}, \mathbf{out}\,\alpha]$ inside the region of `r`. When we exit the region, α also leaves the scope. Therefore, we also need to extrude the function type, and the whole program has type $\mathsf{Int} \xrightarrow{\top} \mathsf{Ref}[\mathsf{Int}, \mathbf{out}\,\top]$. Even though the region is encapsulated by the function, it is fine if we never call such a function that carries unsolvable effects.

The level-based mechanism by which we extrude region variables, described in §4.4, is in fact the very same mechanism that is used to prevent polymorphic type variables from leaking out of their scopes. While the basic idea of type extrusion is very old, a *subtype* extrusion mechanism that works with Boolean-algebraic subtyping and higher-rank polymorphism is one of the important technical contributions of this paper.

2.4 Disjointness with Negation Types

In §1, we presented the ability to reject temporary invalidations by using negations types, preventing an iteration from runtime errors due to the improper `clear` behaviors. Partially freezing some regions highly depends on the witness that two regions are disjoint from each other. If we don’t know two regions are independent, there is no guarantee that manipulating one of them will not affect the other one. Another example can be a function `fork` (a reminiscent of `fork` in many systems [17, 24, 33, 82, 95, 102]) that would be used to run two computations in parallel. To guarantee the absence of data races, we need to ensure that the two parallel computations cannot possibly interfere with each other — that is, their effects must be *disjoint*. We can give such a function the following type:

$$\mathsf{fork} : \forall \alpha, \beta, \gamma_1, \gamma_2\, \{\beta \le \neg\alpha\}.\ ((\,) \xrightarrow{\alpha} \gamma_1,\ (\,) \xrightarrow{\beta} \gamma_2) \xrightarrow{\alpha \vee \beta} (\gamma_1, \gamma_2)$$

whereby we require the second function to have an effect that subtypes the *negation* the effect the first one carries. This is done through the bound $\{\beta \le \neg\alpha\}$, which is equivalent to $\beta \wedge \alpha \le \bot$, a disjointness constraint. In InvalML, we track all regions that are live on the stack:

```
region r1 in region r2 in region r3 in ...
```

In this example, `r1` is a top-level region with type $\mathsf{Region}[\mathbf{out}\,\alpha]$, while `r2` and `r3` are two nested regions with types $\mathsf{Region}[\mathbf{out}\,\beta]$ and $\mathsf{Region}[\mathbf{out}\,\delta]$ respectively. When we enter the region `r2`, we are aware of that `r1` is already in the stack, so we track the type variable of `r1`, i.e., α, and adopt its negation as a upper bound of β. Similarly, for `r3`, we know `r1` and `r2` are alive, so δ has both $\neg\alpha$ and $\neg\beta$ as its upper bounds. We merge the alive regions’ type variables via unions, and δ’s upper bounds can be written as $\neg(\alpha \vee \beta)$. If we are about to pass `r1` and `r3` to some function f, where

[4]Of course, it would not be ideal for such errors manifest potentially far from their source, which is how a naive implementation would behave. Thankfully, we can store extra meta-information when skolems are extruded, so as to present users with accurate error messages, taking inspiration from related work on tracing the causes of type errors with constraint-based type inference [6]. We could also immediately produce a warning whenever a function’s effect is inferred to be $\top$.

$f : \forall \gamma_1, \gamma_2 \{\gamma_2 \leq \neg\gamma_1\}.\, \mathrm{Region}[\mathbf{out}\, \gamma_1] \to \mathrm{Region}[\mathbf{out}\, \gamma_2] \xrightarrow{\gamma_1 \vee \gamma_2} \mathsf{Int}$, we can notice that $\delta \leq \neg\alpha$ by the transitivity of subtyping. Nevertheless, the following expression does not type check (where 'r ⇒ · · · ' is the usual lambda syntax):

```
region r1 in
let g = (r ⇒ region r2 in f(r, r2)) in
region r3 in g(r3)
```

A lambda function that passes the parameter region r and a local region r2 to f is bound to g. Similarly, r1 has type $\mathrm{Region}[\mathbf{out}\, \alpha]$, while r2 and r3 are typed to $\mathrm{Region}[\mathbf{out}\, \beta]$ and $\mathrm{Region}[\mathbf{out}\, \delta]$ respectively. However, since r2 is inside a lambda function and will not be executed immediately, we can only witness that r2 is separated from r1 and r3 is separated from r1! The type check for g(r3) fails because g requires a region that is disjoint from r2. At runtime, r2 will be allocated inside r3 so they should be separated from each other, but the system fails to track the information.

To overcome this problem, we propose the outer scope variable ω . An outer scope variable can be declared as forall-quantified and used in the type annotations. When instantiating a polymorphic type, InvalML will substitute ω with the union of the current alive regions' type variables. In the above example, if we annotate g with the type $\forall \gamma, \omega.\, \{\gamma \leq \omega\}\, \mathrm{Region}[\mathbf{out}\, \gamma] \xrightarrow{\gamma} \mathsf{Int}$, inside the function body, r2 will be considered disjoint with the whole outer environment, and β will have $\neg(\alpha \vee \omega)$ as the upper bound. ω will be instantiated with $\alpha \vee \delta$ on the call site of g (i.e., g(r3)) and we can pick δ for γ.

The use of outer variables also brings out the reason why extrusions can be useful, and we do not raise a type error when an extrusion happens. Let's consider the following function bar that defines a local region and freezes it while executing an unknown function passed in parameter, where freeze is a function that takes a region and an argument function, preventing the given region from being accessed by the argument function:

```
fun freeze: ∀ R, E, T {E ≤ ¬R}. (Region[out R], () →{E} T) →{R ∨ E} T
fun bar(f) = region r in freeze(r, () ⇒ f(123))
```

To type check this program, it requires the type system to automatically prove that any region f accesses (including arbitrary regions at the call sites of bar) is disjoint from r. InvalML can infer the type $\forall \alpha, \beta, \omega\{\beta \leq \omega\}.(\mathsf{Int} \xrightarrow{\beta} \alpha) \xrightarrow{\beta} \alpha$ for the bar function, thanks to the outer variable and a useful extrusion: when we are constraining $\beta \leq \neg\gamma$, where γ is the region skolem of r, we extrude γ to $\neg\omega$, which yields $\beta \leq \neg\neg\omega$, and it is equivalent to $\beta \leq \omega$.

2.5 Temporary Invalidation

To introduce the notion of *temporary invalidation*, it may be useful to relate that notion to the better-known and closely related concept of *borrowing*, as in languages like Rust.

Rust-style borrowing refers to a typing discipline that essentially ensures that

(1) during the *lifetime* — or, equivalently, within the *scope* — of a mutable reference to some resource, no other reference to the same resource can be used; and
(2) if there are multiple references to the same resource, the resource must be frozen: no mutation can happen to it while these references are still in scope.[5]

Rust uses this discipline to ensure memory safety in the face of mutation. Another great use case for this approach is to *statically* rule out iterator invalidation bugs.

Below is how we can express the API of the collections library alluded to in the introduction. It leverages disjointness to prevent iterator invalidation bugs at compile time. First, we specify the signatures of the ArrayList and Iter data types and operations:

[5]Rust actually ensures this by making all aliasing references explicitly immutable.

```
class ArrayList[T, R] with constructor ArrayList(data: Ref[Array[T, R], R])
class Iter[T, R] with constructor Iter(next: () →{R} Option[T])
```

In InvalML, data types are defined using the `class` construct, which defines algebraic data types (in this case, with a single constructor). `Ref[R, T]` is the type of mutable references in region `R` to values of type `T`. Notice that these two mutable data types keep track not only of the element type `T` but also of an extra parameter `R` that represents the region in which they can operate.

```
fun mkArrayList: ∀ R, T. (Region[R]) →{R} ArrayList[T, R]
fun add: ∀ R, T. (ArrayList[T, R], T) →{R} ()
fun clear: ∀ R, T. ArrayList[T, R] →{R} ()
fun foreach: ∀ E, R, T. (Iter[T, R], T →{E} ()) →{R ∨ E} ()
```

The `add` and `clear` functions are straightforward: they simply add an element to the list and clear it, respectively, and obviously have the effect associated with the region `R` of their input mutable list. The `foreach` function is more interesting: in addition to an iterator, it takes a function that is allowed to perform *any* effect `E`, which is reflected in the resulting effect of the function itself, `R ∨ E`, which expresses that the `foreach` performs the effect of its argument function in addition to the effect attached to its iterator argument. In fact, due to the properties of implicit polymorphism and subtyping [68], we could merge `E` and `R`, which are here undistinguishable, resulting in the following simplified signature:[6]

```
fun foreach: ∀ E, T. (Iter[T, E], T →{E} ()) →{E} ()
```

The function used to create an iterator from an existing `ArrayList` is the most interesting, and we give the type annotation as follows:

```
fun iter: ∀ Res, R, E, T {E ≤ ¬R}.
  (ArrayList[T, R], ∀ S. Iter[T, S] →{S ∨ E} Res) →{E ∨ R} Res
```

This function has a higher-rank type: it makes use of a polymorphic argument function, which is used to prevent the iterator passed to it from being used outside of its scope. Moreover, the effect `E` is *constrained* to be a subtype of the *negation* of region `R`, which is denoted by `{E ≤ ¬R}`. This is what crucially ensures that the original region `R` of the `ArrayList` *cannot* be used while the iterator is in scope. Notice that the argument function is still allowed to use the iterator with type `Iter[T, S]`. Accessing the iterator yields effect `S` instead of `R`, so the argument function's overall effect is the union of `S` and `E`. Finally, the effect of the `iter` function is the union of `E` and `R`.

Below is an example correct usage of the above API.[7] Notice how we can create arbitrary amounts of mutable regions, mutating and iterating on them without any issue:

```
region r in let a1 = mkArrayList(r) in add(a1, 12); add(a1, 34);
  iter of a1, it1 ⇒ region s in
    let a2 = mkArrayList(s)
    foreach of it1, v1 ⇒ add(a2, v1)
    iter of a2, it2 ⇒ foreach of it2, v2 ⇒ println(v2)
    clear(a2)
```

Below is an example *incorrect* usage of the same API, causing a type checking error:

```
region r in let a = mkArrayList(r); add(a, 12); add(a, 34);
  iter of a, it ⇒ foreach of it, v ⇒ println(v); clear(a) // type error
```

3 Formalization of the Type System

We now present $\lambda^{!\perp}$, the declarative core type system of InvalML.

[6]To understand this equivalence, consider that all well-typed uses of one version would also be well-typed with the other.
[7]In MLscript, '`f of a, b, c`' means '`f(a, b, c)`' where '`of`' is right-associative (like Haskell's '\$').

Syntax

Type variable-like	$\nu ::= \alpha \mid \omega$	*Monomorphic type*	$\tau, \sigma, \varphi ::= \nu \mid \tau \xrightarrow{\tau} \tau \mid \mathsf{A}[\overline{a}] \mid \top^{\pm} \mid \tau \vee^{\pm} \tau \mid \neg\tau$
Type argument	$a, b ::= \textbf{in}\,\tau\,\textbf{out}\,\tau$	*General type*	$\mathcal{T} ::= \tau \mid \forall V\{\Sigma\}.\,\mathcal{T} \mid \mathcal{T} \xrightarrow{\tau} \mathcal{T}$
Variables	$V, W ::= \omega \mid V\,\alpha$	*Region accumulation*	$\zeta ::= \bot \mid \zeta \vee \nu$
Term	$t ::= \lambda x.\,t \mid t\ t \mid x \mid t : \mathcal{T} \mid \textbf{let}\,x = t\,\textbf{in}\,t \mid \mathsf{C}(\,\overline{t}\,) \mid \textbf{if}\,\overline{t\,\textbf{is}\,\mathsf{C}(\overline{x})\,\textbf{then}\,t}$ $\mid \textbf{region}\,x\,\textbf{in}\,t \mid t.\textbf{ref}\,t \mid\ !t \mid t := t$		
Constructor	$c ::= \mathsf{C}(\overline{\mathcal{T}}) \mid \mathsf{C}[\overline{\beta}](\overline{\mathcal{T}})\ \textbf{extends}\ \mathsf{A}[\overline{a}]$	*Polarity*	$\pm, \diamond ::= + \mid -$
Top-level declaration	$d ::= \textbf{class}\ \mathsf{A}[\overline{\alpha}]\ \textbf{with constructor}\ \overline{c}$		

Contexts

Declaration context	$\mathcal{D} ::= \epsilon \mid \mathcal{D}\ d$
Typing context	$\Gamma ::= \epsilon \mid \Gamma\,(x : \mathcal{T}) \mid \Gamma\,\nu \mid \Gamma\,(\alpha \leq^{\pm} \tau) \mid \Gamma\,\bullet$
Bounds context	$\Sigma ::= \epsilon \mid \Sigma\,(\alpha \leq^{\pm} \tau) \mid \Sigma\,\nu \mid \Sigma\,\bullet \mid \Sigma\,\textbf{\textit{err}}$
Subtyping context	$\Xi ::= \epsilon \mid \Xi\,\Sigma \mid \Xi\,(\tau \leq \tau) \mid \Xi\,\triangleright(\tau \leq \tau) \mid \Xi\,\textbf{\textit{err}}$

Shorthands $\quad \textbf{in}\,\tau \equiv \textbf{in}\,\tau\,\textbf{out}\,\top \qquad \textbf{out}\,\tau \equiv \textbf{in}\,\bot\,\textbf{out}\,\tau \qquad \tau \equiv \textbf{in}\,\tau\,\textbf{out}\,\tau \qquad \mathsf{A} \equiv \mathsf{A}[\overline{\textbf{in}\,\bot\,\textbf{out}\,\top}]$

$$\mathcal{T}_1 \to \mathcal{T}_2 \equiv \mathcal{T}_1 \xrightarrow{\bot} \mathcal{T}_2 \qquad t_1;\ t_2 \equiv \textbf{let}\,x = t_1\,\textbf{in}\,t_2 \quad (x\ \textit{fresh})$$

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

3.1 Types and Syntax

The syntax of $\lambda^{!\bot}$ is given in Figure 1. Because $\lambda^{!\bot}$ is *predicative*, we split *monomorphic* types from *general* types and allow type variables to range over only the former.

Monomorphic types includes type variables (written as lowercase Greek letters $\alpha, \beta, \dots$), outer scope variables (written as ω), function types, algebraic data types, $\top$, $\bot$, and types formed by Boolean algebraic connectives. Outer scope variables are forall-quantified, abstracting the outer environments where the corresponding polymorphic types will be instantiated. When a polymorphic type is instantiated, we always substitute its ω with the current outer region accumulation ζ, a union of all living outer regions' type variables. For instance, if we have two outer regions with type $\mathsf{Region}[\textbf{out}\,\alpha]$ and $\mathsf{Region}[\textbf{out}\,\beta]$ respectively, then $\zeta = \alpha \vee \beta$. If we are at the top level (i.e., no outer region), then $\zeta = \bot$. Function type $\tau_1 \xrightarrow{\tau_2} \tau_3$ carries an effect type τ_2. When $\tau_2 = \bot$, the function is considered pure. An algebraic data type consists of a type name and a series of type arguments. $\lambda^{!\bot}$ supports *use-site variance* via keywords `in` and `out`, following the idea proposed by Tate [87] and later adopted by Kotlin [8]. Declaration-site variance can be considered a syntax sugar. For example, type `Foo[T]` with declaration **`class`** `Foo[`**`out`** `A]` **`with constructor`** `/*...*/` is equivalent to type `Foo[`**`out`** `T]` with declaration **`class`** `Foo[A]` **`with constructor`** `/*...*/`. To avoid repetition, we use polarity notations $\pm$ and $\diamond$ to treat $\top/\bot$, $\vee/\wedge$, and $\leq/\geq$ symmetrically [64, 68]. For example, $\top^{+}$ is $\top$, while $\top^{-}$ is $\bot$.

General types cover both monomorphic and polymorphic types. A polymorphic type $\forall V\{\Sigma\}.\,\mathcal{T}$ involves a set of bounds Σ on quantified variables V. A V context maintains exactly one outer variable without any bound. A polymorphic type, e.g., $\forall\omega, \alpha.\,\alpha \to \alpha$, might not need its outer scope variable. In this case, the outer scope variable can be omitted from the type annotation, as in $\forall\alpha.\,\alpha \to \alpha$. Each bound can be either an upper bound or a lower bound. A polymorphic type $\forall V\{\Sigma\}.\,\mathcal{T}$ is only well-formed when all elements of Σ are bounds. We also support higher-rank polymorphic functions, given by $\mathcal{T} \xrightarrow{\tau} \mathcal{T}$. Notice that the effect type is always monomorphic.

[8] https://rosstate.org/publications/mixedsite/

$$\boxed{\Gamma,\, \zeta \vdash t : \mathcal{T} \,!\, \varphi} \qquad \delta \ ::= \ \cdot \mid \Downarrow \qquad \text{Notation: } \Gamma,\, \zeta \vdash_{\Downarrow} t : \mathcal{T} \,!\, \varphi \text{ means } \Gamma,\, \zeta \vdash (t : \mathcal{T}) : \mathcal{T} \,!\, \varphi$$

$$\textsc{T-Var}\ \frac{\Gamma(x) = \mathcal{T}}{\Gamma,\, \zeta \vdash x : \mathcal{T} \,!\, \bot} \qquad \textsc{T-Abs1}\ \frac{\Gamma\ (x : \tau),\, \zeta \vdash t : \mathcal{T} \,!\, \varphi}{\Gamma,\, \zeta \vdash \lambda x.\, t : \tau \xrightarrow{\varphi} \mathcal{T} \,!\, \bot} \qquad \textsc{T-Abs2}\ \frac{\Gamma\ (x : \mathcal{T}_1),\, \zeta \vdash_{\Downarrow} t : \mathcal{T}_2 \,!\, \varphi}{\Gamma,\, \zeta \vdash_{\Downarrow} \lambda x.\, t : \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2 \,!\, \bot}$$

$$\textsc{T-App}\ \frac{\Gamma,\, \zeta \vdash t_1 : \mathcal{T} \xrightarrow{\varphi} \mathcal{S} \,!\, \varphi' \qquad \Gamma,\, \zeta \vdash_{\Downarrow} t_2 : \mathcal{T} \,!\, \varphi'}{\Gamma,\, \zeta \vdash t_1\, t_2 : \mathcal{S} \,!\, \varphi \vee \varphi'} \qquad \textsc{T-Let}\ \frac{\Gamma,\, \zeta \vdash t_1 : \mathcal{T}_1 \,!\, \varphi \qquad \Gamma\ (x : \mathcal{T}_1),\, \zeta \vdash_{\delta} t_2 : \mathcal{T}_2 \,!\, \varphi}{\Gamma,\, \zeta \vdash_{\delta} \mathbf{let}\, x \,=\, t_1 \,\mathbf{in}\, t_2 : \mathcal{T}_2 \,!\, \varphi} \qquad \textsc{T-Asc}\ \frac{\Gamma,\, \zeta \vdash t : \tau \,!\, \varphi}{\Gamma,\, \zeta \vdash_{\Downarrow} t : \tau \,!\, \varphi}$$

$$\textsc{T-Subs1}\ \frac{\Gamma,\, \zeta \vdash t : \tau_1 \,!\, \varphi \qquad sub(\Gamma) \vdash \tau_1 \leq \tau_2}{\Gamma,\, \zeta \vdash t : \tau_2 \,!\, \varphi} \qquad \textsc{T-Subs2}\ \frac{\Gamma,\, \zeta \vdash t : \mathcal{T} \,!\, \varphi_1 \qquad sub(\Gamma) \vdash \varphi_1 \leq \varphi_2}{\Gamma,\, \zeta \vdash t : \mathcal{T} \,!\, \varphi_2} \qquad \textsc{T-Gen}\ \frac{\begin{array}{c}\Gamma \bullet V\, \Sigma,\, \zeta \vee \omega \vdash_{\Downarrow} t : \mathcal{T} \,!\, \bot \\ \omega \in V \qquad sub(\Gamma) \vdash \forall V\{\Sigma\}\ \textbf{\textit{cons.}}\end{array}}{\Gamma,\, \zeta \vdash_{\Downarrow} t : \forall V\{\Sigma\}.\, \mathcal{T} \,!\, \bot}$$

$$\textsc{T-Inst}\ \frac{\begin{array}{c}\Gamma,\, \zeta \vdash t : \forall V\{\Sigma\}.\, \mathcal{T} \,!\, \varphi \qquad sub(\Gamma) \vDash \rho(\Sigma) \\ dom(\rho) = V \qquad \omega \in V \qquad \rho(\omega) = \zeta\end{array}}{\Gamma,\, \zeta \vdash t : \rho(\mathcal{T}) \,!\, \varphi} \qquad \textsc{T-Region}\ \frac{\begin{array}{c}\alpha \notin FV(\Gamma) \cup FV(\zeta) \cup FV(\tau) \cup FV(\varphi) \\ \Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)\ (x : \text{Region}[\mathbf{out}\, \alpha]),\, \zeta \vee \alpha \vdash t : \tau \,!\, \varphi \vee \alpha\end{array}}{\Gamma,\, \zeta \vdash \mathbf{region}\, x \,\mathbf{in}\, t : \tau \,!\, \varphi}$$

Fig. 2. Typing rules.

The syntax of lambda abstractions, applications, variables, ascriptions, let-bindings, and constructors is standard. We adopt keyword `class`, `with`, and `constructor` for ADT declarations. Pattern matching is done through the `if-is` syntax. A pattern consists of a constructor's name and a series of bindings. Syntax **region** x **in** t creates a new region instance with type Region[**out** α] for some α and binds it as x in the body t, similar to the `region` construct in Flix [49] and `letregion` from other languages. Allocating a new reference cell requires a region instance, while reading $!t$ and writing $t\ :=\ t$ don't. Mutable cells have types of the form Ref[τ, **out** σ], where τ indicates the type of the stored value and σ denotes the associated region(s). A reference cell belongs to only one region, but one can upcast the second type argument to approximate this information. Region and Ref are not included in Figure 1. Instead, we assume the existence of two "primitive" data types: Region[**out** α] and Ref[α, **out** β]. Because they are primitives, Region and Ref should not be instantiated as a user-defined ADT. We consider programs that contain such instantiations *ill-formed.* and assume only well-formed instantiations in the rest of this paper.

We use polymorphism levels [28, 42, 66, 76, 79] to keep track of the *scope* in which types are allowed to live, so that we can properly *extrude* those types that contain scope level violations (explained in §4.4). We insert context separators $\bullet$ in context Γ and Σ. The level of a given v is implied by counting the number of $\bullet$ appearing ahead of v in the context. We assume that all contexts and types in the rest of this section are well-formed. The definitions of type and context well-formedness are given in App. C; they require that type variables that appear in types can always be found in the context and that if a type variable in positive/negative position can reach a function type or data type via intersections/unions, this function type or data type should have level 0. The latter condition is not a requirement for soundness but is only assumed for the inference algorithm to be complete. As usual, we use *Barendregt's convention* [5] for bindings.

3.2 Typing Rules

Figure 2 presents the typing rules of $\lambda^{!\bot}$. Judgment $\Gamma,\ \zeta \vdash t : \mathcal{T}\ !\ \varphi$ says that term t can be typed as $\mathcal{T}$ with effect φ in context Γ, where outer region accumulation is ζ. Since complete type inference for higher-rank systems is undecidable [72], we make use of user-provided type annotations when higher-rank polymorphism is needed but allow unannotated terms elsewhere. For simplicity, we write $\Downarrow$ subscription to indicate the requirement of type annotations. For instance, T-Abs1 specifies a way to type lambdas regardless of the presence of an expected type; it assumes that the parameter type is monomorphic. On the other hand, T-Abs2 shows how the presence of an expected type (in the form of a type annotation) can be used to type a lambda with polymorphic parameter and result types. Notice that the general type $\mathcal{T}_2$ is propagated inwards. These rules overlap, which is fine, as the type system is not meant to be algorithmic. Applications can leverage a known function type to type the argument. We borrow the meta variable δ from the work of PEYTON JONES et al. [72]. If $\delta =\Downarrow$, we switch to the check mode and make use of annotations. In T-Let, for example, if the whole let-binging is annotated, we propagate the annotation $\mathcal{T}_2$ to the body. Otherwise, T-Let still needs to infer the type of t_2.

T-Asc is used when the current type annotation cannot be leveraged further and T-Subs1 is used to widen monomorphic types. Similarly, T-Subs2 is used to widen the monomorphic effects, regardless of the term type $\mathcal{T}$. Function $sub(\Gamma)$ drops variable bindings in the context Γ, returning type variables, subtyping assumptions, and context separators. One might notice that in T-App, we use φ' for t_1's and t_2's effects (and similarly in T-Let). This can be achieved, thanks to T-Subs2. Assume that t_1 yields effect φ_1, and t_2 yields effect φ_2, we can upcast two effect types to $\varphi_1 \vee \varphi_2$. The subtyping rules are introduced in the next subsection. Contrary to much of the previous work on bidirectional typing for higher-rank polymorphism, we do not present a *polymorphic subsumption* rule, which would allow widening general types. Adding such a rule is possible (following, for example, the approaches of Cui et al. [18], Dunfield and Krishnaswami [23], Odersky and Läufer [62], Zhao and Oliveira [103]) but it would introduce vast amounts of complication to the metatheory and type inference algorithm, so we feel that it would detract from the core contributions of our paper.

Rule T-Gen checks and generalizes a given term t with quantified type variables V and bounds Σ. Although our implementation implicitly generalizes all **fun**-bound definitions (even those without a type annotation), for simplicity of our presentation, here we only generalize a term when it has a polymorphic type annotation; this restriction is easy to relax in practice. The premise $\Gamma \bullet V\ \Sigma,\ \zeta \vee \omega \vdash_{\Downarrow} t : \mathcal{T}\ !\ \bot$ increases the level of V by inserting a separator $\bullet$. It then propagates the type $\mathcal{T}$ when typing the body. It also appends the outer variable $\omega \in V$ onto ζ, which enables region $r : \text{Region}[\textbf{out}\,\alpha]$ allocated in t to have $\alpha \leq \neg\omega$ (which is explained in §2.4). To ensure consistency, this rule requires that there exist a substitution of V such that the variable bounds Σ hold in the current bounds context $sub(\Gamma)$, written $sub(\Gamma) \vdash \forall V\{\Sigma\}$ ***cons.***. Crucially, we forbid generalizing terms that are not pure to ensure soundness with mutable references [46, 101]. Consider:

```
let foo: ∀ α . α → α =
  region x in let r = x.ref None() in y ⇒ let res = !r in r := Some(y); res
foo(42); not foo(true)
```

`foo` is not pure due to its use of mutable reference operations; generalizing it would break type soundness, as the second call `foo(true)` would return `Some(42)`, when it would be simultaneously assigned type `Option[Bool]`. Thankfully, this example cannot be typed in $\lambda^{!\bot}$ because T-Gen requires generalized terms to have the effect $\bot$ (i.e., be pure). Rule T-Inst instantiates a polymorphic term by a substitution ρ. Notice that $\omega \in V$ must be substituted with ζ. The premise $sub(\Gamma) \vDash \rho(\Sigma)$ guarantees that all bounds in Σ can be derived by the current bound context after being substituted. Both substitution ρ and entailment $\vDash$ are formally defined in App. C.

T-Region allocates a new region instance with type $\text{Region}[\textbf{out}\,\alpha]$, where α is a new type variable at a higher level (indicated by the separator) and binds the instance to x. To make this region disjoint from outer ones, we add an upper bound $\neg\zeta$ to the region type variable α. Similarly,

Table 1. Typing of Primitives

Desugaring	Builtin Signature
t_1**.ref** $t_2 \rightsquigarrow \mathsf{ref}\ t_1\ t_2$	$\mathsf{ref} : \forall\alpha,\ \beta.\ \mathsf{Region}[\mathbf{out}\,\alpha] \to \beta \xrightarrow{\alpha} \mathsf{Ref}[\beta,\ \mathbf{out}\,\alpha]$
$!t \rightsquigarrow \mathsf{get}\ t$	$\mathsf{get} : \forall\alpha,\ \beta.\ \mathsf{Ref}[\mathbf{out}\,\beta,\ \mathbf{out}\,\alpha] \xrightarrow{\alpha} \beta$
$t_1 := t_2 \rightsquigarrow \mathsf{set}\ t_1\ t_2$	$\mathsf{set} : \forall\alpha,\ \beta.\ \mathsf{Ref}[\mathbf{in}\,\beta,\ \mathbf{out}\,\alpha] \to \beta \xrightarrow{\alpha} \beta$
$\mathsf{C}(\overline{t_i}) \rightsquigarrow \mathsf{construct}_\mathsf{C}\ \overline{t_i}$	$\mathsf{construct}_\mathsf{C} :\ \forall\overline{\alpha}.\ \mathcal{T}_1 \to \ldots \to \mathcal{T}_n \to \mathsf{A}[\overline{b}]$
$\mathbf{if}\ t\ \overline{\mathbf{is}\,p_i\ \mathbf{then}\ t_i} \rightsquigarrow \mathsf{match}_\mathsf{A}\ t\ \overline{\lambda x_{i1}.\ \ldots\lambda x_{in}.\,t_i}$	$\mathsf{match}_\mathsf{A} :\ \forall\overline{a},\ \delta, \gamma\,.\,\mathsf{A}[\overline{a}] \to \overline{(\forall\overline{\beta}.\ \mathcal{T}_{i1} \to \ldots \to \mathcal{T}_{in} \xrightarrow{\gamma} \delta\ )}^{i} \xrightarrow{\gamma} \delta$
$\mathbf{if}\ t\ \overline{\mathbf{is}\,p_i\ \mathbf{then}\ t_i} : \mathcal{T} \rightsquigarrow \mathsf{pmatch}_\mathsf{A}\ t\ \overline{\lambda x_{i1}.\ \ldots\lambda x_{in}.\,t_i}$	$\mathsf{pmatch}_\mathsf{A} : \forall\overline{a}, \gamma\,.\,\mathsf{A}[\overline{a}] \to \overline{(\forall\overline{\beta}.\ \mathcal{T}_{i1} \to \ldots \to \mathcal{T}_{in} \xrightarrow{\gamma} \mathcal{T}\ )}^{i} \xrightarrow{\gamma} \mathcal{T}$

$$\boxed{\Xi \vdash \tau \leq \tau}$$

$$\textsc{S-Top}\pm\ \frac{}{\Xi \vdash \tau \leq^{\pm} \top^{\pm}} \qquad \textsc{S-Refl}\ \frac{}{\Xi \vdash \tau \leq \tau} \qquad \textsc{S-Hyp}\ \frac{(\tau \leq \sigma) \in \Xi}{\Xi \vdash \tau \leq \sigma} \qquad \textsc{S-Assum}\ \frac{\Xi \rhd (\tau \leq \sigma) \vdash \tau \leq \sigma}{\Xi \vdash \tau \leq \sigma} \qquad \textsc{S-Compl}\pm\ \frac{}{\Xi \vdash \top^{\pm} \leq^{\pm} \tau \vee^{\pm} \neg\tau}$$

$$\textsc{S-AndOrL}\pm\ \frac{}{\Xi \vdash \tau_1 \wedge^{\pm} \tau_2 \leq^{\pm} \tau_1} \qquad \textsc{S-AndOrR}\pm\ \frac{}{\Xi \vdash \tau_1 \wedge^{\pm} \tau_2 \leq^{\pm} \tau_2} \qquad \textsc{S-AndOr}\pm\ \frac{\Xi \vdash \tau \leq^{\pm} \sigma_1 \qquad \Xi \vdash \tau \leq^{\pm} \sigma_2}{\Xi \vdash \tau \leq^{\pm} \sigma_1 \wedge^{\pm} \sigma_2} \qquad \textsc{S-Trans}\ \frac{\Xi \vdash \tau_0 \leq \tau_1 \qquad \Xi \vdash \tau_1 \leq \tau_2}{\Xi \vdash \tau_0 \leq \tau_2}$$

$$\textsc{S-Distrib}\pm\ \frac{}{\Xi \vdash \tau \wedge^{\pm} (\tau_1 \vee^{\pm} \tau_2) \leq^{\pm} (\tau \wedge^{\pm} \tau_1) \vee^{\pm} (\tau \wedge^{\pm} \tau_2)} \qquad \textsc{S-FunMrg}\pm\ \frac{}{\Xi \vdash \tau_1 \xrightarrow{\tau_5} \tau_2 \wedge^{\pm} \tau_3 \xrightarrow{\tau_6} \tau_4 \leq^{\pm} (\tau_1 \vee^{\pm} \tau_3) \xrightarrow{\tau_5 \wedge^{\pm} \tau_6} (\tau_2 \wedge^{\pm} \tau_4)}$$

$$\textsc{S-Fun}\ \frac{\begin{array}{c}\lhd\Xi \vdash \tau_1 \leq \tau_2 \\ \lhd\Xi \vdash \tau_3 \leq \tau_4 \\ \lhd\Xi \vdash \tau_5 \leq \tau_6\end{array}}{\Xi \vdash \tau_2 \xrightarrow{\tau_5} \tau_3 \leq \tau_1 \xrightarrow{\tau_6} \tau_4} \qquad \textsc{S-Ctor}\ \frac{\overline{\lhd\Xi \vdash \sigma_i \leq \tau_i,\ \tau'_i \leq \sigma'_i}^{i}}{\Xi \vdash \mathsf{A}[\overline{\mathbf{in}\,\tau_i\ \mathbf{out}\,\tau'_i}^{i}\,] \leq \mathsf{A}[\overline{\mathbf{in}\,\sigma_i\ \mathbf{out}\,\sigma'_i}^{i}\,]} \qquad \textsc{S-CtorBot}\ \frac{\mathsf{A}_1 \neq \mathsf{A}_2}{\Xi \vdash \mathsf{A}_1[\overline{a}] \wedge \mathsf{A}_2[\overline{b}] \leq \bot}$$

$$\textsc{S-CFBot}\ \frac{}{\Xi \vdash \mathsf{A}[\overline{a}] \wedge (\tau \xrightarrow{\varphi} \sigma) \leq \bot} \qquad \textsc{S-CtorMrg}\pm\ \frac{}{\Xi \vdash \mathsf{A}[\overline{\mathbf{in}\,\tau_i\ \mathbf{out}\,\tau'_i}^{i}\,] \wedge^{\pm} \mathsf{A}[\overline{\mathbf{in}\,\sigma_i\ \mathbf{out}\,\sigma'_i}^{i}\,] \leq^{\pm} \mathsf{A}[\overline{\mathbf{in}\,(\tau_i \vee^{\pm} \sigma_i)\ \mathbf{out}\,(\tau'_i \wedge^{\pm} \sigma'_i)}^{i}\,]}$$

Fig. 3. Subtyping rules.

to ensure nested regions are disjoint from this one, we append α to ζ. The body t is allowed to refer to local region variable x to manipulate mutable cells, which is reflected in the effect type of t, $\varphi \vee \alpha$. We explicitly show and highlight the premise $\alpha \notin FV(\Gamma) \cup FV(\zeta) \cup FV(\tau) \cup FV(\varphi)$ in *gray* to emphasize that the overall effect of the region φ does *not* include α (which would be out of scope), which reflects that all corresponding mutable operations are *encapsulated* inside the region's local scope and are not observable from the outside. If a region or a location is leaked we must widen types involving local type variable α (i.e., extrusion, which will be explained in §4.4) to get rid of it before exiting the region, or the result type τ or effect φ will be ill-formed since $\alpha \notin \Gamma$.

Instead of adding specific typing rules for the constructs related to mutable references and data types, we encode these term forms as built-in functions, listed in Table 1. To allocate a new reference cell, a region instance is required, but no region is needed to dereference and set mutable references. For each constructor C, we can generate a construction function $\mathsf{construct}_\mathsf{C}$. For each data type A, we also generate a pattern matching function $\mathsf{match}_\mathsf{A}$ based on its type parameters and constructors. For the last function $\mathsf{pmatch}_\mathsf{A}$, we propagate the type annotation $\mathcal{T}$ into each branch function since we cannot instantiate a type variable to non-monomorphic types.

3.3 Subtyping Rules

Figure 3 presents the subtyping rules of $\lambda^{!\perp}$. The first part of the rules, ranging from S-Top± to S-Distrib±, is essentially the same as in $\lambda^{\neg}$, the core language of MLstruct. Readers might want

<u>Values and Contexts</u>

Region label	$\mathsf{r} \in R$	*Location*	$\ell \in L$
Value	$v ::= \ell_{\mathsf{r}} \mid \mathsf{r} \mid \langle \lambda x.\, t,\, \gamma \rangle \mid \mathsf{C}(\overline{v})$	*Runtime context*	$\gamma ::= \epsilon \mid \gamma\ (x \mapsto v)$
Store	$\psi ::= \epsilon \mid \psi\ (\mathsf{r} \mapsto \mu)$	*Memory*	$\mu ::= \epsilon \mid \mu\ (\ell_{\mathsf{r}} \mapsto v)$
Store typing context	$\Psi ::= \epsilon \mid \Psi\ (\ell_{\mathsf{r}} : \tau) \mid \Psi\ (\mathsf{r} : \alpha)$	*Result*	$\mathbb{R} ::= \mathbf{val}(\psi, v) \mid \boldsymbol{err} \mid \mathbf{kill}$

Fig. 4. Values and contexts.

to refer to the original material for further explanations [13, 68], with the most detailed account recently provided by Chau and Parreaux [14], as we only give a brief overview here. Rules S-Top±, S-Refl, and S-Trans are fundamental properties of subtyping. The addition of rules S-Compl±, S-AndOrL±, S-AndOrR±, S-AndOr±, and S-Distrib± make the subtyping lattice a *Boolean lattice* (or Boolean algebra). Rule S-Hyp leverages the type bounds in the current context. $\lambda^{!\perp}$ supports recursive types via bounds on quantified variables. However, it is impossible to derive subtyping relation on recursive types only with S-Hyp, We then borrow S-Assum and later modality $\rhd$ from previous work [2, 66, 68] to store subtyping assumptions in the context. The later modality prevents the assumptions that possibly do not hold yet until recursive types are expanded from immediate use by S-Hyp. The dual notation $\lhd$ makes assumptions available for S-Hyp, which can be observed in S-Fun and S-Ctor. It reflects that we must prevent forall-quantified type variables from being used recursively before entering a type constructor. Example 3.1 further demonstrates the subtyping derivation of two recursive types using later modality.

Rules S-Fun and S-Ctor are standard depth subtyping rules. S-FunMrg± and S-CtorMrg± are algebraic rules that merge two functions or data types. S-CtorBot is used to show that the intersection $\mathsf{A}_1[\overline{a}] \wedge \mathsf{A}_2[\overline{b}]$ of two distinct data types A_1 and A_2 is empty (i.e., subtypes $\perp$). Note that this implies $\mathsf{A}_1[\overline{a}] \leq \neg\mathsf{A}_2$ by the Boolean rules and by taking $\overline{b} = \overline{\mathbf{in}\perp\mathbf{out}\top}$. All these rules are essentially similar to the rules found $\lambda^{\neg}$, although the syntax of data and function types is slightly different. We additionally require that a data type is disjoint with a function type in rule S-CFBot, which can also prevent the inference algorithm from backtracking. The subtyping relations of region and mutable reference types simply follow the rules for normal data types.

Example 3.1. To better exemplify the subtyping rules and use of later modality, let's consider to derive a subtyping $\alpha \leq \beta$, where $\alpha \leq \mathsf{A}[\mathbf{out}\,\alpha] \in \Xi$ and $\mathsf{A}[\mathbf{out}\,\beta] \leq \beta \in \Xi$. Let $\Xi' = \Xi \rhd (\alpha \leq \beta)$, the derivation is demonstrated as follows:

$$\textsc{S-Assum}\ \dfrac{\textsc{S-Trans}\ \dfrac{\textsc{S-Ctor}\ \dfrac{\textsc{S-Hyp}\ \lhd\Xi' \vdash \alpha \leq \beta}{\Xi' \vdash \mathsf{A}[\mathbf{out}\,\alpha] \leq \mathsf{A}[\mathbf{out}\,\beta]} \qquad \textsc{S-Hyp}\ \begin{matrix}\Xi' \vdash \alpha \leq \mathsf{A}[\mathbf{out}\,\alpha] \\ \Xi' \vdash \mathsf{A}[\mathbf{out}\,\beta] \leq \beta\end{matrix}}{\Xi' \vdash \alpha \leq \beta}}{\Xi \vdash \alpha \leq \beta}$$

Notice that S-Assum must guard the assumption $\alpha \leq \beta$, or we can append arbitrary subtyping relations to Ξ and use them immediately, which breaks the soundness of the system. To make use of $\rhd(\alpha \leq \beta)$, we need to go through a type constructor. S-Ctor eliminates the $\rhd$ guardness by prepending the dual notation $\lhd$. Then $\alpha \leq \beta$ is available for S-Hyp. Finally, we apply S-Trans twice to conclude $\Xi' \vdash \alpha \leq \beta$.

3.4 Values and Operational Semantics

The values and extra contexts of $\lambda^{!\perp}$ are presented in Figure 4. Values are either locations, regions, closures, or data type instances. A location also carries the region label to which it belongs. γ stands for value substitution contexts, mapping identifiers to values, while ψ stands for runtime

Syntax	*Skolem-like*	$\hat{\alpha} ::= \alpha \mid \omega$	*Type variable-like* $\quad \nu ::= \alpha^n \mid \hat{\alpha}$
	Skolem context	$\hat{\Sigma} ::= \epsilon \mid \hat{\Sigma}\, V \mid \hat{\Sigma}\,(\alpha \leq^{\pm} \tau) \mid \hat{\Sigma}\, \bullet$	
	Non-function type	$\tau^{\not\to} ::= \nu \mid \mathsf{A}[\overline{a}] \mid \top^{\pm} \mid \tau \vee^{\pm} \tau \mid \neg\tau$	
	Non-function term	$t^{\not\to} ::= t\ t \mid x \mid t : \mathcal{T} \mid \mathbf{let}\, x = t\, \mathbf{in}\, t \mid \mathsf{C}(\,\overline{t}\,) \mid \mathbf{region}\, x\, \mathbf{in}\, t$	

Fig. 5. Syntax extension for type inference. All other syntactic forms are as in Figure 1.

stores, mapping regions to memory pages μ that further map locations to values. The value typing judgment $\Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}$ says a value can be typed as $\mathcal{T}$ in typing context Γ with storing typing context Ψ and region accumulation ζ. Most value typing rules are standared except we type a dead region instance $\mathsf{r} \notin dom(\Psi)$ to $\mathsf{Region}[\mathbf{out}\,\neg\zeta]$ that can be later upcasted to $\mathsf{Region}[\mathbf{out}\,\top]$. We give the formal definition of value typing in App. C.3.

We give our big-step semantics in the functional style of Radanne et al. [77]. eval $\gamma\ \psi\ k\ t$ evaluates a term t with a runtime environment γ, a store ψ, and a fuel k, yielding a result $\mathbb{R}$. A result can be either a pair of store and value, an error ***err***, or a timeout **kill**. If a given term yields some value successfully, the eval function returns both the value and the store context updated by the term. If a runtime error happens due to, for example, reading a deallocated mutable reference, the eval function returns ***err***. Finally, if the fuel k runs out and the evaluation has not finished, a **kill** will be returned. Since evaluation rules are standard, the formal definition is given in App. C.4 due to lack of space.

3.5 Metatheory

In this section, we present the metatheory of $\lambda^{!\bot}$.

The soundness of subtyping guarantees that we never derive an ill-formed subtyping relation (e.g., $\tau_1 \xrightarrow{\varphi} \tau_2 \leq \mathsf{A}[\sigma_1, \sigma_2]$) from a consistent subtyping context, which is crucial to the soundness of $\lambda^{!\bot}$. The subtyping consistency proof in $\lambda^{\neg}$ [68] is comprehensive but quite complex. We propose a simpler proof by constructing homomorphisms from the Boolean algebra of types to some other Boolean algebras. The formal definition of subtyping soundness and the proof are presented in App. D.2.

The soundness of $\lambda^{!\bot}$ is given in Theorem 3.2. We adopt the methodology of Ernst et al. [25]: if a term is well-typed, given any step k, the evaluation will not produce an error and the yielded value has the same type if it terminates. Besides, we also need to ensure a well-typed term's effect can accurately reflect this term's operations on heaps (i.e., effect soundness): if the term t manipulates some region r : $\mathsf{Region}[\mathbf{out}\,\alpha]$, then α is a subtype of the effect type of t. Then by restricting effects to be a subtype of current alive region accumulation ζ, we can wipe out all operations on dead regions and locations statically. We require the whole program to be pure to guarantee the absence of memory leakage. This $\bot$ effect can be easily extended to primitive effects like io in the implementation.

Theorem 3.2 (Soundness). *Given $\mathcal{D}$ **wf**, if $\vdash t : \mathcal{T}\ !\ \bot$, for all k, if $\mathbb{R} = eval\ \epsilon\ \epsilon\ k\ t$ and $\mathbb{R} \neq$ **kill**, then $\mathbb{R} = \mathbf{val}(\epsilon, v)$ and $\vdash v : \mathcal{T}$.*

We present the full soundness proofs in App. D.

4 Type Inference

In this section, we present the type inference algorithm of InvalML.

We first adapt the syntax of Figure 1 for type inference in Figure 5. We introduce *skolems*, a special kind of type variables that are too polymorphic to be further constrained. Skolems correspond to type variables in the declarative system, so we keep their syntax unchanged. We write $\hat{\alpha}$ for either a skolem or an outer scope variable. v is extended with type variables α^n. These type variables are generated on the fly and can be constrained further. Since they will not appear in the typing context and be delimited by the context separators $\bullet$, we track their polymorphic level n in the superscript. The skolem context $\hat{\Sigma}$ maintains skolems and their bounds. To preserve the *determinism* of our algorithmic rules, We use $\tau^{\not\to}$ and $t^{\not\to}$ to denote non-function types and non-function terms respectively. Polymorphic types are considered *equal modulo renaming and level changes of their quantified type variables*, so we will for instance be able to rename the quantified variables of the same polytype on the fly and instantiate it to different fresh variables at different levels each time, without needing an explicit substitution. We assume that all type annotations are well-formed for the sake of simplicity.

4.1 Type Inference Rules

The type inference rules are presented in Figure 6. The rules are now *algorithmic* as the never guess types and instead infer constraints. The type inference judgment produces *constraints* Ξ (written as $\Rightarrow \Xi$). I-Var is standard. Rules I-Abs1 and I-App1 show that even in the absence of type annotation, we are still able to type check abstractions and applications, albeit with monomorphic types. When creating a new fresh type variable in I-Abs1 and I-App1, we pick the level as the typing context Γ implies, written as $\boldsymbol{lv}(\Gamma)$. Rules I-Abs2 and I-App2 are almost standard for a bidirectional system, except we generate a constraint $\varphi' \leq \varphi$ to check if the effect raised by t satisfies the requirement of the given annotation φ in I-Abs2. Notice that in I-App2 and I-Let, we use unions to bring the effects from subterms and functions together, instead of implicitly upcasting them in T-App and T-Let. I-Asc1 and I-Asc2 constrain inferred monomorphic type σ with a given monomorphic annotation. We used two separate ascription rules to preserve the *determinism* of our algorithmic rules. If a lambda term is given a function type annotation, we can only use the I-Abs2 rule, rather than an ascription rule.

Example 4.1. Consider the following term $f\ a$. If the type of f is a function type in the context, for example $\mathsf{Int} \to \mathsf{Int}$, the algorithm will pick I-App2 to reuse the type information, and a will be checked against type Int. However, for the term $\lambda f.\, f\ 42$, f's type is a fresh type variable α^n allocated by I-Abs1. Then we must pick I-App1 to generate the constraint $\alpha^n \leq \mathsf{Int} \to \beta^n$.

The $\Xi, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma$ judgment (presented in §4.3) solves constraint $\tau \leq \sigma$, yielding new bounds Σ as an output, while $\Xi, \hat{\Sigma} \vdash \Sigma \Rightarrow \Sigma'$ judgment solves all constraints given in Σ one by one. It is notably used to ensure the consistency of the bounds of the ascribed polymorphic type in I-Gen, producing bounds Σ_0, which are discarded (as they only serve to witness consistency). I-Gen then solves Ξ and $\varphi \leq \bot$ immediately to wipe out type level violations. I-Inst implicitly substitutes all quantified type variables with fresh ones (except the outer scope variable ω that should be substituted with ζ) and uses the underlying type. This rule will only trigger when the current inferred type is polymorphic but a monomorphic type or a higher-ranked function type is expected (and it may result in further uses of instantiation down the line). Rule I-Region is similar to I-Gen: it solves Ξ, $\varphi \leq \gamma^n \vee \alpha$, and $\tau \leq \beta^n$ immediately to prevent polymorphism leakage. Notably, solving $\varphi \leq \gamma^n \vee \alpha$ will eliminate positive α and widen negative α to $\bot$ in φ. The former ensures that γ^n will only maintain effects whose levels do not exceed n (a reminiscent of effect masking in other systems), the latter will widen unexpected effects to $\top$.

Example 4.2. Consider the following term: **region** x **in** t. Assume that I-Region allocates α for x and the effect of t is $\varphi = \alpha \vee \alpha_0$ for some α_0 allocated by an outer region. α now is in a *positive* position. Then I-Region will generate and solve the constraint $\varphi \leq \gamma^n \vee \alpha$. Solving this constraint

$\boxed{\Gamma, \zeta \vdash t : \mathcal{T} \,!\, \varphi \Rightarrow \Xi}$ $\quad \delta ::= \cdot \mid \Downarrow \quad$ Notation: $\Gamma, \zeta \vdash_{\Downarrow} t : \mathcal{T} \,!\, \varphi \Rightarrow \Xi$ means $\Gamma, \zeta \vdash (t : \mathcal{T}) : \mathcal{T} \,!\, \varphi \Rightarrow \Xi$

I-Var
$$\frac{\Gamma(x) = \mathcal{T}}{\Gamma, \zeta \vdash x : \mathcal{T} \,!\, \bot \Rightarrow \epsilon}$$

I-Abs1
$$\frac{\alpha^n \ \textbf{\textit{fresh}} \qquad n = \textbf{\textit{lv}}(\Gamma) \qquad \Gamma\ (x : \alpha^n), \zeta \vdash t : \mathcal{T} \,!\, \varphi \Rightarrow \Xi}{\Gamma, \zeta \vdash (\lambda x.\, t) : \alpha^n \xrightarrow{\varphi} \mathcal{T} \,!\, \bot \Rightarrow \Xi}$$

I-Abs2
$$\frac{\Gamma\ (x : \mathcal{T}_1), \zeta \vdash_{\Downarrow} t : \mathcal{T}_2 \,!\, \varphi' \Rightarrow \Xi}{\Gamma, \zeta \vdash_{\Downarrow} \lambda x.\, t : \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2 \,!\, \bot \Rightarrow \Xi\ (\varphi' \leq \varphi)}$$

I-App1
$$\frac{\Gamma, \zeta \vdash t_1 : \tau_1^{\nrightarrow} \,!\, \varphi_1 \Rightarrow \Xi_1 \qquad n = \textbf{\textit{lv}}(\Gamma) \qquad \Gamma, \zeta \vdash t_2 : \tau_2 \,!\, \varphi_2 \Rightarrow \Xi_2 \qquad \beta^n, \gamma^n \ \textbf{\textit{fresh}}}{\Gamma, \zeta \vdash t_1\ t_2 : \beta^n \,!\, \gamma^n \vee \varphi_1 \vee \varphi_2 \Rightarrow \Xi_1\ \Xi_2\ (\tau_1^{\nrightarrow} \leq \tau_2 \xrightarrow{\gamma^n} \beta^n)}$$

I-App2
$$\frac{\Gamma, \zeta \vdash t_1 : \mathcal{T} \xrightarrow{\varphi} \mathcal{S} \,!\, \varphi_1 \Rightarrow \Xi_1 \qquad \Gamma, \zeta \vdash_{\Downarrow} t_2 : \mathcal{T} \,!\, \varphi_2 \Rightarrow \Xi_2}{\Gamma, \zeta \vdash t_1\ t_2 : \mathcal{S} \,!\, \varphi \vee \varphi_1 \vee \varphi_2 \Rightarrow \Xi_1\ \Xi_2}$$

I-Let
$$\frac{\Gamma, \zeta \vdash t_1 : \mathcal{T}_1 \,!\, \varphi_1 \Rightarrow \Xi_1 \qquad \Gamma\ (x : \mathcal{T}_1), \zeta \vdash_{\delta} t_2 : \mathcal{T}_2 \,!\, \varphi_2 \Rightarrow \Xi_2}{\Gamma, \zeta \vdash_{\delta} \textbf{let}\, x = t_1\, \textbf{in}\, t_2 : \mathcal{T}_2 \,!\, \varphi_1 \vee \varphi_2 \Rightarrow \Xi_1\ \Xi_2}$$

I-Asc1
$$\frac{\Gamma, \zeta \vdash t : \sigma \,!\, \varphi \Rightarrow \Xi}{\Gamma, \zeta \vdash_{\Downarrow} t : \tau^{\nrightarrow} \,!\, \varphi \Rightarrow \Xi\ (\sigma \leq \tau^{\nrightarrow})}$$

I-Asc2
$$\frac{\Gamma, \zeta \vdash t^{\nrightarrow} : \sigma \,!\, \varphi \Rightarrow \Xi}{\Gamma, \zeta \vdash_{\Downarrow} t^{\nrightarrow} : \tau \,!\, \varphi \Rightarrow \Xi\ (\sigma \leq \tau)}$$

I-Gen
$$\frac{\epsilon, sub(\Gamma) \vdash \Sigma \Rightarrow \Sigma_0 \qquad \Gamma \bullet V\ \Sigma, \zeta \vee \omega \vdash_{\Downarrow} t : \mathcal{T} \,!\, \varphi \Rightarrow \Xi \qquad \textbf{\textit{err}} \notin \Sigma_0 \qquad \omega \in V \qquad \epsilon, sub(\Gamma \bullet V\ \Sigma) \vdash \Xi\ (\varphi \leq \bot) \Rightarrow \Sigma_1}{\Gamma, \zeta \vdash_{\Downarrow} t : \forall V\{\Sigma\}.\, \mathcal{T} \,!\, \bot \Rightarrow \Sigma_1}$$

I-Inst
$$\frac{\omega \in V \qquad \rho(\omega) = \zeta \qquad V \setminus \omega \ \textbf{\textit{fresh}} \qquad \Gamma, \zeta \vdash t : \forall V\{\Sigma\}.\, \mathcal{T} \,!\, \varphi \Rightarrow \Xi}{\Gamma, \zeta \vdash t : \rho(\mathcal{T}) \,!\, \varphi \Rightarrow \Xi\ \rho(\Sigma)}$$

I-Region
$$\frac{n = \textbf{\textit{lv}}(\Gamma) \qquad \alpha, \beta^n, \gamma^n \ \textbf{\textit{fresh}} \qquad \Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)\ (x : \mathsf{Region}[\textbf{out}\, \alpha]), \zeta \vee \alpha \vdash t : \tau \,!\, \varphi \Rightarrow \Xi \qquad \epsilon, sub(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash \Xi\ (\varphi \leq \gamma^n \vee \alpha)\ (\tau \leq \beta^n) \Rightarrow \Sigma}{\Gamma, \zeta \vdash \textbf{region}\, x\, \textbf{in}\, t : \beta^n \,!\, \gamma^n \Rightarrow \Sigma}$$

Fig. 6. Type inference rules.

will wipe out α in φ and yield $\alpha_0 \leq \gamma^n$. However, if $\varphi = \neg\alpha$, which means we are trying to access a dead region and α now is in a *negative* position, solving the constraint will finally yield $\top \leq \gamma^n$, which will be rejected.

4.2 Normal Forms

To solve constraints, we follow Pearce's approach [70] to solve a constraint $\tau_1 \wedge \neg\tau_2 \leq \bot$ that is equivalent to $\tau_1 \leq \tau_2$. In order to solve constraints without backtracking, we need to first put the involved types into normal form. We adopt a *reduced disjunctive normal form* (RDNF) inspired by that of $\lambda^{\neg}$ [68]:

$$\mathbb{D} ::= \bot \mid \mathbb{C} \mid \mathbb{D} \vee \mathbb{C} \qquad\qquad \mathbb{C} ::= I \wedge \neg U \mid \mathbb{C} \wedge v \mid \mathbb{C} \wedge \neg v$$

$$I ::= \top \mid \mathbb{D} \xrightarrow{\mathbb{D}} \mathbb{D} \mid \mathsf{A}[\overline{\textbf{in}\, \mathbb{D}\, \textbf{out}\, \mathbb{D}}] \qquad\qquad U ::= \bot \mid \mathbb{D} \xrightarrow{\mathbb{D}} \mathbb{D} \mid U \vee \mathsf{A}[\overline{\textbf{in}\, \mathbb{D}\, \textbf{out}\, \mathbb{D}}]$$

We sort *conjunctive* forms (i.e., $\mathbb{C}$) internally so that type variables with higher polymorphic levels will be picked first, while skolems and outer scope variables will be picked last when we solve the constraints. For example, a conjunctive form $\mathsf{A} \wedge \neg\bot \wedge \alpha^1 \wedge \neg\beta \wedge \neg\gamma^3$ will be reordered to $\mathsf{A} \wedge \neg\bot \wedge \neg\gamma^3 \wedge \alpha^1 \wedge \neg\beta$. This can reduce the number of unnecessary extrusions, as we shall see later in Example 4.5. We define the function $dnf(\tau) = \mathbb{D}$ in App. C, which constructs a *disjunctive* normal form from a given type τ. Take $\tau = (\mathsf{A}[\alpha^1] \to \mathsf{A}[\alpha^1]) \wedge \neg\beta^1$ as an example. $dnf(\tau)$ yields $(\mathsf{A}[\top \wedge \neg\bot \wedge \alpha^1] \wedge \neg\bot \to \mathsf{A}[\top \wedge \neg\bot \wedge \alpha^1] \wedge \neg\bot) \wedge \neg\bot \wedge \neg\beta^1$. *dnf* also *reduces* intersections (or unions) that have no useful information to $\bot$ (or $\top$). For instance, $\alpha \wedge \neg\alpha$ will be simplified to $\bot$. The algorithm is also inspired by $\lambda^{\neg}$ [68] but simpler.

$$\boxed{\varrho^{\pm}_{\hat{\Sigma}}(\tau)} : \tau \qquad \varrho_{\hat{\Sigma}}(\tau) = \varrho^{+}_{\hat{\Sigma}}(\tau)$$
$$\varrho^{\pm}_{\hat{\Sigma}}(\alpha) = \alpha \wedge^{\pm} \varrho^{\pm}_{\hat{\Sigma}}(ub^{\pm}_{\hat{\Sigma}}(\alpha)) \qquad \varrho^{\pm}_{\hat{\Sigma}}(\tau_1 \wedge^{\diamond} \tau_2) = \varrho^{\pm}_{\hat{\Sigma}}(\tau_1) \wedge^{\diamond} \varrho^{\pm}_{\hat{\Sigma}}(\tau_2) \qquad \varrho^{\pm}_{\hat{\Sigma}}(\neg\tau) = \neg\varrho^{\mp}_{\hat{\Sigma}}(\tau)$$
$$\varrho^{\pm}_{\hat{\Sigma}}(\tau) = \tau \quad \textit{Otherwise}$$

Fig. 7. Definition of skolem bound inlining $\varrho_{\hat{\Sigma}}$ function.

4.3 Constraint Solving Rules

The constraint-solving rules are presented in Figure 8. Judgment $\Xi, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma$ constrains $\tau_1 \leq \tau_2$ given skolem bounds $\hat{\Sigma}$, assuming subtyping relationships Ξ, and producing new bounds Σ. Most rules straightforwardly follow those of $\lambda^{\neg}$ [68], decomposing disjunctive normal forms into conjunctive ones and solving conjunctive normal forms by (1) matching type constructors in positive positions against those in negative positions, and (2) checking the new bounds of type variables against existing bounds to guarantee consistency of the output. We highlight the crucial differences in *gray*. Once a constraint cannot be found in the assumptions, C-Assum appends it into Ξ with a later guard to prevent the immediate use and construct the corresponding RNDF by calling *dnf* function. Notably, we expand top-level skolems' bounds to propagate them. The function $\rho_{\hat{\Sigma}}(\tau)$, defined in Figure 7, substitutes top-level skolems with either intersection with their upper bounds or union with their lower bounds, depending on the position. The upper bounds and lower bounds are stored in the $\hat{\Sigma}$ context and can be retrieved by functions *ub* and *lb*, defined as follows:

Definition 4.3 (Upper and lower bounds). We use the following *upper* and *lower* bounds functions. For simplicity, we also use $\pm$ to treat ub/lb symmetrically.

$$\boxed{lb^{\pm}_{\Xi}(v)} : \tau \qquad \boxed{ub^{\pm}_{\Xi}(v)} : \tau \qquad lb_{\Xi}(v) = lb^{+}_{\Xi}(v) \qquad ub_{\Xi}(v) = ub^{+}_{\Xi}(v) \qquad ub^{\pm}_{\Xi}(v) = lb^{\mp}_{\Xi}(v)$$
$$lb_{\epsilon}(v) = \bot \qquad lb^{\pm}_{\Xi\ \tau \leq^{\pm} v}(v) = \tau \vee^{\pm} lb^{\pm}_{\Xi}(v) \qquad lb^{\pm}_{\Xi\ \tau \leq^{\pm} \sigma}(v) = lb^{\pm}_{\Xi}(v) \quad \text{if } \sigma \neq v$$
$$lb^{\pm}_{\Xi\ \boldsymbol{err}}(v) = lb^{\pm}_{\Xi\ \triangleright(\tau \leq \sigma)}(v) = lb^{\pm}_{\Xi\ \bullet}(v) = lb^{\pm}_{\Xi\ v}(v) = lb^{\pm}_{\Xi}(v)$$

After inlining, we can drop top-level skolems when solving constraints without losing any information, as shown in C-Sk.

Example 4.4. Consider the following constraint: $\alpha \ll \mathsf{A}[\mathbf{out}\,\top]$ for some α and A, where $(\alpha \leq \mathsf{A}[\mathbf{out}\,\bot]) \in \hat{\Sigma}$. Then $\varrho_{\hat{\Sigma}}(\alpha \wedge \neg\mathsf{A}[\mathbf{out}\,\top]) = \alpha \wedge \mathsf{A}[\mathbf{out}\,\bot] \wedge \neg\mathsf{A}[\mathbf{out}\,\top]$. C-Sk will drop α and yield $\mathsf{A}[\mathbf{out}\,\bot] \wedge \neg\mathsf{A}[\mathbf{out}\,\top]$, which is equivalent to $\mathsf{A}[\mathbf{out}\,\bot] \leq \mathsf{A}[\mathbf{out}\,\top]$. This constraint can later be handled by C-Ctor1. Without inlining, one can only get $\mathsf{A}[\mathbf{out}\,\top] \leq \bot$ after dropping α, which is impossible and will be rejected.

Judgment $\Xi, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma$ solves a RDNF-normalized constraint $\mathbb{D} \leq \bot$. Most rules follow those of $\lambda^{\neg}$ [68] and are straightforward, except C-Var3/4. We define the function $\boldsymbol{lv}(\tau, \hat{\Sigma})$ for the polymorphic level of τ, according to the context $\hat{\Sigma}$. The level of a type is taken to be the maximum level of its parts. When using a more polymorphic type τ to bind a less polymorphic type variable α^m, we will make sure to *extrude* the excessively-polymorphic type variables of τ to prevent any polymorphism leakage. Take C-Var3 as an example. The extrusion $(\hat{\Sigma}, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma', \tau)$ widens those excessively-polymorphic type variables to their bounds in $\neg\mathbb{C}$ to ensure τ's level is at most m. A recursive call $\Xi, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma''$ is necessary for further extruding too polymorphic type variables in widened types. Then C-Var3 propagates the new upper bound τ to existing lower bounds of α^m to guarantee the output is consistent, similarly to the process in C-Var1. Notice that the new upper bound $\alpha^m \leq \tau$ is appended to the context to ensure new lower bounds generated by the recursive call can also be checked against τ, and our algorithm can terminate when there are bound cycles.

$$\boxed{\Xi, \hat{\Sigma} \vdash \Xi \Rightarrow \Sigma}$$

C-Done
$$\frac{}{\Xi, \hat{\Sigma} \vdash \epsilon \Rightarrow \epsilon}$$

C-NotDone
$$\frac{\Xi, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma_0 \qquad \Sigma_0\ \Xi, \hat{\Sigma} \vdash \Xi_0 \Rightarrow \Sigma_1}{\Xi, \hat{\Sigma} \vdash \Xi_0\ (\tau_1 \leq \tau_2) \Rightarrow \Sigma_0\ \Sigma_1}$$

$$\boxed{\Xi, \hat{\Sigma} \vdash \tau \ll \tau \Rightarrow \Sigma}$$

C-Hyp
$$\frac{(\tau_1 \leq \tau_2) \in \Xi\ \hat{\Sigma}}{\Xi, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \epsilon}$$

C-Assum
$$\frac{(\tau_1 \leq \tau_2) \notin \Xi\ \hat{\Sigma} \qquad \Xi \rhd (\tau_1 \leq \tau_2), \hat{\Sigma} \vdash dnf \circ \varrho_{\hat{\Sigma}}(\tau_1 \wedge \neg\tau_2) \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma}$$

$$\boxed{\Xi, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma}$$

C-Or
$$\frac{\Xi, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma \qquad \Sigma\ \Xi, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma'}{\Xi, \hat{\Sigma} \vdash \mathbb{D} \vee \mathbb{C} \Rightarrow \Sigma\ \Sigma'}$$

C-NotBot
$$\frac{}{\Xi, \hat{\Sigma} \vdash I \wedge \neg\bot \Rightarrow \boldsymbol{err}}$$

C-Bot
$$\frac{}{\Xi, \hat{\Sigma} \vdash \bot \Rightarrow \epsilon}$$

C-Ctor1
$$\frac{\overline{\lhd\Xi\ \overline{\Sigma_j}^{j\in 1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i2} \ll \mathbb{D}_{i1} \Rightarrow \Sigma_i}^{i} \qquad \overline{\lhd\Xi\ \overline{\Sigma_j}^{j}\ \overline{\Sigma'_k}^{k\in 1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i3} \ll \mathbb{D}_{i4} \Rightarrow \Sigma'_i}^{i}}{\Xi, \hat{\Sigma} \vdash \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i1}\,\mathbf{out}\,\mathbb{D}_{i3}}^{i}] \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i2}\,\mathbf{out}\,\mathbb{D}_{i4}}^{i}]) \Rightarrow \overline{\Sigma_i}\ \overline{\Sigma'_i}}$$

C-Ctor2
$$\frac{\Xi, \hat{\Sigma} \vdash \mathsf{A}_1[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}] \wedge \neg U \Rightarrow \Sigma \qquad \mathsf{A}_1 \neq \mathsf{A}_2}{\Xi, \hat{\Sigma} \vdash \mathsf{A}_1[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}] \wedge \neg(U \vee \mathsf{A}_2[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}]) \Rightarrow \Sigma}$$

C-Ctor3
$$\frac{\Xi, \hat{\Sigma} \vdash (\mathbb{D} \xrightarrow{\mathbb{D}} \mathbb{D}) \wedge \neg U \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash (\mathbb{D} \xrightarrow{\mathbb{D}} \mathbb{D}) \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}]) \Rightarrow \Sigma}$$

C-Ctor4
$$\frac{\Xi, \hat{\Sigma} \vdash \top \wedge \neg U \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \top \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}]) \Rightarrow \Sigma}$$

C-Fun1
$$\frac{\lhd\Xi, \hat{\Sigma} \vdash \mathbb{D}_3 \ll \mathbb{D}_1 \Rightarrow \Sigma \qquad \Sigma \lhd\Xi, \hat{\Sigma} \vdash \mathbb{D}_2 \ll \mathbb{D}_4 \Rightarrow \Sigma' \qquad \Sigma\ \Sigma' \lhd\Xi, \hat{\Sigma} \vdash \mathbb{D}_5 \ll \mathbb{D}_6 \Rightarrow \Sigma''}{\Xi, \hat{\Sigma} \vdash (\mathbb{D}_1 \xrightarrow{\mathbb{D}_5} \mathbb{D}_2) \wedge \neg(\mathbb{D}_3 \xrightarrow{\mathbb{D}_6} \mathbb{D}_4) \Rightarrow \Sigma\ \Sigma'\ \Sigma''}$$

C-Fun2
$$\frac{}{\Xi, \hat{\Sigma} \vdash \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}] \wedge \neg(\mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2) \Rightarrow \boldsymbol{err}}$$

C-Fun3
$$\frac{}{\Xi, \hat{\Sigma} \vdash \top \wedge \neg(\mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2) \Rightarrow \boldsymbol{err}}$$

C-Sk
$$\frac{\Xi, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \mathbb{C} \wedge \neg^{\pm}\hat{\alpha} \Rightarrow \Sigma}$$

C-Var1
$$\frac{\boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \leq m \qquad \Xi\ (\alpha^m \leq \neg\mathbb{C}), \hat{\Sigma} \vdash \mathsf{lb}_{\Xi}(\alpha^m) \ll \neg\mathbb{C} \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \mathbb{C} \wedge \alpha \Rightarrow \Sigma\ (\alpha^m \leq \neg\mathbb{C})}$$

C-Var2
$$\frac{\boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \leq m \qquad \Xi\ (\mathbb{C} \leq \alpha^m), \hat{\Sigma} \vdash \mathbb{C} \ll \mathsf{ub}_{\Xi}(\alpha^m) \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \mathbb{C} \wedge \neg\alpha^m \Rightarrow \Sigma\ (\mathbb{C} \leq \alpha^m)}$$

C-Var3
$$\frac{m < \boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \qquad (\hat{\Sigma}, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma', \tau) \qquad \Xi, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma'' \qquad \Xi\ \Sigma''\ (\alpha^m \leq \tau), \hat{\Sigma} \vdash \mathsf{lb}_{\Xi}(\alpha^m) \ll \tau \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \mathbb{C} \wedge \alpha^m \Rightarrow \Sigma\ \Sigma''\ (\alpha^m \leq \tau)}$$

C-Var4
$$\frac{m < \boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \qquad (\hat{\Sigma}, \mathbb{C}) \overset{(+,m)}{\rightsquigarrow} (\Sigma', \tau) \qquad \Xi, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma'' \qquad \Xi\ \Sigma''\ (\tau \leq \alpha^m), \hat{\Sigma} \vdash \tau \ll \mathsf{ub}_{\Xi}(\alpha^m) \Rightarrow \Sigma}{\Xi, \hat{\Sigma} \vdash \mathbb{C} \wedge \neg\alpha^m \Rightarrow \Sigma\ \Sigma''\ (\tau \leq \alpha^m)}$$

Fig. 8. Normal form constraining rules.

4.4 Extrusion Rules

Intuitively, it would seem that a type variable α^m cannot be constrained by a higher-level type τ, because the higher-level type variables and skolems in τ cannot be referred to by α^m at all. However, consider a constraint $\gamma^m \leq \neg\beta$, where $\beta \leq \neg\alpha$, $\boldsymbol{lv}(\alpha, \hat{\Sigma}) = m$, and $\boldsymbol{lv}(\beta, \hat{\Sigma}) = m + 1$. It could happen when we are handling two regions $r_1 : \mathsf{Region}[\mathbf{out}\,\alpha]$ and $r_2 : \mathsf{Region}[\mathbf{out}\,\beta]$, where r_2 is nested inside r_1. The constraint says γ^m should represent an outer region that is not indicated by β, which can be easily satisfied by having subtyping relation $\gamma^m \leq \alpha$, and then the constraint $\gamma^m \leq \neg\beta$ can hold by S-Trans since $\beta \leq \neg\alpha$ implies $\alpha \leq \neg\beta$. Instead of summarily rejecting constraints with level violations, *extrusion* widens such higher-level types to seek the proper intermediate types that can bridge both lower-level type variables and higher-level types via the transitivity of subtyping.

$$\boxed{(\hat{\Sigma}, \tau) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma, \tau)}$$

X-AndOr◇
$$\frac{(\hat{\Sigma}, \tau_1) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma_1, \tau_1') \qquad (\hat{\Sigma}, \tau_2) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma_2, \tau_2')}{(\hat{\Sigma}, \tau_1 \vee^{\diamond} \tau_2) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma_1\ \Sigma_2, \tau_1' \vee^{\diamond} \tau_2')}$$

X-Neg
$$\frac{(\hat{\Sigma}, \tau) \overset{(\mp,n)}{\rightsquigarrow} (\Sigma', \tau')}{(\hat{\Sigma}, \neg\tau) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma', \neg\tau')}$$

X-Fun
$$\frac{\begin{array}{c}(\hat{\Sigma}, \tau_1) \overset{(\mp,n)}{\rightsquigarrow} (\Sigma_1, \tau_1') \\ (\hat{\Sigma}, \tau_2) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma_2, \tau_2') \qquad (\hat{\Sigma}, \varphi) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma_3, \varphi')\end{array}}{(\hat{\Sigma}, \tau_1 \xrightarrow{\varphi} \tau_2) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma_1\ \Sigma_2\ \Sigma_3, \tau_1' \xrightarrow{\varphi'} \tau_2')}$$

X-Ctor
$$\frac{\overline{(\hat{\Sigma}, \tau_i) \overset{(\mp,n)}{\rightsquigarrow} (\overline{\Sigma_i}, \tau_i')} \qquad \overline{(\hat{\Sigma}, \sigma_i) \overset{(\pm,n)}{\rightsquigarrow} (\overline{\Sigma_i'}, \sigma_i')}}{(\hat{\Sigma}, \mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \sigma_i}]) \overset{(\pm,n)}{\rightsquigarrow} (\overline{\Sigma_i}\ \overline{\Sigma_i'}, \mathsf{A}[\overline{\textbf{in}\ \tau_i'\ \textbf{out}\ \sigma_i'}])}$$

X-Var
$$\frac{n < m \qquad (\alpha^m, \pm, \beta^n)\ \textbf{\textit{X-fresh}}}{(\hat{\Sigma}, \alpha^m) \overset{(\pm,n)}{\rightsquigarrow} (\alpha^m \leq^{\pm} \beta^n, \beta^n)}$$

X-Skolem
$$\frac{n < \textbf{\textit{lv}}(\hat{\alpha}, \hat{\Sigma}) \qquad (\hat{\alpha}, \pm, \beta^n)\ \textbf{\textit{X-fresh}}}{(\hat{\Sigma}, \hat{\alpha}) \overset{(\pm,n)}{\rightsquigarrow} (ub^{\pm}_{\hat{\Sigma}}(\hat{\alpha}) \leq^{\pm} \beta^n, \beta^n)}$$

X-Skip
$$\frac{\textbf{\textit{lv}}(\tau, \hat{\Sigma}) \leq n}{(\hat{\Sigma}, \tau) \overset{(\pm,n)}{\rightsquigarrow} (\epsilon, \tau)}$$

Fig. 9. Extrusion rules.

Notice that λ-lifting cannot address this level violation problem in its full generality since it does not deal with level-mismatched skolems.

The extrusion rules are presented in Figure 9. The computation $(\hat{\Sigma}, \tau) \overset{(\pm,n)}{\rightsquigarrow} (\Sigma, \sigma)$ takes four inputs: skolem context $\hat{\Sigma}$, the type τ to be extruded, the polarity of τ, and the target level n. It generates two outputs: an extruded type σ, where $\textbf{\textit{lv}}(\sigma, \hat{\Sigma}) = n$, and a new context Σ, maintaining the subtyping relations between the extruded fresh variables and original ones. X-AndOr◇, X-Neg, X-Fun, and X-Ctor simply propagate the extrusion to the subterms. We write $\mp$ to flip a polarity indicated by $\pm$. For example, if $\pm = +$, then $\mp = -$. X-Var and X-Skolem allocate fresh type variables by using $(v, \pm, \beta^n)$ ***X-fresh***, which ensures that for a given v, a polarity $\pm$, and a polymorphic level n, v is extruded to the same type variable β^n. This is crucial for the termination of the algorithm. Since skolems cannot be further constrained, X-Skolem directly assigns the upper (or lower) bounds of the skolem to the lower (or upper) bound of β^n. By contrast, X-Var assigns β^n as a new bound of the original type variable. Finally, X-Skip filter subterms whose levels are not higher than n.

Example 4.5. Suppose we have a constraint $\alpha^{42} \to \alpha^{42} \to \alpha^{42} \ll \beta^1$. It can be yielded by the first sub-derivation of I-App2. The corresponding normal form is $(\alpha^{42} \to \alpha^{42} \to \alpha^{42}) \wedge \neg\beta^1$. Only C-Var4 is available for this constraint and $\alpha^{42} \to \alpha^{42} \to \alpha^{42}$ must be extruded. X-Fun decomposes the function type and X-Var extrudes each α^{42} individually. We extrude the first α^{42} to a fresh type variable γ^1 and we have $\gamma^1 \leq \alpha^{42}$. Similarly, we can get another fresh type variable δ^1 and $\alpha^{42} \leq \delta^1$ for the third α^{42}. Notice that for the second α^{42}, it will not be extruded to a fresh type variable because the extrusion for α^{42} in negative position to level 1 has been cached and X-Var will directly return γ^1 for it. The re-constraining process $\Xi, \hat{\Sigma} \vdash \{\gamma^1 \leq \alpha^{42}, \gamma^1 \leq \alpha^{42}\} \Rightarrow \Sigma''$ will not extrude α^{42} again, thanks to the order of RDNF, as we mentioned in §4.2. If we placed γ^1 or δ^1 first, we would end up constraining a low-leveled type variable with a higher-level one, which would lead to extrusion to prevent α^{42} from leaking. The re-constraining will assign γ^1 to α^{42}'s lower bound δ^1 to α^{42}'s upper bound, and check δ^1 against γ^1, which finally yields $\gamma^1 \leq \delta^1$. Assume that β^1 has an upper bound $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool}$. We solve the constraint by propagating it to the extruded type $\gamma^1 \to \gamma^1 \to \delta^1$. Therefore, $\mathsf{Int} \leq \gamma^1$ by the first sub-derivation of C-Fun1. However, when solving $\sigma^1 \leq \mathsf{Bool}$, we get $\mathsf{Int} \leq \mathsf{Bool}$ by C-Var1. Finally, our algorithm throws an ***err*** by C-NotBot with $\mathsf{Bool} \wedge \neg\bot$.

4.5 Soundness and Completeness of Type Inference

In this section, we present the soundness and completeness metatheoretic results of our algorithm.

Theorem 4.6 (Soundness of type inference). *Given definitions $\mathcal{D}$ **wf**, if $\vdash t : \mathcal{T} \Rightarrow \Xi, \vdash \Xi \Rightarrow \Sigma$, and **err** $\notin \Sigma$, then $\Sigma \vdash t : \mathcal{T}$ and Σ **cons.**.*

Lemma 4.7 (Soundness of constraining). *If $\Sigma\, \hat{\Sigma}$ **cons.**, $\Sigma, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma'$, and **err** $\notin \Sigma'$, then $\Sigma\, \hat{\Sigma}\, \Sigma'$ **cons.** and $\Sigma\, \hat{\Sigma}\, \Sigma' \vdash \tau \leq \sigma$.*

Theorem 4.8 (Constraining termination). *For all $\mathcal{D}$ **wf**, $\Sigma\, \hat{\Sigma}$ **wf**, τ, and σ, $\Sigma, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma'$ for some Σ'.*

In the result below, we use the $\leq^{\forall}$ relation, a slight (and harmless) extension of $\leq$ to general types, defined in App. C (Figure 11). This is *not* a polymorphic subtyping relation. We use *fresh*(D) to indicate the fresh variables generated by the derivation D.

Theorem 4.9 (Completeness of type inference). *Given definitions $\mathcal{D}$ **wf**, if $\vdash t : \mathcal{T} \;!\; \bot$, then $\vdash t : \mathcal{S} \;!\; \varphi \Rightarrow \Xi, \vdash \Xi \Rightarrow \Sigma$, and there exists some type variable substitution ρ, such that $\epsilon \vDash \rho(\Sigma)$, $\vdash \rho(\mathcal{S}) \leq^{\forall} \mathcal{T}$, and $\vdash \rho(\varphi) \leq \bot$.*

Lemma 4.10 (Completeness of constraining). *If $\Sigma\, \hat{\Sigma}$ **cons.**, $\Sigma\, \hat{\Sigma} \vdash \rho(\tau_1) \leq \rho(\tau_2)$ for some type variable substitution ρ, $\Sigma\, \hat{\Sigma} \vDash \rho(\Sigma_0)$, then $\Sigma_0, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma_1$ (denoted as D), where **err** $\notin \Sigma_1$, $\Sigma\, \hat{\Sigma} \vDash \rho'(\Sigma_1)$, ρ' extends ρ, and $dom(\rho') \setminus dom(\rho) = fresh(D)$.*

We present the proofs of the above theorems and lemmas in App. D.

5 Applications and Extensions

We now present some additional applications and extensions of InvalML.

5.1 Case Studies

We have implemented several applications, including a dynamic programing example and a constraint solver, to demonstrate InvalML can correctly infer the types reject erroneous uses of mutable collections. In these two case studies, InvalML properly reasons about the disjointness among regions, including local regions inside function bodies and outer unkown regions. Unexpected mutable operations, e.g., modifying `ArrayList` during the iteration, are rejected to prevent the programs from runtime errors. These case studies justify the need for features like useful extrusion we mentioned in §2. We leave the detailed case studies for App. B due to the lack of space.

5.2 Effect and Exception Handlers

We now discuss how to encode general type-safe exceptions and effect handlers in InvalML. We define two functions, `throw` and `handle` as follows, to encode the exceptions and handlers:

```
fun throw: ∀ P, Q : (e: Exc[P, Q], payload: P) →{Q} ⊥
fun handle: ∀ P, Res, E : ( body: ∀ Q : (e: Exc[P, Q]) →{Q ∨ E} Res,
                            catch: P →{E} Res ) →{E} Res
```

The `throw` function takes an instance of exception and a payload. A function throwing an exception will carry the corresponding effect `Q`. The function `handle` has two parameters. The first one is a higher-ranked function that may throw an exception by using `e`, and the second one will catch the exception, process the payload and return a value. The `body` function can also have its own effect `E` and it is reflected in both the `catch`'s effects and the final effects of the function `handle`. Notice that the exception handler cannot be leaked outside the handler scope and does not appear in either `catch`'s or `handle`'s type. Assume `println` has type '`Str →{io} ()`' and `e` has type '`Exc[Str, Q]`'.

```
handle(e ⇒ ... throw(e, "oops") ... , msg ⇒ { println(msg); 42 })
```

In the above example, even though the lambda function `e ⇒ ...` has effect `Q`, `Q` will not be reflected in the whole term's effect, which is `io`. Leaking `e` would extrude its type to `Exc[Str, ⊤]`, which would make it impossible to handle, making the program ill-typed.

5.3 Region-Based and Stack-Based Memory Management

It is well known that type-and-effect systems with static regions can be used to implement region-based memory management. Here, we review how region-based as well as *stack*-based memory management can be encoded in InvalML.

5.3.1 Region-Based Memory Management. The regions presented in this paper can readily be used to implement type-safe region-based memory management [84]. Notice that in our operational semantics, when a region that was allocated with **region** `r` goes out of scope, the region and all the references allocated in it are *immediately removed* from the heap ψ, even when there are leftover references to these objects still reachable from live values (meaning that a garbage collector would keep these objects alive, increasing memory usage). Our type system makes sure that these reachable but dead objects can never be accessed (Lemma D.45). In practice, on the type level, any reference of type $\mathrm{Ref}[\tau,\ \mathbf{out}\,\alpha]$ that outlive its region of type $\mathrm{Region}[\mathbf{out}\,\alpha]$ is *extruded* to a "useless" type like $\mathrm{Ref}[\tau,\ \mathbf{out}\,\top]$, ensuring memory safety. Indeed, using that reference would produce effect $\top$, but we require that programs never have the $\top$ effect by forcing the main function to be pure (i.e., to have effect $\bot$). Based on this insight, we plan to eventually implement region-based memory management as part of a future version of MLscript.

5.3.2 Stack-Based Memory Management. The difference between region-based and stack-based memory management is that the latter is strictly more restrictive: since there is a single stack instead of a multitude of independently growable regions, all stack allocations and deallocations must be performed in strict "last in, first out" order. The upshot is that stack-based memory management can often be implemented more easily and more efficiently. We can encode this more restrictive API using an approach analogous to invalidation-safe iterators (§2.5):

```
fun allocStack: ∀ E, Res. (∀ S, R. Stack[S, R] →{S ∨ R ∨ E} Res) →{E} Res
fun alloc: ∀ S, R, A. (Stack[S, R], A) →{R} StackRef[A, R]
fun read: ∀ R, A. (StackRef[A, R]) →{R} A
fun write: ∀ R, A. (StackRef[A, R], A) →{R} ()
fun push: ∀ Res, R, E, S {E ≤ ¬S}.
  (Stack[S, R], ∀ U. Stack[U, R] →{U ∨ R ∨ E} Res) →{S ∨ R ∨ E} Res
```

To create a new stack, one uses `allocStack`, and to create a new stack frame, one uses `push`. While a stack frame is pushed (i.e., while the continuation argument to `push` is executing), no other stack frame can be pushed onto the same original stack, due to the '`E ≤ ¬R`' bound.

5.4 Scope-Safe Metaprogramming

Last but not least, we show how to add scope-safe analytic quasiquotes to $\lambda^{!\bot}$. Quasiquotes allow users to construct and inspect abstract syntax trees using quoted code templates [30, 69]. Scope-safe quasiquotes must prevent open code fragments (i.e., code fragments with free variables) from being executed. $\lambda^{!\bot}$ achieves scope-safe metaprogramming essentially the same as how we guarantee memory safety in §3. Each code fragment type contains a type argument for the contextual requirement. If the contextual requirement is $\bot$, the quoted term contains no free variable. Each next-stage variable is allocated, assigned a fresh skolem as the contextual requirement, and passed to a higher-ranked function that uses this variable to assemble a quoted function body. We use unions to bring different free variables' contextual requirements in a code fragment together. An escaped next-stage variable will have a contextual requirement $\top$ due to the extrusion.

Table 2. Comparison Table. Each row indicates whether a feature is supported (✓) or not (✗). A feature is partially supported (○) if it is restricted to some special cases. “CL” stands for “Capability Language”, “CT” stands for “Capturing Types”, “RT” stands for “Reachability Types”, and “MC” stands for “Mode Calculus”.

Feature/Approach	InvalML	ReML [24]	Flix [50]	CL [94]	Mezzo [75]	Effekt [11]	CT [102]	Rust [3]	RT [4]	MC [47]
Subtyping	✓	○	✗	○	✓	○	✓	✓	✓	○
Type Inference	Principal	Principal	Principal	✗	○	Local	Local	Local	Local	Principal
Permanent Invalidation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Temporary Invalidation	✓	○	○	✓	✓	○	○	✓	✓	✓
Region System	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓
Disjointness Reasoning	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No Separation-Default	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓
Use-Mention Distinction	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗
Reusability	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Uniqueness	✗	✗	✗	✓	✓	✗	✗	✓	✓	✓

This can be done without modifying the type system at all, by simply adding a few primitives, thanks to the use of abstract type variables and higher-ranked polymorphism, inspired by the `runST` approach in Haskell [44]. Both abstract type variables and higher-ranked polymorphism are simply type system features and less related to the concrete features like memory management or metaprogramming. Boolean algebraic subtyping with type inference further provides the ability to encode and infer contextual requirements by using union and complementary types. We leave more discussions for App. A.5 due to the lack of space.

6 Related Work

This section reviews the related work. Table 2 presents a comparison between InvalML and relevant systems. The row “Subtyping” indicates if implicit subtyping is supported. Partially supported subtyping can only be applied to effect or capability sets. The next row specifies if type inference is supported. “Permanent Inval” and "Temporary Inval" stand for “Permanent Invalidation” and "Temporary Invalidation" respectively, and are described in §2. Notably, some systems can only support some forms of temporary invalidation (e.g., freeze a given region temporarily), but cannot support a more general form like iterator invalidation safety that rely on higher-rank interfaces. The row “Region System” specifies if the system supports regions or similar concepts like bags or lifetimes. “Disjointness Reasoning” indicates the ability to reason about interference (or separation) between two elements or operations [81]. “No Sep-Default” means a system does not require two elements to be separated from each other by default. A `callSeq` example that will be rejected by a “Separation-Default” system defaultly is given in Linearity and Uniqueness part. “Use-mention distinction” denotes the ability to clarify whether a variable is actually used or merely mentioned, instead of conflating them [33]. “Reusability” means the ability to reuse the same system to define interfaces that check other sorts of language features, beyond memory safety and resource management, such as scope-safe metaprogramming. Finally, the row “Uniqueness” denotes the ability to track the uniqueness of mutable references. It is a critical property to support powerful features like strong updates that allow one to change the type of a unique object [29, 75]. We leave detailed discussions on Table 2 in App. A.1. Due to a lack of space, we cannot list all systems or features in Table 2. More systems and features are discussed below.

Effect Systems. The idea of effect checking dates back to Gifford and Lucassen [32]’s work. Lucassen and Gifford [48] later improved the idea of effect systems by using *effect masking*. Side effects are delimited by regions. When exiting a region, one can erase the effects indicated by this region if the region descriptor is not leaked. The idea of effect masking is also adopted by many consequent systems (e.g., ReML [24]). InvalML supports effect masking via subtype extrusion, marking leaked region descriptors and mutable references invalid instead of rejecting the code directly. DPJ [7] makes use of effect disjointness for parallel programming, but only supports monomorphic methods.

Another approach, Boolean unification [50–52], also proposes effect disjointness reasoning and is implemented in Flix language. Flix does not support subtyping but guarantees that it is free from the poisoning problem [86, 98]. Nevertheless, it is still acknowledged that the effect cast is required in some cases [9]. Lacking subtyping also makes type annotations hard to understand. Finally, Algebraic effects [16, 45] have gained considerable popularity recently. They provide a flexible and powerful mechanism to encode effect handing and control-flow abstraction [45]. We do not include Algebraic effects in this paper and consider algebraic effects a future work.

ReML. Recent work ReML [24] equips the traditional region- and effect systems with explicit effect constraints for effect inclusion and disjunction. However, such effect constraints are not sufficient to encode temporary invalidation safety like borrowing semantics or iterator invalidation safety, which can be provided by InvalML thanks to the higher-ranked polymorphism. ReML also requires access to implementations since it sticks with SML type signatures. Furthermore, ReML's implementation struggles with cases where we need to reason about the disjointness of yet-unknown outer regions. This problem is addressed by the outer variable ω in $\lambda^{!\perp}$. Local regions inside function bodies are separated from all yet-unknown outer regions abstracted by ω.

Capabilities. Capability Calculus [17] (and consequent Capability Language [94]) specifies the capabilities of the effects that can occur in expressions. Compared with traditional effect systems, Capability Calculus provides more flexible controls like explicit deallocation. A capability-based approach can also easily encode temporary invalidation like borrow semantics [10]. It is widely adopted by low-level and system languages like Cyclone [35], Vault [19], Sing# [26]. However, relying on the linearity of capabilities still puts restrictions on programmers:

```
fun foo1(r1, r2) = freeze(r1, () ⇒ /* code that does not use r2 */); bar(r2)
fun foo2(r1, r2) = freeze(r1, () ⇒ bar(r2))
```

To freeze `r1`, Capability Calculus requires the corresponding capabilities to be *unique*, which prevents `r1` and `r2` from aliasing. In above example, an application `foo1(r, r)` will not introduce any problem, but it is rejected by Capability Calculus because `foo1`'s type and `foo2`'s type cannot be distinguished from each other by Capability Calculus. Recently, the idea of effects-as-capabilities [11, 12, 55] has gained a lot of popularity. It allows users to omit some effect annotations to provide a more lightweight surface language. To ensure scope safety statically, they adopt *second-class values*, formalized by Osvald et al. [65] and used in various languages, such as C# with its references. Second-class values can be passed to callees but cannot be returned and cannot be stored or captured as part of returned values (e.g., closures). Brachthäuser et al. [11] later relaxes this restriction by introducing boxes to lift second-class values to first-class ones. However, this is still a very strong restriction, as it curtails expressiveness and imposes a language-wide segregation between first-class and second-class values, which can be cumbersome and lead to code duplication. Capturing types [8, 9, 102] follow a similar approach to provide a lightweight effect polymorphism with disjointness reasoning. Capturing types track not only the types of the given terms, but also parameters or local variables that are captured by the given terms. Once local variables are leaked, corresponding capabilities will be widened to a top-level capability `cap`, which is similar to our extrusion process. Xu et al. [102] further distinguish reading capabilities from writing ones, allowing two reading operations to be executed in parallel, which is not supported in our system yet and can be considered as future work. Nevertheless, Capturing types are still not as fine-grained as $\lambda^{!\perp}$. Consider the following program:

```
let ys = in xs.map(x ⇒ throw(e, "oops"); x + 1) // xs: LazyList[Int, ⊥]
```

[9]https://doc.flix.dev/casts.html

In Capturing Types, the type of `ys` has the capability `e` and cannot be treated as a pure term, even though it is a lazy list and will not be executed immediately. If a function captures `ys` without actually using it (e.g., a function simply returns `ys`), the function will be treated as impure by Capturing Types due to the lack of use-mention distinction [33]. An effect system can accurately tell whether an effectful term is used or just mentioned and compute precise effects. On the other hand, capability systems are more compatible with object-oriented features [33], which is not our focus in this paper. We consider such object-oriented extensions as future work.

Linearity and Uniqueness. Linear values must be used exactly once and cannot be duplicated or destroyed [92]. However, linearity puts heavy restrictions on aliasing. Approaches like adoption [27] are proposed to allow temporary aliases. Rust [37, 54, 100] is one of the most popular languages relying on ownership, linearity, and borrowing semantics to achieve memory safety and data-race-free concurrency. Immutable (or shared) borrows can be duplicated, while mutable (or exclusive) ones must be unique. Once a variable is borrowed, it must be frozen until the lifetime of the borrow ends. We have not distinguished between shared borrows and exclusive ones and leave it as future work. Such linearity (or uniqueness) properties are also required by some capability-based approaches, as we mentioned in the previous subsection. However, such substructural-type systems put a lot of restrictions on programmers. Assume some mutable reference `r`, the following program is rejected by Rust due to the non-unique exclusive borrow, even though it is actually memory-safe since `callSeq` invokes `f` and `g` one by one, instead of calling them in parallel:

```
fun callSeq(f, g) = f(0); g(1) in callSeq(x ⇒ r := x, y ⇒ r := y)
```

Pottier and Protzenko [75] presented Mezzo with permissions, inspired by Alias Types [82, 95], to enable features like gradual initialization and strong updates. But Mezzo only supports a limited form of type inference and requires existential types to encode affine functions that capture non-duplicable permissions. Radanne et al. [77] later proposed a principal type inference algorithm for shared and exclusive borrowing based on $\mathsf{HM}(X)$ [63], making it an extension of ML-like languages. However, $\mathsf{HM}(X)$ is unwieldy, compared to Parreaux [66]'s approach we follow, where all one has to do is duplicate constraint sets. Reachability types [4, 99] track reachability sets of terms to achieve better aliasing control and separation across higher-order functions. The `callSeq` function can be implemented in their system, but it still requires explicit type annotations to permit the aliases. Lorenzen et al. [47] proposed a mode-based approach for manual memory management in OCaml based on uniqueness and locality. Once a pair is splited, one can reuse the space credit for a new pair. We also consider similar memory management features as future work.

Scope Safety for Metaprogramming. One of the contributions of InvalML is the extensibility for other language features like scope-safe metaprogramming. Environment classifiers approaches [38, 43, 83] are also known to have abilities to encode both scope-safe metaprogramming and effect systems. Compared with our approach, their system is less fine-grained and difficult to support analytic-style quasiquotes. It also needs more annotations for classifier upcasting due to the lack of implicit subtyping. Another area of scope safety for metaprogramming is contextual types [39, 56–58, 69, 104], inspired by contextual modal type theory [59]. It is also adapted for Algebraic effects [56] recently. InvalML mainly follows Gao and Parreaux [30]'s approach to adopt higher-ranked polymorphism and constraint-based type inference, which requires fewer type annotations.

7 Conclusion

In this paper, we described a powerful type-and-effect system to express invalidation and scope safety through disjointness constraints in a lightweight manner. We have demonstrated the versatility of this system by presenting various applications such as memory management, exception handling, data-race-free concurrency, metaprogramming, and modular Rust-style borrowing.

Data-Availability Statement

The implementation of our system, based on the existing programming language MLscript, was submitted for Artifact Evaluation [31]. The extension consists of only 1570 lines of code on top of the existing MLscript implementation, which already handles parsing and code generation. A web demonstration is available at https://hkust-taco.github.io/invalml-web-demo/.

Acknowledgments

We thank the anonymous reviewers, Andong Fan, Chun Yin Chau, and Martin Elsman for their helpful comments. We thank Luyu Cheng, Ruqing Yang, Yijia Chen, Ching Hang Mak, and Heung Tung Au for their help with the artifact.

References

[1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 174–185. doi:10.1145/207110.207137 ↪ page 2

[2] 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. doi:10.1145/1190216.1190235 ↪ page 14

[3] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 156–161. doi:10.1145/3102980.3103006 ↪ pages 23 and 32

[4] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 139 (oct 2021), 32 pages. doi:10.1145/3485516 ↪ pages 23, 25, and 32

[5] H P Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*. Oxford University Press. arXiv:https://academic.oup.com/book/0/chapter/421961562/chapter-pdf/52352320/isbn-9780198537618-book-part-2.pdf doi:10.1093/oso/9780198537618.003.0002 ↪ page 11

[6] Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (Oct. 2023), 29 pages. doi:10.1145/3622812 ↪ page 7

[7] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 97–116. doi:10.1145/1640089.1640097 ↪ pages 2 and 23

[8] Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. arXiv:2105.11896 [cs.PL] doi:10.48550/arXiv.2105.11896 ↪ page 24

[9] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4, Article 21 (nov 2023), 52 pages. doi:10.1145/3618003 ↪ page 24

[10] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27. doi:10.1007/3-540-45337-7_2 ↪ pages 2 and 24

[11] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 76 (apr 2022), 30 pages. doi:10.1145/3527320 ↪ pages 3, 23, 24, 32, and 34

[12] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (nov 2020), 30 pages. doi:10.1145/3428194 ↪ page 24

[13] Chun Yin Chau. 2023. *Subtyping in a Boolean algebra of structural types*. Master's thesis. Hong Kong. doi:10.14711/thesis-991013223047203412 ↪ page 14

[14] Chun Yin Chau and Lionel Parreaux. 2024. *Boolean-Algebraic Subtyping: Intersections, Unions, Negations, and Principal Type Inference.* https://lptk.github.io/files/boolean-algebraic-subtyping.pdf ↪ pages 14 and 41

[15] Dave Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 292–310. doi:10.1145/582419.582447 ↪ page 2

[16] LUKAS CONVENT, SAM LINDLEY, CONOR MCBRIDE, and CRAIG MCLAUGHLIN. 2020. Doo bee doo bee doo. *Journal of Functional Programming* 30 (2020), e9. doi:10.1017/S0956796820000039 ↪ page 24

[17] Karl Crary, David Walker, and Greg Morrisett. 1999. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. doi:10.1145/292540.292564 ↪ pages 2, 3, 7, and 24

[18] Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2023. Greedy Implicit Bounded Quantification. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 295 (oct 2023), 29 pages. doi:10.1145/3622871 ↪ page 12

[19] Robert DeLine and Manuel Fähndrich. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 59–69. doi:10.1145/378795.378811 ↪ pages 2, 4, and 24

[20] Stephen Dolan. 2017. *Algebraic subtyping.* Ph. D. Dissertation. ↪ pages 3, 5, and 6

[21] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 60–72. doi:10.1145/3093333.3009882 ↪ pages 5 and 6

[22] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. doi:10.1145/3450952 ↪ page 3

[23] Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. doi:10.1145/2500365.2500582 ↪ page 12

[24] Martin Elsman. 2024. Explicit Effects and Effect Constraints in ReML. *Proc. ACM Program. Lang.* 8, POPL, Article 79 (Jan. 2024), 25 pages. doi:10.1145/3632921 ↪ pages 2, 3, 6, 7, 23, 24, 32, and 34

[25] Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. *SIGPLAN Not.* 41, 1 (jan 2006), 270–282. doi:10.1145/1111320.1111062 ↪ page 15

[26] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (Leuven, Belgium) *(EuroSys '06)*. Association for Computing Machinery, New York, NY, USA, 177–190. doi:10.1145/1217935.1217953 ↪ page 24

[27] Manuel Fahndrich and Robert DeLine. 2002. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/512529.512532 ↪ pages 2 and 25

[28] Andong Fan, Han Xu, and Ningning Xie. 2025. Practical Type Inference with Levels. *Proc. ACM Program. Lang.* 9, PLDI, Article 235 (June 2025), 24 pages. doi:10.1145/3729338 ↪ pages 3 and 11

[29] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/512529.512531 ↪ pages 2, 4, and 23

[30] Cunyuan Gao and Lionel Parreaux. 2024. Seamless Scope-Safe Metaprogramming through Polymorphic Subtype Inference (Short Paper). In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Pasadena, CA, USA) *(GPCE '24)*. Association for Computing Machinery, New York, NY, USA, 121–127. doi:10.1145/3689484.3690733 ↪ pages 22 and 25

[31] Cunyuan Gao and Lionel Parreaux. 2025. *Artifact for "A Lightweight Type-and-Effect System for Invalidation Safety: Tracking Permanent and Temporary Invalidation With Constraint-Based Subtype Inference"*. doi:10.5281/zenodo.16918061 ↪ page 26

[32] David K Gifford and John M Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 28–38. ↪ pages 2, 3, and 23

[33] Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants. 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, 10:1–10:25. doi:10.4230/LIPIcs.ECOOP.2020.10 ↪ pages 7, 23, and 25

[34] Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 4 (April 2021), 79 pages. doi:10.1145/3450272 ↪ page 4

[35] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. doi:10.1145/512529.512563 ↪ page 24

[36] Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 253–289. doi:10.1145/169701.169692 ↪ page 3

[37] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 251 (Aug. 2024), 29 pages. doi:10.1145/3674640 ↪ pages 2 and 25

[38] Kanaru Isoda, Ayato Yokoyama, and Yukiyoshi Kameyama. 2024. Type-Safe Code Generation with Algebraic Effects and Handlers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Pasadena, CA, USA) *(GPCE '24)*. Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/3689484.3690731 ↪ page 25

[39] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–27. doi:10.1145/3498700 ↪ page 25

[40] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203. doi:10.1007/3-540-15975-4_37 ↪ page 3

[41] A. J. Kfoury and J. B. Wells. 1994. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, USA) *(LFP '94)*. Association for Computing Machinery, New York, NY, USA, 196–207. doi:10.1145/182409.182456 ↪ page 3

[42] Oleg Kiselyov. 2013. Efficient and Insightful Generalization. https://okmij.org/ftp/ML/generalization.html ↪ pages 3 and 11

[43] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Vol. 10017. Springer International Publishing, Cham, 271–291. doi:10.1007/978-3-319-47958-3_15 Series Title: Lecture Notes in Computer Science. ↪ pages 25, 35, and 36

[44] John Launchbury and Simon L Peyton Jones. 1995. State in haskell. *Lisp and symbolic computation* 8 (1995), 293–341. doi:10.1007/BF01018827 ↪ pages 23 and 34

[45] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. doi:10.1145/3009837.3009872 ↪ page 24

[46] Xavier Leroy. 1992. *Polymorphic typing of an algorithmic language.* Ph. D. Dissertation. INRIA. ↪ page 12

[47] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, Article 253 (Aug. 2024), 30 pages. doi:10.1145/3674642 ↪ pages 23, 25, 32, and 33

[48] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. 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, 47–57. doi:10.1145/73560.73564 ↪ pages 2, 3, and 23

[49] Matthew Lutze and Magnus Madsen. 2024. Associated Effects: Flexible Abstractions for Effectful Programming. *Proc. ACM Program. Lang.* 8, PLDI, Article 163 (June 2024), 23 pages. doi:10.1145/3656393 ↪ page 11

[50] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *Proc. ACM Program. Lang.* 7, ICFP, Article 204 (Aug. 2023), 28 pages. doi:10.1145/3607846 ↪ pages 23, 24, 32, 34, and 35

[51] Magnus Madsen and Jaco van de Pol. 2020. Polymorphic types and effects with Boolean unification. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 154 (Nov. 2020), 29 pages. doi:10.1145/3428222 ↪

[52] Magnus Madsen and Jaco van de Pol. 2023. Programming with Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:27. doi:10.4230/LIPIcs.ECOOP.2023.18 ↪ pages 3, 6, 24, and 34

[53] Daniel Marino and Todd Millstein. 2009. A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (Savannah, GA, USA) *(TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 39–50. doi:10.1145/1481861.1481868 ↪ page 34

[54] Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing

Machinery, New York, NY, USA, 103–104. doi:10.1145/2663171.2663188 ↪ pages 2 and 25

[55] Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 255 (oct 2023), 30 pages. doi:10.1145/3622831 ↪ page 24

[56] Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. 2023. Contextual Modal Type Theory with Polymorphic Contexts. In *Programming Languages and Systems*, Thomas Wies (Ed.). Vol. 13990. Springer Nature Switzerland, Cham, 281–308. doi:10.1007/978-3-031-30044-8_11 Series Title: Lecture Notes in Computer Science. ↪ page 25

[57] Aleksandar Nanevski. 2002. Meta-programming with names and necessity. *ACM SIGPLAN Notices* 37, 9 (Sept. 2002), 206–217. doi:10.1145/583852.581498 ↪

[58] Aleksandar Nanevski and Frank Pfenning. 2005. Staged computation with names and necessity. *Journal of Functional Programming* 15, 6 (Nov. 2005), 893–939. doi:10.1017/S095679680500568X ↪ page 25

[59] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. doi:10.1145/1352582.1352591 ↪ page 25

[60] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Type and Effect Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 283–363. doi:10.1007/978-3-662-03811-6_5 ↪ page 34

[61] Martin Odersky. 1992. Observers for linear types. In *ESOP '92*, Bernd Krieg-Brückner (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 390–407. doi:10.1007/3-540-55253-7_23 ↪ page 2

[62] Martin Odersky and Konstantin Läufer. 1996. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. doi:10.1145/237721.237729 ↪ page 12

[63] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and practice of object systems* 5, 1 (1999), 35–55. doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4 ↪ page 25

[64] Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman. 2020. The Duality of Subtyping. (2020), 29 pages, 728719 bytes. doi:10.4230/LIPICS.ECOOP.2020.29 ↪ page 10

[65] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 234–251. doi:10.1145/2983990.2984009 ↪ page 24

[66] 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. doi:10.1145/3409006 ↪ pages 3, 6, 11, 14, 25, and 39

[67] Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 48 (jan 2024), 33 pages. doi:10.1145/3632890 ↪ pages 3 and 5

[68] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: principal type inference in a Boolean algebra of structural types. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 449–478. doi:10.1145/3563304 ↪ pages 3, 5, 6, 9, 10, 14, 15, 17, 18, 44, 45, and 47

[69] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–33. doi:10.1145/3158101 ↪ pages 22, 25, and 36

[70] David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 335–354. doi:10.1007/978-3-642-35873-9_21 ↪ page 17

[71] Tomas Petricek and Don Syme. 2012. Syntax Matters: Writing abstract computations in F#. *Pre-proceedings of TFP (Trends in Functional Programming), St. Andrews, Scotland* (2012). ↪ page 34

[72] SIMON PEYTON JONES, DIMITRIOS VYTINIOTIS, STEPHANIE WEIRICH, and MARK SHIELDS. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82. doi:10.1017/S0956796806006034 ↪ pages 3, 5, 11, and 12

[73] François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://inria.hal.science/inria-00073205 ↪ page 3

[74] François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://inria.hal.science/inria-00073205 ↪ page 6

[75] François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 173–184. doi:10.1145/2500365.2500598 ↪ pages 2, 23, 25, and 32

[76] Francois Pottier and Didier Rémy. 2005. *The Essence of ML Type Inference.* 389–489. ↪ pages 3 and 11

[77] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2020. Kindly bent to free us. *Proc. ACM Program. Lang.* 4, ICFP, Article 103 (aug 2020), 29 pages. doi:10.1145/3408985 ↪ pages 2, 15, 25, and 43

[78] D. Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 77–88. doi:10.1145/75277.75284 ↪ page 6

[79] Didier Rémy. 1992. *Extension of ML type system with a sorted equation theory on types.* Research Report RR-1766. INRIA. https://hal.inria.fr/inria-00077006 Projet FORMEL. ↪ pages 3 and 11

[80] Didier Rémy. 1994. *Type Inference for Records in Natural Extension of ML.* MIT Press, Cambridge, MA, USA, 67–95. ↪ page 6

[81] John C. Reynolds. 1978. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 39–46. doi:10.1145/512760.512766 ↪ page 23

[82] Frederick Smith, David Walker, and Greg Morrisett. 2000. Alias Types. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 366–381. doi:10.1007/3-540-46425-5_24 ↪ pages 7 and 25

[83] Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. *ACM SIGPLAN Notices* 38, 1 (Jan. 2003), 26–37. doi:10.1145/640128.604134 ↪ page 25

[84] J.P. Talpin and P. Jouvelot. 1994. The Type and Effect Discipline. *Information and Computation* 111, 2 (1994), 245–296. doi:10.1006/inco.1994.1046 ↪ pages 22 and 34

[85] Jean-Pierre Talpin and Pierre Jouvelot. 1992. Polymorphic type, region and effect inference. *Journal of Functional Programming* 2, 3 (1992), 245–271. doi:10.1017/S0956796800000393 ↪ pages 2 and 3

[86] Yan Mei Tang and Pierre Jouvelot. 1995. Effect systems with subtyping. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (La Jolla, California, USA) *(PEPM '95)*. Association for Computing Machinery, New York, NY, USA, 45–53. doi:10.1145/215465.215552 ↪ pages 3 and 24

[87] Ross Tate. 2013. Mixed-Site Variance. In *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages* (Indianapolis, IN, USA). http://www.cs.cornell.edu/~ross/publications/mixedsite/ ↪ page 10

[88] Mads Tofte and Lars Birkedal. 1998. A region inference algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 724–767. doi:10.1145/291891.291894 ↪ page 6

[89] Mads Tofte and Lars Birkedal. 2000. *Unification and polymorphism in region inference.* MIT Press, Cambridge, MA, USA, 389–423. ↪ page 34

[90] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. 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, 188–201. doi:10.1145/174675.177855 ↪ page 2

[91] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. doi:10.1006/inco.1996.2613 ↪ page 2

[92] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5. ↪ pages 2 and 25

[93] Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1–14. ↪ page 34

[94] David Walker, Karl Crary, and Greg Morrisett. 2000. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (jul 2000), 701–771. doi:10.1145/363911.363923 ↪ pages 2, 23, 24, and 32

[95] David Walker and Greg Morrisett. 2001. Alias Types for Recursive Data Structures. In *Types in Compilation*, Robert Harper (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 177–206. doi:10.1007/3-540-45332-6_7 ↪ pages 7 and 25

[96] Mitchell Wand. 1991. Type inference for record concatenation and multiple inheritance. *Inf. Comput.* 93, 1 (jul 1991), 1–15. doi:10.1016/0890-5401(91)90050-C ↪ page 6

[97] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *Proceedings of the Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSYS '12)*. Association for Computing Machinery, New York, NY, USA, Article 9, 7 pages. doi:10.1145/2349896.2349905 ↪ page 2

[98] Keith Wansbrough and Simon Peyton Jones. 1999. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/292540.292545 ↪ pages 6, 24, and 34

[99] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 14 (jan 2024), 32 pages. doi:10.1145/3632856 ↪ page 25

[100] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. arXiv:1903.00982 [cs.PL] doi:10.48550/arXiv.1903.00982 ↪ pages 2 and 25

[101] Andrew K. Wright. 1995. Simple imperative polymorphism. *LISP and Symbolic Computation* 8, 4 (Dec. 1995), 343–355. doi:10.1007/BF01018828 ↪ page 12

[102] Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2024. Degrees of Separation: A Flexible Type System for Safe Concurrency. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 136 (April 2024), 27 pages. doi:10.1145/3649853 ↪ pages 7, 23, 24, and 32

[103] Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. 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, 2:1–2:28. doi:10.4230/LIPIcs.ECOOP.2022.2 ↪ page 12

[104] Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual modal types for algebraic effects and handlers. *Proc. ACM Program. Lang.* 5, ICFP, Article 75 (Aug. 2021), 29 pages. doi:10.1145/3473580 ↪ page 25

A Additional Discussions & Examples

In this appendix, we provide additional discussions and examples.

A.1 Discussions on the Comparison Table

In this section, we justify some non-trivial cells in Table 2.

Type Inference.

- CL [94] focuses on type checking instead of type inference. Explicit type annotations also make type checking for first-class polymorphism straightforward.
- Mezzo's [75] prototype only supports a limited form of type inference, which is not discussed in their paper.

Temporary Invalidation.

- ReML [24], Flix [50], Effekt [11], and CT [102] can encode the `freeze` example discussed in §6 by making use of effect disjointness. However, it is not clear how these systems can support the iteration example in §2 due to the lack of higher-ranked polymorphism.
- It is possible for Mezzo [75] to encode both examples by suspending the given unique permission. For the iteration example, one still needs to allocate a new non-duplicable permission and require the argument function to return it.

Disjointness Reasoning.

- Effekt [11] has not mentioned disjointness (or separation, interference). But we believe it can be achieved by computing the disjointness of capability sets on boxed types.

No Separation-Default.

- Mezzo [75] requires exclusive permissions for writing, and functions in Mezzo can only capture duplicable permissions by default.
- RT [4] must annotate the parameters' reachability sets to allow aliases.

Use-Mention Distinction.

- CL [94] supports the use-mention distinction since only mutable operations require capabilities. By contrast, Mezzo [75] mixes types and permissions. Therefore, accessing a variable requires a corresponding capability in the context.
- Rust [3] and MC [47] do not have the use-mention distinction since they do not track effects at all.

Reusability.

- No system in Table 2 mentioned reusability for other language features. Even though Effekt [11], CT [102], and MC [47] adopt modal types, which are also widely used for scope-safe metaprogramming, it is not clear how their systems can encode metaprogramming interfaces *without* changing the core type systems. On the one hand, modes in these systems are handled separately from types. On the other hand, these modes are more related to effects or uniqueness and are not general enough.

A.2 Mutable State Encapsulation

Consider the function below, written in some imaginary ML dialect, which implements the `mapi` function using `List.map`, where `mapi` is the same as `map` except that it also provides to its argument function the current element index along with the element itself:

```
fun mapi[A, B](xs: List[A])(f: (Int, A) → B): List[B] =
  let index = ref -1
  List.map(xs)(x ⇒ index := !index + 1; f(!index, x))
```

This function allocates a mutable reference `index` through the **`ref`** constructor and updates it through the `!` (read) and `:=` (write) operators as the underlying list mapping process progresses. While `mapi` uses mutable state internally, this state is encapsulated within the function's scope and is not observable from the outside. This is referred to as *referential transparency*: `mapi` always returns the same outputs given the same inputs and does not have any side effects. As a consequence, for example, if a program uses an expression like `mapi(ls)(foo)` several times, it can always let-bind that expression and thereby avoid recomputing it each time, with the guarantee provided by referential transparency that the overall program result will not change.

Now, consider the following slight variations on `mapi`:

```
fun mapi2[A, B](xs: Seq[A])(f: (Int, A) → B): Seq[B] =
  let index = ref -1
  Seq.map(xs)(x ⇒ index := !index + 1; f(!index, x))

fun mapi3[A, B](xs: List[A]): ((Int, A) → B) → List[B] =
  let index = ref -1
  f ⇒ List.map(xs)(x ⇒ index := !index + 1; f(!index, x))
```

On one hand, `map2` uses a *sequence* data structure instead of a list. Sequences are lazy and only compute new elements on demand. Like in OCaml, we assume that sequences are not memorized by default, such that `map` will be recomputed every time the returned sequence is queried.[10] This `mapi2` function is no longer referentially transparent, as demonstrated by the two programs below returning different results:

```
// Program P1:
let s2 = mapi2(s2)((i, x) ⇒ x * i)
let size = Seq.count(s2)
Seq.sum(s2) / size
```

```
// Program P2:
let size =
  Seq.count(mapi2(s2)((i, x) ⇒ x * i))
Seq.sum(mapi2(s2)((i, x) ⇒ x * i)) / size
```

Indeed, for the sequence `1, 2, 3`, program `P1` returns `(1 * 3) + (2 * 4) + (3 * 5) / 3 = 26/3`, while `P2` returns `(1 * 0) + (2 * 1) + (3 * 2) / 3 = 8/3`. What happened in `P1` is that the counter mutable state captured by the returned sequence retained its old value from the first call to `mapi2` at the time `Seq.sum` was called, therefore producing incorrect indices.

On the other hand, `mapi3` instantiates its mutable state *before* returning a closure that performs the mapping computation. Although it manipulates eagerly-computed lists and has the same type as `mapi`, it still leaks its mutable state through the returned closure, and has similar referential transparency problems as `mapi2` when partially applied.

In both cases, the mutable state has leaked through the returned value, a violation of scope safety that was not caught because languages like OCaml do not guarantee scope safety statically (though some recent work has been pushing in that direction [47]). *Mutable state* leakage does not lead to *unsoundness* of the type system — it merely makes programs harder to reason about. But in many (if not most) other domains, violating scope safety routinely leads to unsoundness and crashes at runtime. For instance, it can lead to exceptions being thrown outside of their handlers, pointer being dereferenced after their memory has been deallocated, metaprogram fragments being run in the wrong environments, runtime type mismatches, etc.

Many approaches have been proposed over the years for ensuring scope safety statically in various domains. One way is to rely on higher-rank polymorphic types, as these types naturally

[10]Like in OCaml, we assume that to memorize a call to `Seq.map(xs)(f)` so that the function `f` is only applied at most once per element, one needs to write `Seq.memorize(Seq.map(xs)(f))`.

come with a notion of scope — i.e., quantified type variables are only available within the scope of the polymorphic type and should not leak to the outside. For instance, consider Haskell's well-known "`runST` trick" [44], which uses higher-rank polymorphism to prevent local mutable state from being observable. In Haskell, the `runST` function has the following type:

```
runST: ∀ A . (∀ S . ST[S, A]) → A
```

where `ST` is the state monad and `ST[S, A]` denotes a (delayed) computation returning a value of type `A` and restricted to some scope marked as `S`. This approach requires the entire function to be converted to monadic style [93]. While this programming style does the job, rejecting programs like `mapi2` and `mapi3` with the compiler complaining that the locally-quantified type `S` escapes its scope,[11] it is quite inconvenient. Beyond the syntactic clutter, which can largely be alleviated through **do** or **let**! notations [71], it requires all combinators like `map` to have a monadic version like `mapM`, and intermixing various effects requires heavyweight scaffolding such as monad transformers. Various approaches try to avoid monad transformers by using a single monad along with an *effect system* to parameterize the monad type. By contrast, *direct-style* approaches use type-and-effect systems [53, 60, 84] to let users mix various effects in their non-monadic (i.e., direct-style) programs while keeping track of them in the type system. In such systems, function types are typically augmented with an effect annotation φ, as in $\mathsf{Int} \xrightarrow{\varphi} \mathsf{Int}$, the type of integer-to-integer functions with effect φ. We focus on that approach in this paper.

A.3 Subtyping, Polymorphism, and Effect Poisoning

Unlike approaches based on *unification*, and thanks to *subtyping*, our type system does not suffer from the *poisoning problem* [98] and does not require extraneous polymorphism or manual coercions.

Madsen and van de Pol [52] report one problem of systems that suffer from the poisoning problem is that the sub-expressions are required to have the same effects. Consider the following contrived example:

```
fun foo(xs, f) = foreach of map(xs, f), x ⇒ println(x)
```

Given $\mathsf{println} : \mathsf{Str} \xrightarrow{\mathsf{console}} (\,)$, Koka will infer effect `<console|_e>` for the argument function `f`, where `_e` is a wildcard effect for effect polymorphism. Notice that `f` is polluted by the `println` invocation inside the function body and has effect `console`. In Madsen and van de Pol [52]'s system, it will prevent the sub-expression `map(xs, f)` from being optimized when we pass a pure function `f` to `foo`. Furthermore, assume that Koka can support effect disjointness like Flix does [50]. It is impossible to pass `f` to another function that requires an argument function involving no `console` effect unless the type annotation for `f` is given. Brachthäuser et al. [11] also find that even though in such systems with row polymorphism which largely alleviate the problem one still gets surprising types (e.g., duplicable effect entries) in some situations. InvalML can correctly infer the effect-polymorphic type for `foo`, thanks to Boolean-algebraic subtyping:

```
fun foo(xs, f) = foreach of map(xs, f), x ⇒ println(x)
// foo: ∀ T, E. (List[T], T →{E} Str) →{E ∨ console} ()
```

Notice that only `foo` carries `console` effect, while `f`'s effect is only denoted by `E`.

ReML [24] relies on Tofte and Birkedal [89]'s region inference algorithm, adopting latent effect $\epsilon.\varphi$ that is similar to row-polymorphism, where φ is a known effect and ϵ is an effect variable that can be used to extend the effect φ. Thanks to the effect subsumption, ReML can avoid unexpected pollutions in the above example. However, its implementation still fails to propagate effect constraints in some cases:

[11] To encode the type of effectful sequences in this approach, we would need to use a monad transformer stack where `S` would be reflected in the sequence type.

```
fun foo(r, f) = fork of () ⇒ f(1), () ⇒ r.ref 2
```

In the above definition, ReML's implementation will complain that it cannot ensure that `r`'s effect will not appear in `f`'s effect variable:

```
The effect e26 contains the atomic effect put(`r_23),
which I cannot conclude does not appear in e28, which contains e22
```

Flix [50] addresses the poisoning problem by using Boolean unification. However, due to the lack of subtyping, it usually produces confusing results:

```
fun print: Str →{console} ()
fun foo: ∀ E. (Int →{E ∧ ¬console} ()) →{E} ()
fun foo(f) = f(42)
```

In this example, `foo` accepts an argument function that cannot print anything. Flix compiler fails to unify effect `E ∧ ¬ console` and effect `E`, even though `E ∧ ¬ console` is a smaller effect than `E`. For the same reason, the following program is also rejected by Flix:

```
fun doPotentialConsoleTwice: ∀ T. (() →{console} T) →{console} T
doPotentialConsoleTwice of () ⇒ print("foo")
doPotentialConsoleTwice of () ⇒ 42 // Flix cannot unify pure and console!
```

Besides the poisoning problem, unification-based approaches also require more annotations for polymorphism and coercions. For example, auxiliary type constructors and upcasting functions are highly demanded in <NJ> [43] to ensure the scope-safety in metaprograms and upcast classifiers to pass code fragments somewhere else. In InvalML, these are automatically done thanks to the subtyping and type propagations.

A.4 Modularity and Non-Lexicality

Because our approach is *type-based*, it is naturally modular, in that any parts of an implementation can be reasoned about independently and moved to an externally-defined helper function. This is *not* the case in Rust, whose borrowing system needs to inspect the *syntax* of functions in a way that cannot be completely abstracted in types.

One interesting consequence of this is that while deep modifications were required in order to accommodate so-called *non-lexical lifetimes* in Rust, the corresponding pattern naturally falls out of our type-and-effect system without any additional effort. Consider the following example:

```
region r; let a = mkArrayList(r)
iter of a, it ⇒ if next(it) is
  None then println("none")
  Some(v) then println(v); clear(a)
```

This code is ill-typed due to the effect of `clear(a)`, but it can be fixed by simply moving the clearing of the vector to a function that delays it. This now type checks because the effect is delayed to after the iteration is done:

```
region r; let a = mkArrayList(r)
let k = iter of a, it ⇒ if next(it) is
  None then println("none"); () ⇒ ()
  Some(v) then println(v); () ⇒ clear(a)
k()
```

A.5 Scope-Safe Metaprogramming

We first show how to add scope-safe analytic quasiquotes to $\lambda^{!\perp}$. As we shall see, this can be done without modifying the type system at all, by simply adding a few primitives.

$$\begin{array}{lll}
& \underline{\text{Syntax}} & \\
\textit{Term} & t ::= \ldots \mid \mathsf{run}\, t \mid \underline{\lambda}x.\, t \mid t \mathbin{\underline{@}} t \mid \mathbf{try}\, y = t \setminus t\, \mathbf{in}\, t\, \mathbf{else}\, t \mid \mathbf{subst}\, t\, t\, \mathbf{in}\, t \\
\textit{Pattern} & p ::= \underline{x} \mid \underline{\lambda}x.\, x \mid x \mathbin{\underline{@}} x \mid \mathsf{C}(\overline{x})
\end{array}$$

Fig. 10. Quasiquotes syntax.

A.5.1 Informal presentation. The syntax of code quotation is inspired by Lisp and environment classifiers [43]. We use the notation ` to quote each literal value (e.g., `` `42``, `` `"hello"``), keyword (e.g., **`if**, **`let**, `` `⇒``), and global identifier (e.g., , `` `*``, `` `+``, `` `id``) to be part of the residual program. Quoted applications are denoted as ``f`(a)``. For convenience, quotes can be distributed over subexpressions. For instance, `` `2 `+ `2`` can be written as `` `(2 + 2)`` for short.

Passing staged code to the `run` function moves it to the current stage for execution. To ensure both the type- and scope-safety of generated code statically, we adapt the typing approach of Squid [69]. Each staged code fragment has type Code[**out** τ, **out** σ], where τ indicates the type of the quoted term and σ states its context requirement. Only closed code (i.e., code without free variables) can be executed, which is done by making `run` take a parameter of type Code[**out** τ, **out** $\bot$].

```
run(`(x ⇒ x))  // ok! x ⇒ x is closed
x `⇒ run(x)    // error! x is open
y ⇒ `y  // error! y is unbound at this stage
```

We don't support unquotes explicitly, but current-stage code inside a quoted abstraction or application can execute normally, working as unquoting in other systems:

```
x `⇒ id(x) `* x  // id executes immediately
```

To type check this term, we first deduced that the type of `x` as used in the next stage is Int and so the type assigned to the `x` variable in the metaprogram is Code[**out** Int, **out** α], where type variable α is a fresh *locally-quantified* rigid type variable (or *skolem*), whose scope extends over the lambda's body. This notably ensures `x` cannot be executed within the function body. Non-quoted identifier `id` refers to a current-stage function that returns its argument unchanged at code generation time, so that the code above is equivalent to `` `(x ⇒ x * x)``. The Code type constructor is covariant in both of its type parameters. Moreover, a quoted identifier like `` `*`` has type $\forall\alpha.$ (Code[**out** Int, **out** α], Code[**out** Int, **out** α]) $\rightarrow$ Code[**out** Int, **out** α]. Thanks to subtyping, this means that given x : Code[**out** Int, **out** τ] and y : Code[**out** $0 \vee 1$, **out** σ], expression x `` `* `` y has type Code[**out** Int, **out** $\tau \vee \sigma$].

A.5.2 Formal development. We introduce a *staged code* type Code[**out** τ, **out** σ], where τ represents the type of the quoted term and σ represents its context requirements. The syntax of quasiquotes is given in Figure 10. No unquoting syntax is provided, as we prefer to use a more lightweight notation instead. We use $\underline{\lambda}x.\ t$ to create a staged lambda function and $t \mathbin{\underline{@}} t$ for staged application. To support code inspection, we use $\underline{x}$ to match quoted bindings. The remaining patterns correspond to the introduction syntax. Type Var[τ, σ] is introduced for the quoted variables and can be used in **`try`** and **`subst`**. The **try** $y = t_1 \setminus t_2$ **in** t_3 **else** t_4 construct checks if a given fragment t_1 contains no free variable x yielded by t_2. If x is not detected in t_1, then we re-assign t_1 to the variable y and execute the branch t_3. Otherwise, the fallback branch t_4 will be executed. **subst** t_1 t_2 **in** t_3 substitutes the variable yielded by t_1 appearing in the t_3 with the term t_2. Similarly to Ref and Region, we define Code and Var as built-in primitive classes:

Table 3. Typing of Quasiquotes

Desugaring	Builtin Signature
$\underline{\lambda} x.\ t \rightsquigarrow \mathrm{abs}\ (\lambda x.\ t)$	$\mathrm{abs} : \forall \alpha, \beta, \gamma_1, \delta.$ $(\forall \gamma_2. \mathrm{Var}[\alpha, \gamma_2] \xrightarrow{\delta} \mathrm{Code}[\mathbf{out}\ \beta, \mathbf{out}\ \gamma_1 \vee \gamma_2]) \xrightarrow{\delta} \mathrm{Code}[\mathbf{out}\ \alpha \to \beta, \mathbf{out}\ \gamma_1]$
$\mathrm{run}\ t$	$\mathrm{run} : \forall \alpha. \mathrm{Code}[\mathbf{out}\ \alpha, \mathbf{out}\ \bot] \to \alpha$
$\mathbf{try}\ y = t_1 \setminus t_2\ \mathbf{in}\ t_3\ \mathbf{else}\ t_4$ $\rightsquigarrow \mathrm{close}\ t_1\ t_2\ (\lambda y.\ t_3)\ (\lambda\_.\ t_4)$	$\mathrm{close} : \forall \alpha, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta.$ $\mathrm{Code}[\mathbf{out}\ \beta_1, \mathbf{out}\ \gamma_1 \vee \gamma_2] \to \mathrm{Var}[\beta_2, \gamma_2] \to (\mathrm{Code}[\mathbf{out}\ \beta_1, \mathbf{out}\ \gamma_1] \xrightarrow{\delta} \alpha) \to (\top \xrightarrow{\delta} \alpha) \xrightarrow{\delta} \alpha$
$\mathbf{subst}\ t_1\ t_2\ \mathbf{in}\ t_3 \rightsquigarrow \mathrm{subst}\ t_1\ t_2\ t_3$	$\mathrm{subst} : \forall \beta_1, \beta_2, \gamma_1, \gamma_2.$ $\mathrm{Var}[\beta_1, \gamma_1] \to \mathrm{Code}[\mathbf{out}\ \beta_1, \mathbf{out}\ \gamma_2] \to \mathrm{Code}[\mathbf{out}\ \beta_2, \mathbf{out}\ \gamma_1 \vee \gamma_2] \to \mathrm{Code}[\mathbf{out}\ \beta_2, \mathbf{out}\ \gamma_2]$

Definition A.1 (Primitive Definition for Quasiquotes).

$$
\begin{array}{l}
\mathbf{class}\ \mathrm{CodeBase}[\alpha, \beta, \gamma]\ \mathbf{with\ constructor} \\
\quad \mathrm{Var}[\alpha, \beta](\mathrm{Str})\ \mathbf{extends}\ \mathrm{CodeBase}[\alpha, \beta, \mathbf{out}\ \bot] \\
\quad \mathrm{Abs}[\alpha, \beta, \gamma, \delta](\mathrm{Var}[\alpha, \delta], \mathrm{Code}[\mathbf{out}\ \beta, \mathbf{out}\ \gamma \vee \delta]) \\
\qquad \mathbf{extends}\ \mathrm{CodeBase}[\mathbf{out}\ \alpha \to \beta, \mathbf{out}\ \gamma, \mathbf{out}\ \top] \\
\quad \mathrm{App}[\alpha, \beta, \gamma](\mathrm{Code}[\mathbf{out}\ \alpha \to \beta, \mathbf{out}\ \gamma], \mathrm{Code}[\mathbf{out}\ \alpha, \mathbf{out}\ \gamma]) \\
\qquad \mathbf{extends}\ \mathrm{CodeBase}[\mathbf{out}\ \beta, \mathbf{out}\ \gamma, \mathbf{out}\ \top]
\end{array}
$$

We use the shorthand $\mathrm{Code}[\mathbf{out}\ \tau, \mathbf{out}\ \sigma]$ for $\mathrm{CodeBase}[\mathbf{out}\ \tau, \mathbf{out}\ \sigma, \mathbf{out}\ \top]$ and $\mathrm{Var}[\tau, \sigma]$ for $\mathrm{CodeBase}[\tau, \sigma, \mathbf{out}\ \bot]$ for simplicity.

All typing rules for quasiquotes can also be encoded as built-in functions. We can directly construct an `App` instance via the construct function. The remaining functions are shown in Table 3. $\underline{\lambda} x.\ t$ creates a fresh next-stage variable and passes it to the continuation. The continuation is a higher-ranked function to ensure the next-stage variable will not be compiled inside the body. It finally yields a quoted term that represents the body of the lambda function. run t compiles and executes the quoted program fragments. The contextual requirement of t must be $\bot$, which reflects that t is closed. This is essentially the same as the approach we guarantee a term is pure in §3: if the effect is $\bot$, we show that the term will not modify the heap; if the contextual requirement is $\bot$, the quoted term contains no free variable. $\mathbf{try}\ y = t_1 \setminus t_2\ \mathbf{in}\ t_3\ \mathbf{else}\ t_4$ checks if a given next-stage variable, indicated by t_2, is actually free in the quoted term t_1. If it is, we rebind the term to y and evaluate the first branch. Otherwise, the fallback branch will be executed. $\mathbf{subst}\ t_1\ t_2\ \mathbf{in}\ t_3$ substitutes the given next-stage variable with another quoted term. Notice that both functions requires a next-stage variable that is still in the scope. An escaped one, typed as $\mathrm{Var}[\tau, \mathbf{in}\ \bot\ \mathbf{out}\ \top]$ for some τ, will be rejected.

B Case Studies

B.1 A Dynamic Programming Example

Let's study a practical example of InvalML. We first show some necessary builtin functions as follows. `iter` function iterates over the given `ArrayList` and has the same type as we introduced in §2, while `revIter` does the iteration in the reversal direction. `next` function checks if the given iteratror reaches the end of the corresponding `ArrayList`, returning a `Some` object with the data if there are still remaining data and returning `None` if the iteration is over. Since we do not introduce **`do-while`** syntax, we encode it as a function `whileDo` that terminates the loop if the argument function returns **`false`**. `push` function appends a new element into the given `ArrayList`. `init`, `update`, `get` are used for `Array2D`'s initialization, modification, and data accessing, respectively. Finally, `max` function is provided to pick the larger one from the given two integers.

```
fun iter, revIter: ∀ Res, R, T {E extends ¬R}.
  (ArrayList[T, R], ∀ S. Iter[T, S] →{S ∨ E} Res) →{E ∨ R} Res
fun next: ∀ T, S. Iter[T, S] →{S} Option[T]
fun whileDo: ∀ R. (() →{R} Bool) →{R} ()
fun push: ∀ A, R. (ArrayList[A, R], A) →{R} ()
fun init: [A, R] → (Region[out R], Int, Int, A) →{R} Array2D[A, R]
fun update: [A, R] → (Array2D[A, R], Int, Int, A) →{R} ()
fun get: [A, R] → (Array2D[A, R], Int, Int) →{R} A
fun max: (Int, Int) → Int
```

Our task is to select some interviewees from the candidate list within a limited budget, such that the sum of the selected interviewees' estimation scores is highest, which is a classical Knapsack problem. Suppose `dp[i][j]` indicates the maximum score we can get for the first `i`-th candidates and the budget `j`. Then consider having one more candidate. For any budget `k`, either we ignore the candidate (i.e., `dp[i][k] = dp[i - 1][k]`) or we pick the candidate (i.e., `dp[i][k] = dp[i - 1][k - salary] + score`). We demonstrate the implementation of `select` function as follows:

```
class Interv with constructor Interv(score: Int, salary: Int)

fun select(candidates, budget, results) =
  region r
  let size = len(interviewees), let i = r.ref 1
  let dp = init(r, size + 1, budget + 1, 0)
  iter of interviewees, it ⇒
    whileDo of () ⇒ if next(it) is
      Some(Interv(score, salary)) then
        let j = r.ref 0
        whileDo of () ⇒
          if !j < salary then update(dp, !i, !j, get(dp, !i - 1, !j))
          else
            let p = get(dp, !i - 1, !j - salary), let np = get(dp, !i - 1, !j)
            update(dp, !i, !j, max of np, p + score)
          j := !j + 1; !j ≤ budget
        i := !i + 1; true
      None then false
  // ... (continued below) ...
```

Recall that `iter` prevents the original region from being used while the provided argument function executes. Notice that we have several mutable reference operations during the iteration. We must reason that the local region `r` is separated from all possible outer regions. InvalML can infer the type of `candidates`, thanks to the outer variable ω. Assume that candidates : ArrayList[Interv, α^1] for some fresh type variable α^1 and r : Region[**out** β], where β's level is 2. InvalML will assign $\neg\omega$ to β's upper bound. For any operation on `r`, our algorithm will ensure β is different from α^1 by solving constraint $\alpha^1 \leq \neg\beta$, which leads to an *expected* extrusion that widens β to $\neg\omega$. Therefore, we can get $\alpha^1 \leq \neg\neg\omega$, and it is equivalent to $\alpha^1 \leq \omega$. This tells us that `candidates` is an `ArrayList` that is stored in an outer region, and any operation on local regions will not break the iteration.

Finally, we want to copy all selected candidates into another `ArrayList`. We traverse the candidate list in reversal direction. If for the `i`-th interviewee and remaining budget `rest`, the maximum score is derived from `dp[i - 1][rest - salary]`, then we know this interviewee should be put into the `results` list: [12]

```
  i := size, let rest = r.ref budget
  revIter of interviewees, it ⇒
    whileDo of () ⇒
      if next(it) is
        Some(Interv(score, salary)) then
```

[12]'**if** e1 **do** e2' is a syntax sugar for '**if** e1 **then** e2; () **else** ()'.

```
          if get(dp, !i, !rest) == get(dp, !i - 1, !rest - salary) + score
            do push(results, Interv(score, salary)); rest := !rest - salary
          i := !i - 1; true
        None then false
  get(dp, size, budget)
```

This program can be correctly type checked by InvalML. If users wrongly wrote the code `push(candidates, Interv(score, salary))`, InvalML would reject all applications of `select` since we cannot modify `candidates` when iterating over it. Finally, our algorithm will infer the following type for `select` function: $\forall \alpha, \delta, \omega \{\alpha \leq \omega, \delta \leq \neg\alpha\}. \, (\mathsf{ArrayList}[\mathsf{Interv}, \, \alpha], \mathsf{Int}, \mathsf{ArrayList}[\mathsf{Interv}, \, \delta]) \overset{\alpha \vee \delta}{\rightarrow} \mathsf{Int}$.

B.2 A Constraint Solver Example

In this section, let's implement the constraint solver algorithm proposed by Parreaux [66]. We first show some necessary builtin functions as follows. Most functions are the same as we have seen in the previous section. We further introduce `foreach` function that takes an iterator and an argument function. It checks if the iterator reaches the end of the `ArrayList` by calling `next`. If there exists a value, it passes the value to the argument function.

```
fun empty: ∀ A, R. Region[out R] →{R} ArrayList[A, R]
fun push: ∀ A, R. (ArrayList[A, R], A) →{R} ()
fun iter: ∀ Res, R, T {E extends ¬R}.
  (ArrayList[T, R], ∀ S. Iter[T, S] →{S ∨ E} Res) →{E ∨ R} Res
fun next: ∀ T, S. Iter[T, S] →{S} Option[T]
fun whileDo: ∀ R. (() →{R} Bool) →{R} ()
fun foreach: foreach E, R, T. (Iter[T, R], T →{E} ()) →{R ∨ E} ()
```

We now define `Type` as follows. `Type` takes a type argument to indicate which region it makes use of to store the mutable lower bounds and upper bounds of type variables. For simplicity, we omit other constructors. A simplified `solve` function is given. It takes an immutable list of constraints to be solved. If the list is empty, the function does nothing. Otherwise, it matches the left-hand side of the constraint against constructors. Let's focus on the most interesting case, where the constraint has a shape like $\alpha \leq \tau$ for some type variable α and some type τ. If τ has a higher level, we must extrude τ to get rid of level violations. Otherwise, we can append `rhs` into the upper bound list of the current type variable and propagate `rhs` by constrain all lower bounds of the current type variable to be less than `rhs`. Notably, we create a local region and a reference storing the constraint list. During the iteration, we keep pushing new constraints into `ncs`. The recursive call happens when the iteration over.

```
class Type[R] with
  constructor
    IntType()
    /* other constructors */
    TypeVariable(id: Str, lvl: Int,
      lowerBds: ArrayList[Type[out R], R], upperBds: ArrayList[Type[out R], R])

fun solve(constraints) = if constraints is
  Nil() then ()
  Cons(Pair(lhs, rhs), cs) then if lhs is
    /* other cases */
    TypeVariable(name, level, lb, ub) then
      if levelOf(rhs) ≤ level then
        push(ub, rhs)
        region r in
```

$$\boxed{\Xi \vdash \mathcal{T} \leq^{\forall} \mathcal{T}} \qquad \textsc{S-PTop}\ \frac{}{\Xi \vdash \mathcal{T} \leq^{\forall} \top} \qquad \textsc{S-PRefl}\ \frac{}{\Xi \vdash \mathcal{T} \leq^{\forall} \mathcal{T}} \qquad \textsc{S-Mono}\ \frac{\Xi \vdash \tau \leq \sigma}{\Xi \vdash \tau \leq^{\forall} \sigma}$$

$$\textsc{S-Forall}\ \frac{\begin{array}{c}\Sigma = \overline{\alpha_i \leq^{\pm i} \tau_i}^{i} \qquad \Sigma' = \overline{\alpha_i \leq^{\pm i} \sigma_i}^{i} \\ \Xi\,\Sigma' \vdash \mathcal{T} \leq^{\forall} \mathcal{S} \qquad \Xi \vdash \forall V\{\Sigma'\}\ \textbf{\textit{cons.}} \qquad \overline{\Xi \vdash \sigma_i \leq^{\pm i} \tau_i}^{i}\end{array}}{\Xi \vdash \forall V\{\Sigma\}.\,\mathcal{T} \leq^{\forall} \forall V\{\Sigma'\}.\,\mathcal{S}}$$

$$\textsc{S-PFun}\ \frac{\Xi \vdash \mathcal{T}_1 \leq^{\forall} \mathcal{T}_2 \qquad \Xi \vdash \mathcal{T}_3 \leq^{\forall} \mathcal{T}_4 \qquad \Xi \vdash \tau_5 \leq \tau_6}{\Xi \vdash \mathcal{T}_2 \xrightarrow{\tau_5} \mathcal{T}_3 \leq^{\forall} \mathcal{T}_1 \xrightarrow{\tau_6} \mathcal{T}_4}$$

Fig. 11. General subtyping rules.

```
        let ncs = r.ref cs
        iter(lb, it ⇒ foreach(it, b ⇒ ncs := Cons(Pair(b, rhs), !ncs); ()))
        solve(!ncs)
      else /* extrusion */
```

Again, `iter` prevents the original region from being used while the provided argument function executes. Due to the mutable operations on the region `r` and reference `ncs`, we must know that all type information is stored in another region that has no interference with `r`. Thanks to the outer variable and subtype extrusion, InvalML can infer type List[**out** Pair[**out** Type[**out** α], **out** Type[**out** α]]] for `constraints`, where $\alpha \leq \omega$.

Finally, if we propagate the new upper bound and solve corresponding constraints on the fly, shown as follows, it is possible that these new constraints might introduce new upper bounds to the current type variable, which can break the iteration and lead to wong propagation. In this case, InvalML can infer type List[**out** Pair[**out** Type[**out** $\bot$], **out** Type[**out** $\bot$]]] for `constraints`. It is only useful when there is no type variable in the given constraints.

```
fun solve(constraints) = if constraints is
  Nil() then ()
  Cons(Pair(lhs, rhs), cs) then if lhs is
    /* other cases */
    TypeVariable(name, level, lb, ub) then
      if levelOf(rhs) ≤ level then
        push(ub, rhs)
        iter(lb, it ⇒ foreach(it, b ⇒ solve(Cons(Pair(b, rhs), Nil()))))
        solve(cs)
      else /* extrusion */
```

C Additional Definitions

This appendix lists the missing definitions of $\lambda^{!\perp}$ which did not fit in the main body of the paper.

C.1 General Subtyping and Entailment

Though merely monomorphic subtyping is allowed in T-Subs1 and T-Subs2, one can still partially upcast monomorphic types nested in polymorphic types via data types. Assume that we have **class** A[α] **with constructor** $C(\forall\beta.\,\beta \rightarrow \alpha) \in \mathcal{D}$ and $\Gamma(x) = \mathsf{A}[\textbf{out}\,\mathsf{Nat}]$. In the declarative system, T-Subs1 can be arbitrarily used to upcast x, for example, to A[**out** Int] with the assumption Nat $\leq$ Int. After the upcasting operation, extracting the field of x via pattern matching will be typed to $\forall\beta.\,\beta \rightarrow \mathsf{Int}$, instead of the original $\forall\beta.\,\beta \rightarrow \mathsf{Nat}$. This will not introduce any problem to the system's soundness but bring inconvenience when we discuss the value typing and inference algorithm. Therefore, we define general subtyping relations in Figure 11. We also define the entailment judgment $\Xi_1 \models \Xi_2$ in Figure 12 to ensure that all subtyping relations in Ξ_2 hold in Ξ_1.

$$\boxed{\Xi \vDash \Xi} \qquad \textsc{S-Empty}\ \frac{}{\Xi \vDash \epsilon} \qquad \textsc{S-Cons}\ \frac{\Xi_1 \vDash \Xi_2 \quad \Xi_1 \vdash \tau \leq \sigma}{\Xi_1 \vDash \Xi_2\ (\tau \leq \sigma)} \qquad \textsc{S-Cons}\rhd\ \frac{\Xi_1 \vDash \Xi_2 \quad \lhd\Xi_1 \vdash \tau \leq \sigma}{\Xi_1 \vDash \Xi_2 \rhd (\tau \leq \sigma)}$$

Fig. 12. Subtyping context entailment rules.

C.2 Well-Formedness, Substitution, and Consistency

The well-formedness rules are given in Figure 13. All well-formed types can only refer to other well-formed types, declared type variables, and well-defined classes. For any generalized type, the well-formedness also checks the bound context Σ to ensure the absence of recursive occurrences of type variables, such as $\alpha \leq \alpha \vee \beta$ and $\alpha \leq \beta, \beta \leq \alpha$. We give the formal definition of *guarded* check in Figure 15. As we mentioned in §2, we check the intersections and unions $\tau_1 \wedge^{\pm} \tau_2$ to prevent a type variable from being composed with a function type or a data type that contains a nested type variable. The reason we need this well-formedness check, and the '*gd*' function in particular, is that without them, there would be some terms that would be typeable in the declarative system but that our algorithm would reject. Thankfully, these are mostly corner cases that do not arise in practice for a system like InvalML.

The substitution $\rho = [\overline{\tau/v}]$ is formally defined in Figure 16. The *subtyping context consistency* rules are given in Figure 17. They largely follow those of Chau and Parreaux [14]. Notice that we write $\diamond = pol(\alpha_i)$ to retrieve the polarity of α_i. If α_i is only in a positive position, then τ_i is required to be well-formed in a positive positions. If α_i appears in both positive and negative positions, we require $\Gamma \vdash \tau\ \boldsymbol{wf}^{+}$ and $\Gamma \vdash \tau\ \boldsymbol{wf}^{-}$.

Lemma C.1. *If* Γ $\boldsymbol{wf}$, *then* $\Gamma, \zeta \vdash t : \mathcal{T}\ !\ \varphi$ *implies* $\Gamma \vdash \varphi\ \boldsymbol{wf}^{+}$, $\Gamma \vdash \mathcal{T}\ \boldsymbol{wf}^{+}$, *and* $\Gamma \vdash \zeta\ \boldsymbol{wf}^{+}$.

Lemma C.2. *If* Γ $\boldsymbol{wf}$, *then* $sub(\Gamma) \vdash \mathcal{T}_1 \leq^{\forall} \mathcal{T}_2$ *implies both* $\Gamma \vdash \mathcal{T}_1\ \boldsymbol{wf}^{+}$ *and* $\Gamma \vdash \mathcal{T}_2\ \boldsymbol{wf}^{-}$.

Example C.3. Consider the term $f(a)(g)$, where $f : \forall\alpha.\, \alpha \to (\forall\beta.\, (C[\mathbf{out}\,\beta] \wedge \alpha) \to \beta) \to \mathsf{Int}$, $g : \forall\delta.\, C[\mathbf{in}\,\mathsf{Int}\ \mathbf{out}\,\delta] \to \delta$, and $a : C[\mathbf{in}\,\mathsf{Int}]$. If we didn't have the '*gd*' well-formedness condition, the term below would be typable in the declarative system:

$$\frac{\dfrac{\textbf{(2)}\ \dfrac{f : \forall\alpha.\, \alpha \to \ldots}{f : C[\mathbf{in}\,\mathsf{Int}] \to \ldots} \qquad a : C[\mathbf{in}\,\mathsf{Int}]}{f(a) : (\forall\beta.\, (C[\mathbf{out}\,\beta] \wedge C[\mathbf{in}\,\mathsf{Int}]) \to \beta) \to \mathsf{Int}} \qquad \dfrac{\textbf{(1)}\ \dfrac{\dfrac{g : \forall\delta.\, C[\mathbf{in}\,\mathsf{Int}\ \mathbf{out}\,\delta] \to \delta}{g : C[\mathbf{in}\,\mathsf{Int}\ \mathbf{out}\,\beta] \to \beta}}{g : (C[\mathbf{out}\,\beta] \wedge C[\mathbf{in}\,\mathsf{Int}]) \to \beta}}{g : \forall\beta.\, (C[\mathbf{out}\,\beta] \wedge C[\mathbf{in}\,\mathsf{Int}]) \to \beta}}{f(a)(g) : \mathsf{Int}}$$

Notice that the step **(1)** is done via T-Subs1 with S-Fun, S-CtorMrg+, and S-Refl. However, the inference algorithm will reject this program. In step **(2)**, the algorithm will allocate a fresh variable α' for α instead of *guessing* a type for α, and α' has a lower bound $C[\mathbf{in}\,\mathsf{Int}]$. In step **(1)**, the algorithm will try to constrain $C[\mathbf{out}\,\beta] \wedge \alpha' \ll C[\mathbf{in}\,\mathsf{Int}\ \mathbf{out}\,\delta']$, where δ' is a fresh variable for δ. Later δ' will be constrained to β, which leads to a new upper bound $C[\mathbf{in}\,\mathsf{Int}\ \mathbf{out}\,\beta]$ of α'. This cannot hold, since we cannot assert that $\top \leq \beta$.

Notice how we pick $C[\mathbf{in}\,\mathsf{Int}]$ for α and β for δ and S-CtorMrg+ is applicable since $\beta \wedge \tau \leq \beta$ always holds for any τ. By contrast, the algorithm can only solve the constraints by C-Var3, which yields a bad constraint $\top \leq \beta$.

C.3 Value Typing and Context Conformance

We define the value typing rules and context conformance in Figure 18. As we explained in §C.1, we allow a more flexible but harmless form of subtyping in T-VSubs. Rule T-RegA is standard. We use

$$\boxed{\mathcal{D}\ \boldsymbol{wf}} \qquad \textsc{W-Dec}\ \dfrac{\overline{d\ \boldsymbol{wf}}^{d\in\mathcal{D}}}{\mathcal{D}\ \boldsymbol{wf}} \qquad \boxed{d\ \boldsymbol{wf}} \qquad \textsc{W-ADT}\ \dfrac{\overline{\mathsf{A}, \{\overline{\alpha}\} \vdash c\ \boldsymbol{wf}}}{\textbf{class}\ \mathsf{A}[\overline{\alpha}]\ \textbf{with constructor}\ \overline{c}\ \boldsymbol{wf}} \qquad \boxed{\Gamma \vdash a\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-Arg}\ \dfrac{\Gamma \vdash \tau\ \boldsymbol{wf}^{\mp} \quad \Gamma \vdash \sigma\ \boldsymbol{wf}^{\pm}}{\Gamma \vdash \textbf{in}\ \tau\ \textbf{out}\ \sigma\ \boldsymbol{wf}^{\pm}}$$

$$\boxed{\mathsf{A}, \{\overline{\alpha}\} \vdash c\ \boldsymbol{wf}} \qquad \textsc{W-Ctor1}\ \dfrac{\overline{\{\overline{\alpha}\} \vdash \mathcal{T}\ \boldsymbol{wf}^{+}}}{\mathsf{A}, \{\overline{\alpha}\} \vdash \mathsf{C}(\overline{\mathcal{T}})\ \boldsymbol{wf}} \qquad \textsc{W-Ctor2}\ \dfrac{\overline{\{\overline{\beta}\} \vdash \mathcal{T}\ \boldsymbol{wf}^{+}} \quad \overline{\{\overline{\beta}\} \vdash b\ \boldsymbol{wf}^{+}} \quad |\overline{b}| = |\overline{\alpha}|}{\mathsf{A}, \{\overline{\alpha}\} \vdash \mathsf{C}[\overline{\beta}](\overline{\mathcal{T}})\ \textbf{extends}\ \mathsf{A}[\overline{b}]\ \boldsymbol{wf}}$$

$$\boxed{\Gamma \vdash \mathcal{T}\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-Top}\diamond\ \dfrac{}{\Gamma \vdash \top^{\diamond}\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-Var}\ \dfrac{v \in \Gamma}{\Gamma \vdash v\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-AndOr}\diamond\ \dfrac{\Gamma \vdash \tau_1\ \boldsymbol{wf}^{\pm} \quad \Gamma \vdash \tau_2\ \boldsymbol{wf}^{\pm} \quad gd^{\pm}_{sub(\Gamma)}(\tau_1 \wedge^{\diamond} \tau_2)}{\Gamma \vdash \tau_1 \wedge^{\diamond} \tau_2\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-Neg}\ \dfrac{\Gamma \vdash \tau\ \boldsymbol{wf}^{\mp}}{\Gamma \vdash \neg\tau\ \boldsymbol{wf}^{\pm}}$$

$$\textsc{W-Fun}\ \dfrac{\Gamma \vdash \mathcal{T}_1\ \boldsymbol{wf}^{\mp} \quad \Gamma \vdash \mathcal{T}_2\ \boldsymbol{wf}^{\pm} \quad \Gamma \vdash \varphi\ \boldsymbol{wf}^{\pm}}{\Gamma \vdash \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-Ctor}\ \dfrac{\overline{\Gamma \vdash a\ \boldsymbol{wf}^{\pm}} \quad \textbf{class}\ \mathsf{A}[\overline{\alpha}] \in \mathcal{D} \quad |\overline{a}| = |\overline{\alpha}|}{\Gamma \vdash \mathsf{A}[\overline{a}]\ \boldsymbol{wf}^{\pm}} \qquad \textsc{W-Forall}\ \dfrac{\Gamma \bullet V \vdash \Sigma\ \boldsymbol{wf} \quad guarded(\Sigma) \quad v, \boldsymbol{err}, \bullet \notin \Sigma \quad \Gamma \bullet V\ \Sigma \vdash \mathcal{T}\ \boldsymbol{wf}^{\pm}}{\Gamma \vdash \forall V\{\Sigma\}.\mathcal{T}\ \boldsymbol{wf}^{\pm}}$$

$$\boxed{\Gamma \vdash \Sigma\ \boldsymbol{wf}} \qquad \textsc{W-Empty}\ \dfrac{}{\Gamma \vdash \epsilon\ \boldsymbol{wf}} \qquad \textsc{W-Bound}\ \dfrac{\Gamma \vdash \Sigma\ \boldsymbol{wf} \quad \Gamma \vdash \alpha\ \boldsymbol{wf}^{\pm} \quad \Gamma \vdash \tau\ \boldsymbol{wf}^{\mp}}{\Gamma \vdash \Sigma\ (\alpha \leq^{\pm} \tau)\ \boldsymbol{wf}} \qquad \textsc{W-TV}\ \dfrac{\Gamma \vdash \Sigma\ \boldsymbol{wf} \quad \alpha \in \Gamma}{\Gamma \vdash \Sigma\ \alpha\ \boldsymbol{wf}} \qquad \textsc{W-Sep}\ \dfrac{\Gamma \vdash \Sigma\ \boldsymbol{wf} \quad \Gamma \bullet V \vdash \Sigma'\ \boldsymbol{wf}}{\Gamma \vdash \Sigma \bullet V\ \Sigma'\ \boldsymbol{wf}}$$

$$\boxed{\Gamma \vdash \Xi\ \boldsymbol{wf}} \qquad \textsc{W-Empty}\ \dfrac{}{\Gamma \vdash \epsilon\ \boldsymbol{wf}} \qquad \textsc{W-Hyp}\ \dfrac{\Gamma \vdash \Xi\ \boldsymbol{wf} \quad \Gamma \vdash \tau\ \boldsymbol{wf}^{\pm} \quad \Gamma \vdash \sigma\ \boldsymbol{wf}^{\mp}}{\Gamma \vdash \Xi\ (\tau \leq^{\pm} \sigma)\ \boldsymbol{wf}} \qquad \textsc{W-Assum}\ \dfrac{\Gamma \vdash \Xi\ \boldsymbol{wf} \quad \Gamma \vdash \tau\ \boldsymbol{wf}^{+} \quad \Gamma \vdash \sigma\ \boldsymbol{wf}^{-}}{\Gamma \vdash \Xi \rhd (\tau \leq \sigma)\ \boldsymbol{wf}}$$

$$\boxed{\Gamma\ \boldsymbol{wf}} \qquad \dfrac{\overline{\Gamma \vdash \mathcal{T}\ \boldsymbol{wf}^{+}}^{(x:\mathcal{T})\in\Gamma} \quad sub(\Gamma)\ \boldsymbol{wf}}{\Gamma\ \boldsymbol{wf}} \qquad \boxed{\Gamma \vdash \Psi\ \boldsymbol{wf}} \qquad \textsc{W-Reg}\ \dfrac{\Gamma \vdash \Psi\ \boldsymbol{wf} \quad \Gamma \vdash \alpha\ \boldsymbol{wf}^{+}}{\Gamma \vdash \Psi\ (\mathsf{r} : \alpha)\ \boldsymbol{wf}} \qquad \textsc{W-Loc}\ \dfrac{\Gamma \vdash \Psi\ \boldsymbol{wf} \quad \Gamma \vdash \tau\ \boldsymbol{wf}^{+}}{\Gamma \vdash \Psi\ (\ell : \tau)\ \boldsymbol{wf}}$$

Fig. 13. Well-formedness rules.

$$\boxed{gd^{\pm}_{\Sigma}(\tau) \triangleq \textit{for all } C \textit{ s.t. } \tau \rightsquigarrow^{\pm} C, \textit{if } \alpha \in C \textit{ for some } \alpha, \textit{then for all } \sigma \in C \textit{ s.t. } \sigma \neq v, \boldsymbol{lv}(\sigma, \Sigma) = 0} \qquad \dfrac{}{\top^{\diamond} \rightsquigarrow^{\pm} \emptyset}$$

$$\dfrac{}{\tau \xrightarrow{\varphi} \sigma \rightsquigarrow^{\pm} \{\tau \xrightarrow{\varphi} \sigma\}} \qquad \dfrac{}{\mathsf{A}[\overline{a}] \rightsquigarrow^{\pm} \{\mathsf{A}[\overline{a}]\}} \qquad \dfrac{}{v \rightsquigarrow^{\pm} \{v\}} \qquad \dfrac{ub^{\pm}_{\Sigma}(\alpha) \rightsquigarrow^{\pm} C}{\alpha \rightsquigarrow^{\pm} C} \qquad \dfrac{\tau \rightsquigarrow^{\mp} C}{\neg\tau \rightsquigarrow^{\pm} C}$$

$$\dfrac{\tau \rightsquigarrow^{\pm} C_1 \quad \sigma \rightsquigarrow^{\pm} C_2 \quad mgd^{\pm}_{\Sigma}(\tau \wedge^{\pm} \sigma)}{(\tau \wedge^{\pm} \sigma) \rightsquigarrow^{\pm} C_1 \cup C_2} \qquad \dfrac{\tau \rightsquigarrow^{\pm} C}{(\tau \vee^{\pm} \sigma) \rightsquigarrow^{\pm} C} \qquad \dfrac{\sigma \rightsquigarrow^{\pm} C}{(\tau \vee^{\pm} \sigma) \rightsquigarrow^{\pm} C}$$

$$\dfrac{gd^{\mp}_{\Sigma}(\tau_1 \vee^{\pm} \sigma_1) \quad gd^{\pm}_{\Sigma}(\tau_2 \wedge^{\pm} \sigma_2) \quad gd^{\pm}_{\Sigma}(\varphi \wedge^{\pm} \varphi')}{mgd^{\pm}_{\Sigma}((\tau_1 \xrightarrow{\varphi} \tau_2) \wedge^{\pm} (\sigma_1 \xrightarrow{\varphi'} \sigma_2))} \qquad \dfrac{\overline{gd^{\mp}_{\Sigma}(\tau_i \vee^{\pm} \sigma_i)} \quad \overline{gd^{\pm}_{\Sigma}(\tau'_i \wedge^{\pm} \sigma'_i)}}{mgd^{\pm}_{\Sigma}(\mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \tau'_i}] \wedge^{\pm} \mathsf{A}[\overline{\textbf{in}\ \sigma_i\ \textbf{out}\ \sigma'_i}])}$$

$$\dfrac{\boldsymbol{lv}(\Sigma, \tau_1 \xrightarrow{\varphi} \tau_2) \times \boldsymbol{lv}(\Sigma, \sigma_1 \xrightarrow{\varphi'} \sigma_2) = 0}{mgd^{\pm}_{\Sigma}((\tau_1 \xrightarrow{\varphi} \tau_2) \wedge^{\pm} \neg(\sigma_1 \xrightarrow{\varphi'} \sigma_2))} \qquad \dfrac{\boldsymbol{lv}(\Sigma, \mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \tau'_i}]) \times \boldsymbol{lv}(\Sigma, \mathsf{A}[\overline{\textbf{in}\ \sigma_i\ \textbf{out}\ \sigma'_i}]) = 0}{mgd^{\pm}_{\Sigma}(\mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \tau'_i}] \wedge^{\pm} \neg\mathsf{A}[\overline{\textbf{in}\ \sigma_i\ \textbf{out}\ \sigma'_i}])}$$

Fig. 14. Well-formed intersections and unions checking.

T-RegD to type a dead region. Since a closure may encapture a dead region, passing it to another function that requires a region with the same `in` and `out` parameters, we cannot directly widen it to Region[**out in** $\bot$ **out** $\neg\zeta$]. T-Ctor instantiates the polymorphic construction function first and

$$\boxed{guarded(\Sigma) \triangleq for\ all\ \alpha \in FV(\Sigma),, if\ lb_{\Sigma}(\alpha) \succ^{-} C_1 and ub_{\Sigma}(\alpha) \succ^{+} C_2, then\ \alpha \notin C_1 \cup C_2} \qquad \frac{}{\top^{\diamond} \succ^{\pm} \emptyset}$$

$$\frac{}{\tau \xrightarrow{\varphi} \sigma \succ^{\pm} \emptyset} \qquad \frac{}{\mathsf{A}[\overline{a}] \succ^{\pm} \emptyset} \qquad \frac{ub^{\pm}_{\Sigma}(\nu) \succ^{\pm} C}{\nu \succ^{\pm} C \cup \{\nu\}} \qquad \frac{\tau \succ^{\mp} C}{\neg\tau \succ^{\pm} C} \qquad \frac{\tau \succ^{\pm} C_1 \quad \sigma \succ^{\pm} C_2}{(\tau \wedge^{\diamond} \sigma) \succ^{\pm} C_1 \cup C_2}$$

Fig. 15. Guarded bound context checking.

$$\rho(\nu) = \begin{cases} \tau & \text{if } [\tau/\nu] \in \rho \\ \alpha & \text{if } \nu \notin dom(\rho) \end{cases} \qquad \rho(\tau \xrightarrow{\varphi} \sigma) = \rho(\tau) \xrightarrow{\rho(\varphi)} \rho(\sigma)$$

$$\rho(\mathsf{A}[\overline{a_i}]) = \mathsf{A}[\overline{\rho(a_i)}] \qquad \rho(\top^{\pm}) = \top^{\pm}$$

$$\rho(\tau \vee^{\pm} \sigma) = \rho(\tau) \vee^{\pm} \rho(\sigma) \qquad \rho(\neg\tau) = \neg(\rho(\tau))$$

$$\rho(\forall V\{\Sigma\}.\,\mathcal{T}) = \forall V\{\rho(\Sigma)\}.\,\rho(\mathcal{T}) \qquad \rho(\mathcal{T} \xrightarrow{\varphi} \mathcal{S}) = \rho(\mathcal{T}) \xrightarrow{\rho(\varphi)} \rho(\mathcal{S})$$

$$\rho(\mathbf{in}\,\tau\,\mathbf{out}\,\sigma) = \mathbf{in}\,\rho(\tau)\,\mathbf{out}\,\rho(\sigma) \qquad \rho(\tau \leq \sigma) = \rho(\tau) \leq \rho(\sigma)$$

$$dom([\,]) = \emptyset \qquad dom(\overline{[\tau/\nu]} \circ [\sigma/\nu']) = dom(\overline{[\tau/\nu]}) \cup \{\nu'\}$$

Fig. 16. Formal definition of type substitution

$$\boxed{\Gamma\ \boldsymbol{cons.}} \qquad \frac{\epsilon \vdash sub(\Gamma)\ \boldsymbol{cons.}}{\Gamma\ \boldsymbol{cons.}}$$

$$\boxed{\Xi \vdash \Xi\ \boldsymbol{cons.}} \qquad \frac{\rho = \overline{[\tau_i/\alpha_i]}^{i} \quad \overline{\Xi_1 \vdash \tau_i\ \boldsymbol{wf}^{\diamond} \quad \diamond = pol(\alpha_i)}^{i} \quad \rhd\Xi_2\ \rho(\Xi_1) \vDash \rho(\Xi_2)}{\Xi_1 \vdash \Xi_2\ \boldsymbol{cons.}}$$

$$\boxed{\Xi \vdash \forall V\{\Sigma\}\ \boldsymbol{cons.}} \qquad \frac{\Xi \vdash \Sigma\ \boldsymbol{cons.}}{\Xi \vdash \forall V\{\Sigma\}\ \boldsymbol{cons.}}$$

Fig. 17. Consistency of subtyping and typing contexts.

then checks each parameter. We merge the alive location typing and dead location typing rules into T-Loc, which says we need to find a store typing context Ψ' that can witness the existence of ℓ_{ζ} and a typing context Γ', such that $\Gamma' \vdash \tau\ \boldsymbol{wf}^{+}$. We immediately upcast τ to τ', such that $\Gamma \vdash \tau'\ \boldsymbol{wf}^{+}$ and make use of region typing rules to retrieve the region types. Finally, T-VGen and T-Clos are standard.

C.4 Semantics of $\lambda^{!\perp}$

The big-step semantics of $\lambda^{!\perp}$ are presented in Figure 19. We give our big-step semantics in the functional style of Radanne et al. [77]. The function `eval`, written in OCaml-inspired syntax, requires an initial runtime environment γ and an initial store ψ, manipulating term e with fuel i, and finally returns a result $\mathbb{R}$. A result can be either a pair of store and value, an error ***err***, or a timeout **kill**. The function firstly checks if time is up (i.e., runs out of fuel) and returns **kill** if so. Otherwise, it starts handling the term e by pattern matching. We borrow the `let*` expression from Radanne et al. [77]'s interpretation. It works as a monadic mapping function, evaluating the body only if the binding right-hand-side yields a correct value instead of ***err*** or **kill**. Evaluations for variables, ascriptions, lambda abstractions, and let-bindings are standard. For the `Construct` case, we retrieve the constructor function *ctor* and the length of parameters s by function `getdef`. If the length of parameters is not matched, we return ***err*** directly. `fold_left*` is a folding function that terminates immediately if the intermediate result is either ***err*** or **kill**. Evaluation for `Region` allocates a fresh

$$\boxed{\Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}} \qquad \textsc{T-VSubs}\ \frac{\Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}_1 \quad sub(\Gamma) \vdash \mathcal{T}_1 \leq^{\forall} \mathcal{T}_2}{\Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}_2} \qquad \textsc{T-RegA}\ \frac{\Psi(\mathsf{r}) = \alpha}{\Gamma \mid \Psi, \zeta \vdash \mathsf{r} : \mathsf{Region}[\mathbf{out}\,\alpha]}$$

$$\textsc{T-RegD}\ \frac{\mathsf{r} \notin dom(\Psi)}{\Gamma \mid \Psi, \zeta \vdash \mathsf{r} : \mathsf{Region}[\mathbf{out}\,\neg\zeta]} \qquad \textsc{T-Ctor}\ \frac{\Gamma \vdash \mathsf{construct}_{\mathsf{C}} : \overline{\mathcal{T}_i \to}^{i\in 1\ldots n}\ \mathsf{A}[\overline{a}] \quad \overline{\Gamma \mid \Psi, \zeta \vdash v_j : \mathcal{T}_j}^{j\in 1\ldots n}}{\Gamma \mid \Psi, \zeta \vdash \mathsf{C}(\overline{v_j}^{j\in 1\ldots n}) : \mathsf{A}[\overline{a}]}$$

$$\textsc{T-Loc}\ \frac{\Psi'(\ell_{\mathsf{r}}) = \tau \quad sub(\Gamma') \vdash \tau \leq \tau' \quad \Gamma \mid \Psi, \zeta \vdash \mathsf{r} : \mathsf{Region}[\mathbf{out}\,\sigma]}{\Gamma \mid \Psi, \zeta \vdash \ell_{\mathsf{r}} : \mathsf{Ref}[\tau', \mathbf{out}\,\sigma]}$$

$$\textsc{T-VGen}\ \frac{\Gamma\ V\ \Sigma \mid \Psi, \zeta \vee \omega \vdash v : \mathcal{T} \quad \omega \in V \quad sub(\Gamma) \vdash \forall V\{\Sigma\}\ \boldsymbol{cons.}}{\Gamma \mid \Psi, \zeta \vdash v : \forall V\{\Sigma\}.\,\mathcal{T}} \qquad \textsc{T-Clos}\ \frac{\Gamma' \mid \Psi \vDash_{\zeta} \gamma \mid \psi \quad \Gamma'\ (x : \mathcal{T}), \zeta \vdash (t : \mathcal{S}) : \mathcal{S} \mathrel{!} \varphi}{\Gamma \mid \Psi, \zeta \vdash \langle \lambda x.\, t, \gamma \rangle : \mathcal{T} \xrightarrow{\varphi} \mathcal{S}}$$

$$\boxed{\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi} \qquad \textsc{CF-Empty}\ \frac{}{\epsilon \mid \epsilon \vDash_{\bot} \epsilon \mid \epsilon} \qquad \textsc{CF-Var}\ \frac{\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi \quad \Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}}{\Gamma\ (x : \mathcal{T}) \mid \Psi \vDash_{\zeta} \gamma\ (x \mapsto v) \mid \psi} \qquad \textsc{CF-Reg}\ \frac{\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi \quad \tau = \neg\zeta}{\Gamma \bullet \alpha\ (\alpha \leq \tau) \mid \Psi\ (\mathsf{r} : \alpha) \vDash_{\zeta \vee \alpha} \gamma \mid \psi\ (\mathsf{r} \mapsto \epsilon)}$$

$$\textsc{CF-Loc}\ \frac{\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi \quad \Gamma \mid \Psi, \zeta \vdash v : \tau \quad \psi(\mathsf{r}) = \mu}{\Gamma \mid \Psi\ (\ell_{\mathsf{r}} : \tau) \vDash_{\zeta} \gamma \mid \psi\ (\mathsf{r} \mapsto (\mu\ (\ell_{\mathsf{r}} \mapsto v)))} \qquad \textsc{CF-Forall}\ \frac{\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi \quad \omega \in V \quad sub(\Gamma) \vdash \forall V\{\Sigma\}\ \boldsymbol{cons.}}{\Gamma \bullet V\ \Sigma \mid \Psi \vDash_{\zeta \vee \omega} \gamma \mid \psi}$$

Fig. 18. Typing rules for values and context conformance.

region instance r and an empty memory area μ. newreg returns both the region instance r and the new store context ψ_1, where $\psi_1 = \psi\ (\mathsf{r} \mapsto \mu)$. It then evaluates the body t with the region instance bound to x and the store context ψ_1. All locations belonging to the region r will be discarded when we exit this region, written as $\psi_2 \setminus \mathsf{r}$. `Ref` requires t_1 to yield a region instance (checked by the `getreg` function) and updates the Heap ψ'' with a fresh location. Evaluations for `Deref` and `Set` are standard. Case `Match` accepts a scrutinee s and a series of branches bs. Function `fold_left*?` works like `fold_left*`, but wrapping everything in an `Option` monad, continuing iteration only if the result is still `none`. If the instance matches the current branch, we zip the binders and the extracted values (i.e., $\overline{[x \mapsto p]}^{xs\cdot ps}$) and then evaluate the branch body.

C.5 RDNF Construction Functions

We define the *dnf* function in Figure 20. The algorithm is also inspired by $\lambda^{\neg}$ [68] but much simpler.

Lemma C.4 (Correctness of *dnf*). *For all τ and $\mathbb{D} = dnf(\tau)$, $\vdash \tau \equiv \mathbb{D}$.*[13]

C.6 Semantics of Quasiquotes

The runtime semantics of quasiquotes is shown in Figure 21. We adopt π to store the next-satge variables to ensure the generated code is hygienic. Evaluations for terms mentioned in §3 and notions for stores are omitted for simplicity.

D Full Metatheory

We now provide the proofs of the main theorems and lemmas stated in the main text.

[13]We use the standard notion of type equivalence as mutual subtyping $\tau_1 \equiv \tau_2 \overset{\text{def}}{=} \tau_1 \leq \tau_2$ *and* $\tau_2 \leq \tau_1$

```
let rec eval(γ: ctx)(ψ: st) i e: ℝ =
  if i = 0 then kill else
  let i′ = i - 1 in
  match e with
  | Var(x) →let* v = γ(x)
    in val(ψ, v)
  | Asc(e, _) →eval γ ψ i′ e
  | Let(x, t₁, t₂) →
    let* (ψ₁, v₁) =
      eval γ ψ i′ t₁
    in eval (γ[x ↦ v₁]) ψ₁ i′ t₂
  | Abs(x, e) →val(ψ, ⟨λx.e, γ⟩)
  | App(f, a) →
    let* (ψ₁, v₁) =
      eval γ ψ i′ f in
    let* (ψ₂, v₂) =
      eval γ ψ₁ i′ a in
    let* (x, e, γ′) = getclos v₁
    in eval (γ′[x ↦ v₂]) ψ₂ i′ e
  | Construct(c, xs) →
    let* (ctor, s) = getdef c in
    if List.length xs <> s
    then err else let* (ψ′, vs) =
    fold_left* fun (ψ₀, r) x →
      let* (ψ′, v) = eval γ ψ₀ i′ x
      in (ψ′, v :: r)
    (ψ, []) xs
    in val(ψ′, ctor(rev vs))
  | Region(x, t) →
    let (r, ψ₁) = newreg ψ in
    let* (ψ₂, v) =
      eval (γ[x ↦ r]) ψ₁ i′ t
    in val(ψ₂ \ r, v)
  | Ref(t₁, t₂) →
    let* (ψ₁, v₁) =
      eval γ ψ i′ t₁ in
    let* (ψ₂, v₂) =
      eval γ ψ₁ i′ t₂ in
    let* r = getreg ψ₂ v₁ in
    let* (ψ₃, ℓᵣ) =
      alloc ψ₂ r v₂
    in val(ψ₃, ℓᵣ)
  | Deref(t) →
    let* (ψ₁, v₁) =
      eval γ ψ i′ t in
    let* ℓᵣ = getloc ψ₁ v₁ in
    let* v₂ = read ψ₁ ℓᵣ in
    val(ψ₁, v₂)
  | Set(t₁, t₂) →
    let* (ψ₁, v₁) =
      eval γ ψ i′ t in
    let* ℓᵣ = getloc ψ₁ v₁ in
    let* (ψ₂, v₂) =
      eval γ ψ₁ i′ t₂ in
    let* ψ₃ = write ψ₂ ℓᵣ v₂ in
    val(ψ₃, v₂)
  | Match(s, bs) →
    let* (ψ₁, v₁) =
      eval γ ψ i′ s in
    let* (c, ps) = getctor v₁ in
    let* (ψ′, v₂) = fold_left*?
      fun r (c′, xs, t) →
        if c <> c′ then r else
        let γ′ = $\gamma\,\overline{[x \mapsto p]}^{xs \cdot ps}$
        in some eval γ′ ψ i′ t
      none bs in val(ψ′, v₂)
```

Fig. 19. Big-step operational semantics.

D.1 Some Lemmas on Subtyping

We start by reviewing a few important lemmas on subtyping.

Theorem D.1 (Swapping). *For all type τ_1, τ_2, and τ_3:*

(1) If $\Xi \vdash \tau_1 \wedge \tau_2 \leq \tau_3$, then $\Xi \vdash \tau_1 \leq \tau_3 \vee \neg\tau_2$.
(2) If $\Xi \vdash \tau_3 \leq \tau_1 \vee \tau_2$, then $\Xi \vdash \tau_3 \wedge \neg\tau_2 \leq \tau_1$.

Proof. See the proof by Parreaux and Chau [68]. □

Theorem D.2 (Duality of Extrema). $\Xi \vdash \top^{\pm} \equiv \neg\bot^{\pm}$.

Proof. See the proof by Parreaux and Chau [68]. □

Theorem D.3 (De Morgan's laws). $\Xi \vdash \neg(\tau_1 \vee^{\pm} \tau_2) \equiv \neg\tau_1 \wedge^{\pm} \neg\tau_2$.

$$\boxed{dnf : \tau \rightarrow \mathbb{D}}$$

$$\begin{aligned}
dnf(\top) = dnf(\neg\bot) &= \top \wedge \neg\bot \\
dnf(\bot) = dnf(\neg\top) &= \bot \\
dnf(v) &= \top \wedge \neg\bot \wedge v \\
dnf(\tau_1 \xrightarrow{\varphi} \tau_2) &= (dnf(\tau_1) \xrightarrow{dnf(\varphi)} dnf(\tau_2)) \wedge \neg\bot \\
dnf(\mathsf{A}[\overline{\textbf{in}\,\tau_i\ \textbf{out}\,\sigma_i}^{i}]) &= \mathsf{A}[\overline{\textbf{in}\,dnf(\tau_i)\ \textbf{out}\,dnf(\sigma_i)}^{i}] \wedge \neg\bot \\
dnf(\tau_1 \wedge \tau_2) &= inter(dnf(\tau_1),\ dnf(\tau_2)) \\
dnf(\tau_1 \vee \tau_2) &= union(dnf(\tau_1),\ dnf(\tau_2)) \\
dnf(\neg v) &= \top \wedge \neg\bot \wedge \neg v \\
dnf(\neg(\tau_1 \xrightarrow{\varphi} \tau_2)) &= \top \wedge \neg(dnf(\tau_1) \xrightarrow{dnf(\varphi)} dnf(\tau_2)) \\
dnf(\neg\mathsf{A}[\overline{\textbf{in}\,\tau_i\ \textbf{out}\,\sigma_i}^{i}]) &= \top \wedge \neg(\bot \vee \mathsf{A}[\overline{\textbf{in}\,dnf(\tau_i)\ \textbf{out}\,dnf(\sigma_i)}^{i}]) \\
dnf(\neg(\tau_1 \wedge \tau_2)) &= union(dnf(\neg\tau_1),\ dnf(\neg\tau_2)) \\
dnf(\neg(\tau_1 \vee \tau_2)) &= inter(dnf(\neg\tau_1),\ dnf(\neg\tau_2))
\end{aligned}$$

$$\boxed{union : (\mathbb{D},\ \mathbb{D}) \rightarrow \mathbb{D}}$$

$$\begin{aligned}
union(\bot,\ \mathbb{D}) = union(\mathbb{D},\ \bot) &= \mathbb{D} \\
union(\mathbb{D},\ \mathbb{C}) &= \begin{cases} \mathbb{D} & \textit{if } \mathbb{C} \in \mathbb{D} \\ \mathbb{D} \vee \mathbb{C} & \textit{otherwise} \end{cases} \\
union(\mathbb{D}_1,\ \mathbb{D}_2 \vee \mathbb{C}) &= union(union(\mathbb{D}_1,\ \mathbb{C}),\ \mathbb{D}_2)
\end{aligned}$$

$$\boxed{inter : (\mathbb{D},\ \mathbb{D}) \rightarrow \mathbb{D}}$$

$$\begin{aligned}
inter(\mathbb{D},\ \bot) = inter(\bot,\ \mathbb{D}) &= \bot \\
inter(\mathbb{D} \vee \mathbb{C}_1,\ \mathbb{C}_2) = inter(\mathbb{C}_2,\ \mathbb{D} \vee \mathbb{C}_1) &= union(inter(\mathbb{D},\ \mathbb{C}_2),\ inter(\mathbb{C}_1,\ \mathbb{C}_2)) \\
inter(\mathbb{D}_1 \vee \mathbb{C},\ \mathbb{D}_2) &= union(inter(\mathbb{D}_1,\ \mathbb{D}_2), inter(\mathbb{C},\ \mathbb{D}_2))
\end{aligned}$$

$$\boxed{inter : (\mathbb{C},\ \mathbb{C}) \rightarrow \mathbb{C}\ \mid\ \bot}$$

$$\begin{aligned}
inter(\mathbb{C}_1,\ \mathbb{C}_2 \wedge \neg^{\pm} v) &= \begin{cases} \bot & \textit{if } \mp v \in \mathbb{C}_1 \\ inter(\mathbb{C}_1,\ \mathbb{C}_2) & \textit{if } \neg^{\pm} v \in \mathbb{C}_1 \\ inter(\mathbb{C}_1 \wedge \neg^{\pm} v,\ \mathbb{C}_2) & \textit{otherwise} \end{cases} \\
inter(I_1 \wedge \neg U_1 \overline{\wedge \neg^{\pm} v},\ I_2 \wedge \neg U_2) &= \begin{cases} \bot & \textit{if } mrg^{+}(I_1,\ I_2) = \bot \\ mrg^{+}(I_1,\ I_2) \wedge \neg mrg^{-}(U_1,\ U_2)\overline{\wedge\neg^{\pm} v} & \textit{otherwise} \end{cases}
\end{aligned}$$

$$\boxed{mrg^{+} : (I,\ I) \rightarrow I\ \mid\ \bot}$$

$$\begin{aligned}
mrg^{+}(I,\ \top) = mrg^{+}(\top,\ I) &= I \\
mrg^{+}(\mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2,\ \mathbb{D}_4 \xrightarrow{\mathbb{D}_6} \mathbb{D}_5) &= union(\mathbb{D}_1,\ \mathbb{D}_4) \xrightarrow{inter(\mathbb{D}_3, \mathbb{D}_6)} inter(\mathbb{D}_2,\ \mathbb{D}_5) \\
mrg^{+}(\mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i1}\ \textbf{out}\,\mathbb{D}_{i2}}^{i}],\ \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i3}\ \textbf{out}\,\mathbb{D}_{i4}}^{i}]) &= \mathsf{A}[\overline{\textbf{in}\ union(\mathbb{D}_{i1},\ \mathbb{D}_{i3})\ \textbf{out}\ inter(\mathbb{D}_{i2},\ \mathbb{D}_{i4})}^{i}] \\
mrg^{+}(I_1,\ I_2) &= \bot \qquad \textit{otherwise}
\end{aligned}$$

$$\boxed{mrg^{-} : (U,\ U) \rightarrow U}$$

$$\begin{aligned}
mrg^{-}(U,\ \bot) = mrg^{-}(\bot,\ U) &= U \\
mrg^{-}(\mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2,\ \mathbb{D}_4 \xrightarrow{\mathbb{D}_6} \mathbb{D}_5) &= inter(\mathbb{D}_1,\ \mathbb{D}_4) \xrightarrow{union(\mathbb{D}_3, \mathbb{D}_6)} union(\mathbb{D}_2,\ \mathbb{D}_5) \\
mrg^{-}(U \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}\ \textbf{out}\,\mathbb{D}}],\ \mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2) &= mrg^{-}(U,\ \mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2) \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}\ \textbf{out}\,\mathbb{D}}] \\
mrg^{-}(U \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i1}\ \textbf{out}\,\mathbb{D}_{i2}}^{i}],\ U' \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i3}\ \textbf{out}\,\mathbb{D}_{i4}}^{i}]) &= mrg^{-}(U,\ U') \vee \mathsf{A}[\overline{\textbf{in}\ inter(\mathbb{D}_{i1},\ \mathbb{D}_{i3})\ \textbf{out}\ union(\mathbb{D}_{i2},\ \mathbb{D}_{i4})}^{i}] \\
mrg^{-}(U,\ U' \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}\ \textbf{out}\,\mathbb{D}}]) &= mrg^{-}(U,\ U') \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}\ \textbf{out}\,\mathbb{D}}] \qquad \textit{if}\, \mathsf{A} \notin U
\end{aligned}$$

Fig. 20. RDNF construction functions.

```
let rec eval (γ: env)
(π: syms) i e: ℝ =
  if i = 0 then kill else
  let i′ = i - 1 in
  match e with
  | ...
  | QAbs(f) ⇒
    let* (π1, v1) =
      eval γ π i′ f in
    let* (x, e, γ′) = getclos v1 in
    let (s, π2) = freshVar π1 in
    let* (π3, v2) =
      eval γ′·[x ↦ s] π2 i′ e
    in val(π3, Abs(s, v2))
  | Run(t) ⇒
    let* (π1, v) =
      eval γ π i′ t
    in eval γ π1 i′ (compile v)
  | Try(t1, t2, t3, t4) ⇒
    let* (π1, v1) =
      eval γ π i′ t1
    in let* (π2, v2) =
      eval γ π1 i′ t2
    in let* b = test π v1 v2 in
    if not b then
      eval γ · [y ↦ v1] π i′ t3
    else eval γ π i′ t4
  | Subst(t1, t2, t3) ⇒
    let* (π1, v1) =
      eval γ π i′ t1
    in let* (π2, v2) =
      eval γ π1 i′ t2
    in let* (π3, v3) =
      eval γ π2 i′ t3
    in let* r = substs π3 v1 v2 v3
    in val(π3, r)
```

Fig. 21. Quasiquotes semantics.

Proof. See the proof by Parreaux and Chau [68]. □

Theorem D.4 (Absorption). $\Xi \vdash \tau_1 \vee^{\pm} (\tau_1 \wedge^{\pm} \tau_2) \equiv \tau_1$.

Proof. See the proof by Parreaux and Chau [68]. □

D.2 Soundness of Subtyping

We now show that subtyping is sound (Theorem D.31 below). We first need a few definitions and lemmas.

Definition D.5 (Function data types). For the sake of simplicity, in the rest of this subsection, we define the function data type $\mathsf{F}[\textbf{in}\,\tau_1,\ \textbf{out}\,\tau_2,\ \textbf{out}\,\varphi] \equiv \tau_1 \xrightarrow{\varphi} \tau_2$, where F is different from all other data types. All subtyping rules in Figure 3 preserve under this new definition.

Definition D.6. The Boolean homomorphism $[\![\cdot]\!]_{\mathbb{E}}$ from the Boolean algebra of types to $P(U)$ orderred by inclusion, where $U = \{\overline{\mathsf{A}}\}$ is the set of all data types, is defined as:

$$[\![\mathsf{A}[\overline{a}]]\!]_{\mathbb{E}} = \mathsf{A}$$

Lemma D.7 (Monotonicity of $[\![\cdot]\!]_{\mathbb{E}}$). *$[\![\cdot]\!]_{\mathbb{E}}$ is monotonic.*

Proof. We show that the subtyping relation is compatible with $[\![\cdot]\!]_{\mathbb{E}}$. The only non-trivial cases are S-CtorBot and S-CFBot. For any two different data types A_1 and A_2 where $\mathsf{A}_1 \neq \mathsf{A}_2$, we have $\{\mathsf{A}_1\} \cap \{\mathsf{A}_2\} = \emptyset$. We conclude by $\emptyset \subseteq \emptyset$. Similarly, we have $\mathsf{F} \neq \mathsf{A}'$ for any other data types by Definition D.5. □

Definition D.8. The Boolean homomorphism $[\![\cdot]\!]^{+i}_{\mathsf{A}}$ from the Boolean algebra of types ordered by subtyping under the context Ξ to the Boolean algebra of types ordered by subtyping under the context $\lhd\Xi$ is defined as:

$$[\![\mathsf{A}'[\overline{a}]]\!]^{+i}_{\mathsf{A}} = \bot \text{ if } \mathsf{A}' \neq \mathsf{A}$$

$$[\![\mathsf{A}[\overline{\textbf{in}\ \tau_j\ \textbf{out}\ \sigma_j}^{j}]]\!]^{+i}_{\mathsf{A}} = \sigma_j \text{ where } i = j$$

Lemma D.9 (Monotonicity of $[\![\cdot]\!]^{+i}_{\mathsf{A}}$). *$[\![\cdot]\!]^{+i}_{\mathsf{A}}$ is monotonic.*

Proof. We show that the subtyping relation is compatible with $[\![\cdot]\!]^{+i}_{\mathsf{A}}$. The only non-trivial cases are S-Fun, S-Ctor, S-CtorBot and S-CFBot.

Case S-Fun. Then $\Xi \vdash \tau_2 \xrightarrow{\tau_5} \tau_3 \leq \tau_1 \xrightarrow{\tau_6} \tau_4$. If $\mathsf{A} \neq \mathsf{F}$, then we conclude by S-Refl. Consider $[\![\cdot]\!]^{+i}_{\mathsf{F}}$. By premises, we have $\lhd\Xi \vdash \tau_5 \leq \tau_6$ and $\lhd\Xi \vdash \tau_3 \leq \tau_4$. For $i = 1$, we have $\lhd\Xi \vdash \top \leq \top$ and we conclude by S-Refl. For $i = 2$ and $i = 3$, we conclude by the premises above.

Case S-Ctor. Similar to the case S-Fun.

Case S-CtorBot. By the premise, $\mathsf{A}_1 \neq \mathsf{A}_2$. Then at least one of $\mathsf{A}_1 \neq \mathsf{A}$ and $\mathsf{A}_2 \neq \mathsf{A}$ holds. Therefore, at least one of $[\![\mathsf{A}_1[\overline{a}]]\!]^{+i}_{\mathsf{A}} = \bot$ and $[\![\mathsf{A}_2[\overline{b}]]\!]^{+i}_{\mathsf{A}} = \bot$ holds. We conclude by S-AndOrL+ or S-AndOrR+.

Case S-CFBot. Similar to the case S-CtorBot.

□

Definition D.10. The Boolean homomorphism $[\![\cdot]\!]^{-i}_{\mathsf{A}}$ from the Boolean algebra of types ordered by subtyping under the context Ξ to the Boolean algebra of types ordered by subtyping under the context $\lhd\Xi$ is defined as:

$$[\![\mathsf{A}'[\overline{a}]]\!]^{-i}_{\mathsf{A}} = \bot \text{ if } \mathsf{A}' \neq \mathsf{A}$$

$$[\![\mathsf{A}[\overline{\textbf{in}\ \tau_j\ \textbf{out}\ \sigma_j}^{j}]]\!]^{-i}_{\mathsf{A}} = \neg\tau_j \text{ where } i = j$$

Lemma D.11 (Monotonicity of $[\![\cdot]\!]^{-i}_{\mathsf{A}}$). *$[\![\cdot]\!]^{-i}_{\mathsf{A}}$ is monotonic.*

Proof. We show that the subtyping relation is compatible with $[\![\cdot]\!]^{-i}_{\mathsf{A}}$. The only non-trivial cases are S-Fun, S-Ctor, S-CtorBot and S-CFBot.

Case S-Fun. Then $\Xi \vdash \tau_2 \xrightarrow{\tau_5} \tau_3 \leq \tau_1 \xrightarrow{\tau_6} \tau_4$. If $\mathsf{A} \neq \mathsf{F}$, then we conclude by S-Refl. Consider $[\![\cdot]\!]^{-i}_{\mathsf{F}}$. By premises, we have $\lhd\Xi \vdash \tau_1 \leq \tau_2$. By Theorem D.1 twice, $\lhd\Xi \vdash \neg\tau_2 \leq \neg\tau_1$. For $i = 1$, we conclude by the premise above. For $i = 2$ and $i = 3$, we have $\lhd\Xi \vdash \top \leq \top$ and we conclude by S-Refl.

Case S-Ctor. Similar to the case S-Fun.

Case S-CtorBot. By the premise, $\mathsf{A}_1 \neq \mathsf{A}_2$. Then at least one of $\mathsf{A}_1 \neq \mathsf{A}$ and $\mathsf{A}_2 \neq \mathsf{A}$ holds. Therefore, at least one of $[\![\mathsf{A}_1[\overline{a}]]\!]^{+i}_{\mathsf{A}} = \bot$ and $[\![\mathsf{A}_2[\overline{b}]]\!]^{+i}_{\mathsf{A}} = \bot$ holds. We conclude by S-AndOrL+ or S-AndOrR+.

Case S-CFBot. Similar to the case S-CtorBot.

□

Lemma D.12. *If $\rhd\Sigma \vdash \tau \leq \sigma$, where*

$$\tau \in \{\top,\ \bot,\ \tau_1 \xrightarrow{\varphi_1} \tau_2,\ \mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \tau_i'}^{i}]\}$$

$$\sigma \in \{\top,\ \bot,\ \sigma_1 \xrightarrow{\varphi_2} \sigma_2,\ \mathsf{A}[\overline{\textbf{in}\ \sigma_i\ \textbf{out}\ \sigma_i'}^{i}]\}$$

then exactly one of the following is true:

- *$\tau = \bot$ or $\sigma = \top$;*
- *$\tau = \tau_1 \xrightarrow{\varphi_1} \tau_2$, $\sigma = \sigma_1 \xrightarrow{\varphi_2} \sigma_2$, $\Sigma \vdash \sigma_1 \leq \tau_1$, $\Sigma \vdash \varphi_1 \leq \varphi_2$, and $\Sigma \vdash \tau_2 \leq \sigma_2$;*

- $\tau = \mathsf{A}_1[\overline{\textbf{in } \tau_i \textbf{ out } \sigma_i}^{i \in 1\ldots m}]$, $\sigma = \mathsf{A}_2[\overline{\textbf{in } \tau'_j \textbf{ out } \sigma'_j}^{j \in 1\ldots n}]$, $\mathsf{A}_1 = \mathsf{A}_2$, $m = n$, $\overline{\Sigma \vdash \tau'_i \leq \tau_i}^i$, *and* $\overline{\Sigma \vdash \sigma_i \leq \sigma'_i}^i$.

Proof. By case analysis on σ.

Case $\sigma = \top$**.** Immediately.

Case $\sigma = \bot$**.** Consider the Boolean homomorphism $[\![\cdot]\!]_{\mathbb{E}}$ in Definition D.6. Then $[\![\sigma]\!]_{\mathbb{E}} = \emptyset$. By case analysis on τ. The only possible case is $\tau = \bot$ and we have $\emptyset \subseteq \emptyset$. For other cases, it is impossible since $\{\mathsf{A}\} \nsubseteq \emptyset$ for all A and $U \nsubseteq \emptyset$.

Case $\sigma = \sigma_1 \xrightarrow{\varphi_2} \sigma_2$**.** First, consider the Boolean homomorphism $[\![\cdot]\!]_{\mathbb{E}}$ in Definition D.6. Then $[\![\sigma]\!]_{\mathbb{E}} = \{\mathsf{F}\}$. By case analysis on τ. The only possible cases are $\tau = \bot$ and $\tau = \tau_1 \xrightarrow{\varphi_1} \tau_2$. For other cases, it is impossible since $\{\mathsf{A}\} \nsubseteq \{\mathsf{F}\}$ for any other A and $U \nsubseteq \{\mathsf{F}\}$. Then consider $\rhd\Sigma \vdash \tau_1 \xrightarrow{\varphi_1} \tau_2 \leq \sigma_1 \xrightarrow{\varphi_2} \sigma_2$. and the Boolean homomorphism $[\![\cdot]\!]^{+i}_{\mathsf{F}}$ in Definition D.8. We have $\Sigma \vdash \tau_2 \leq \sigma_2$ and $\Sigma \vdash \varphi_1 \leq \varphi_2$. Similarly, by the Boolean homomorphism $[\![\cdot]\!]^{-i}_{\mathsf{F}}$ in Definition D.10 and Theorem D.1, we have $\Sigma \vdash \sigma_1 \leq \tau_1$.

Case $\sigma = \mathsf{A}[\overline{\textbf{in } \sigma_i \textbf{ out } \sigma'_i}^i]$**.** Similar to the case $\sigma = \sigma_1 \xrightarrow{\varphi_2} \sigma_2$.

□

Lemma D.13. *$\Xi \vdash \alpha \equiv \alpha \wedge^{\pm} ub^{\pm}_{\Xi}(\alpha)$.*

Proof. We prove the case where $\pm = +$. The proof for $\pm = -$ is symmetric. By S-AndOr+, $\Xi \vdash \alpha \leq ub_{\Xi}(\alpha)$. By S-Refl, $\Xi \vdash \alpha \leq \alpha$. Then by S-AndOr+, $\Xi \vdash \alpha \leq \alpha \wedge^{\pm} ub^{\pm}_{\Xi}(\alpha)$. By S-AndOrL+, $\Xi \vdash \alpha \wedge^{\pm} ub^{\pm}_{\Xi}(\alpha) \leq \alpha$. □

Corollary D.14. *$\Xi \vdash \alpha \equiv \alpha \wedge ub_{\Xi}(\alpha) \vee lb_{\Xi}(\alpha)$.*

Proof. By Lemma D.13. □

Theorem D.15 (Subtyping context weakening). *If $\Xi \vdash \tau \leq \sigma$ and $\Xi' \vDash \Xi$, then $\Xi' \vdash \tau \leq \sigma$.*

Proof. By induction on the subtyping derivations. □

Lemma D.16 (Reflexivity and weakening of subtyping context). *For all Ξ and Ξ', $\Xi\,\Xi' \vDash \Xi$ and $\Xi\,\Xi' \vDash \rhd\Xi$.*

Proof. By repeated applications of S-Cons and S-Cons$\rhd$ on S-Hyp. □

Lemma D.17 (Transitivity). *If $\Xi_1 \vDash \Xi_2$ and $\Xi_2 \vDash \Xi_3$, then $\Xi_1 \vDash \Xi_3$.*

Proof. By induction on the entailment derivations with Theorem D.15. □

Lemma D.18 (Merging). *If $\Xi_1 \vDash \Xi'_1$ and $\Xi_2 \vDash \Xi'_2$, then $\Xi_1\,\Xi_2 \vDash \Xi'_1\,\Xi'_2$.*

Proof. By induction on the entailment derivations with Lemma D.16, Lemma D.17, and Theorem D.15. □

Lemma D.19 (Unguarding). *If $\Xi \vDash \Xi'$, then $\lhd\Xi \vDash \lhd\Xi'$.*

Proof. By induction on the entailment derivations. □

Lemma D.20 (Preservation of subtyping under substitution). *If $\Xi \vdash \tau \leq \sigma$, then $\rho(\Xi) \vdash \rho(\tau) \leq \rho(\sigma)$.*

Proof. By induction on the subtyping derivations. □

Corollary D.21. *If $\Xi \vdash \mathcal{T} \leq^{\forall} \mathcal{S}$, then $\rho(\Xi) \vdash \rho(\mathcal{T}) \leq^{\forall} \rho(\mathcal{S})$.*

Proof. By induction on the general subtyping derivations and Lemma D.20. □

Lemma D.22. *If* $\Sigma \vdash \sigma \equiv \sigma'$ *and* $\alpha \notin TTV(\tau)$, *then* $\rhd\Sigma \vdash [\sigma/\alpha]\tau \equiv [\sigma'/\alpha]\tau$.

Proof. By induction on the syntax of τ.

Case $\tau = \alpha$, $\tau = \omega$. Impossible.

Case $\tau = \tau_1 \xrightarrow{\varphi} \tau_2$. By Lemma D.16 and Theorem D.15, $\lhd\Sigma \vdash \sigma \equiv \sigma'$. By IH, $\lhd\Sigma \vdash [\sigma/\alpha]\tau_1 \equiv [\sigma'/\alpha]\tau_1$, $\lhd\Sigma \vdash [\sigma/\alpha]\tau_2 \equiv [\sigma'/\alpha]\tau_2$, and $\lhd\Sigma \vdash [\sigma/\alpha]\varphi \equiv [\sigma'/\alpha]\varphi$. We conclude by S-Fun.

Case $\tau = \mathrm{A}[\overline{a}]$. Similar to the case $\tau = \tau_1 \xrightarrow{\varphi} \tau_2$.

Case $\tau = \top^{\pm}$. Immediately.

Case $\tau = \tau_1 \vee^{\pm} \tau_2$. We prove the case where $\pm = +$. For $\pm = -$, the proof is symmetric. By IH, $\rhd\Sigma \vdash [\sigma/\alpha]\tau_1 \equiv [\sigma'/\alpha]\tau_1$ and $\rhd\Sigma \vdash [\sigma/\alpha]\tau_2 \equiv [\sigma'/\alpha]\tau_2$. We conclude by S-AndOrL−, S-AndOrR−, and S-AndOr−.

Case $\tau = \neg\tau'$. Then by IH, $\rhd\Sigma \vdash [\sigma/\alpha]\tau' \equiv [\sigma'/\alpha]\tau'$. We conclude by using Theorem D.1 twice.

□

Lemma D.23 (Inlining of consistent bounds). *If* $\Xi \vdash \Sigma$ ***cons.***, *witnessed by some substitution* ρ, *and* $\Xi\,\Sigma \vdash \tau \leq \sigma$, *then* $\rho(\Xi) \rhd\Sigma \vdash \rho(\tau) \leq \rho(\sigma)$.

Proof. By the definition of consistency, $\Xi \vdash \Sigma$ ***cons.*** implies $\rho(\Xi) \rhd\Sigma \models \rho(\Sigma)$ for some substitution ρ. By Lemma D.20, $\rho(\Xi\,\Sigma) \vdash \rho(\tau) \leq \rho(\sigma)$. By Lemma D.16, $\rho(\Xi) \rhd\Sigma \models \rho(\Xi)$. By Lemma D.18, $\rho(\Xi) \rhd\Sigma \models \rho(\Xi\,\Sigma)$. We conclude by Theorem D.15. □

Corollary D.24. *If* Σ ***cons.***, *witnessed by some substitution* ρ, *and* $\Sigma \vdash \tau \leq \sigma$, *then* $\rhd\Sigma \vdash \rho(\tau) \leq \rho(\sigma)$.

Proof. A special case of Lemma D.23. □

Corollary D.25. *If* $\Xi \vdash \Sigma$ ***cons.***, *witnessed by some substitution* ρ, *and* $\Xi\,\Sigma \models \Xi'$, *then* $\rho(\Xi) \rhd\Sigma \models \rho(\Xi')$.

Proof. By induction on the entailment derivations.

Case S-Empty. Immediately.

Case S-Cons. Then $\Xi' = \Xi''\ (\tau \leq \sigma)$. By IH, $\Xi\,\Sigma \models \Xi''$ and $\rho(\Xi) \rhd\Sigma \models \rho(\Xi'')$. By the second premise, $\Xi\,\Sigma \vdash \tau \leq \sigma$. Then by Lemma D.23, $\rho(\Xi) \rhd\Sigma \vdash \rho(\tau) \leq \rho(\sigma)$. We conclude by S-Cons.

Case S-Cons$\rhd$. Then $\Xi' = \Xi'' \rhd(\tau \leq \sigma)$. By IH, $\Xi\,\Sigma \models \Xi''$ and $\rho(\Xi) \rhd\Sigma \models \rho(\Xi'')$. By the second premise, $\lhd(\Xi\,\Sigma) \vdash \tau \leq \sigma$. Since $\Xi \vdash \Sigma$ ***cons.*** implies $\rhd\Sigma\,\rho(\Xi) \models \rho(\Sigma)$ for some substitution ρ, by Lemma D.19, $\Sigma\,\rho(\lhd\Xi) \models \rho(\lhd\Sigma)$, which implies $\lhd\Xi \vdash \lhd\Sigma$ ***cons.***. Then by Lemma D.23, $\lhd(\rho(\Xi) \rhd\Sigma) \vdash \rho(\tau) \leq \rho(\sigma)$. We conclude by S-Cons$\rhd$.

□

Lemma D.26. *If* Σ ***cons.***, *witnessed by some substitution* ρ, $\Sigma \vdash \tau \leq \sigma$, *and* $TTV(\tau) \cup TTV(\sigma) = \emptyset$, *then* $\rhd\Sigma \vdash \tau \leq \sigma$.

Proof. By Corollary D.14, we can let $\rho = [\alpha \wedge ub_{\Sigma}(\alpha) \vee lb_{\Sigma}(\alpha)/\alpha]$ and $\Sigma \vdash \alpha \equiv \rho(\alpha)$. By Lemma D.24, $\rhd\Sigma \vdash \rho(\tau) \leq \rho(\sigma)$. Then by Lemma D.22, $\rhd\Sigma \vdash \tau \leq \rho(\tau)$ and $\rhd\Sigma \vdash \rho(\sigma) \leq \sigma$. We conclude by S-Trans. □

Lemma D.27. *If* $\Xi \vdash \tau \leq \sigma$, *then* $\Xi\,\Sigma \vdash \tau \leq \sigma$.

Proof. By Lemma D.29, $\Xi\,\Sigma \models \Xi$. We conclude by Theorem D.15. □

Corollary D.28. *If* $\Gamma,\ \zeta \vdash t :^{\varphi} \mathcal{T}$, *then* $\Gamma\ \Sigma,\ \zeta \vdash t :^{\varphi} \mathcal{T}$.

Proof. For each T-Subs1 and T-Subs2 in the typing derivations, we conclude by Lemma D.27. □

Lemma D.29 (Preservation of subtyping entailment under substitution). $\Xi \vDash \Xi'$ *implies* $\rho(\Xi) \vDash \rho(\Xi')$.

Proof. By induction on the derivations of $\Xi \vDash \Xi'$ with Corollary D.21. □

Lemma D.30 (Weakening of Subtyping Contexts in Consistent Judgements). *If* $\Xi \vdash \Sigma$ ***cons.*** *and* $\Xi'\ \Sigma \vDash \Xi$, *then* $\Xi' \vdash \Sigma$ ***cons.*** *witnessed by the same substitution* ρ.

Proof. By induction on the size of Σ.

Case $\Sigma = \epsilon$**.** We conclude by Theorem D.15 immediately.

Case $\Sigma = \Sigma'\ (\alpha \leq^{\pm} \tau)$**.** Then $\Xi \vdash \Sigma'\ (\alpha \leq^{\pm} \tau)$ ***cons.*** and $\Xi'\ \Sigma'\ (\alpha \leq^{\pm} \tau) \vDash \Xi$. By IH, $\Xi \vdash \Sigma'$ ***cons.***, $\Xi'\ \Sigma' \vDash \Xi$, and $\Xi' \vdash \Sigma'$ ***cons.***. By the definition of consistency, $\Xi \vdash \Sigma'\ (\alpha \leq^{\pm} \tau)$ ***cons.*** implies $\rhd(\Sigma'\ (\alpha \leq^{\pm} \tau))\ \rho(\Xi) \vDash \rho(\Sigma'\ (\alpha \leq^{\pm} \tau))$ for some substitution ρ. By Corollary D.25, $\rhd\Sigma'\ \rho(\Xi') \vDash \rho(\Xi)$. Then by Lemma D.16, $\rhd(\Sigma'\ (\alpha \leq^{\pm} \tau))\ \rho(\Xi') \vDash \rho(\Xi)$. By Lemma D.16 and Lemma D.18, $\rhd(\Sigma'\ (\alpha \leq^{\pm} \tau))\ \rho(\Xi') \vDash \rho(\Xi)\ \rhd(\Sigma'\ (\alpha \leq^{\pm} \tau))$. By Lemma D.17, we have $\rhd(\Sigma'\ (\alpha \leq^{\pm} \tau))\ \rho(\Xi') \vDash \rho(\Sigma'\ (\alpha \leq^{\pm} \tau))$, which implies $\Xi' \vdash \Sigma$ ***cons.***.

□

Theorem D.31 (Subtyping consistency). *If* Σ ***cons.*** *and* $\Sigma \vdash \tau \leq \sigma$, *where*

$$\tau \in \{\top,\ \bot,\ \tau_1 \xrightarrow{\varphi_1} \tau_2,\ \mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \tau_i'}^{i}]\}$$

$$\sigma \in \{\top,\ \bot,\ \sigma_1 \xrightarrow{\varphi_2} \sigma_2,\ \mathsf{A}[\overline{\textbf{in}\ \sigma_i\ \textbf{out}\ \sigma_i'}^{i}]\}$$

then exactly one of the following is true:

- $\tau = \bot$ *or* $\sigma = \top$;
- $\tau = \tau_1 \xrightarrow{\varphi_1} \tau_2$, $\sigma = \sigma_1 \xrightarrow{\varphi_2} \sigma_2$, $\Sigma \vdash \sigma_1 \leq \tau_1$, $\Sigma \vdash \varphi_1 \leq \varphi_2$, *and* $\Sigma \vdash \tau_2 \leq \sigma_2$;
- $\tau = \mathsf{A}_1[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \sigma_i}^{i\in 1\ldots m}]$, $\sigma = \mathsf{A}_2[\overline{\textbf{in}\ \tau_j'\ \textbf{out}\ \sigma_j'}^{j\in 1\ldots n}]$, $\mathsf{A}_1 = \mathsf{A}_2$, $m = n$, $\overline{\Sigma \vdash \tau_i' \leq \tau_i}^{i}$, *and* $\overline{\Sigma \vdash \sigma_i \leq \sigma_i'}^{i}$.

Proof. By Lemma D.12 and Lemma D.26. □

Lemma D.32 (General Subtyping Consistency). *If* Σ ***cons.*** *and* $\Sigma \vdash \mathcal{T} \leq^{\forall} \mathcal{S}$, *where*

$$\mathcal{T} \in \{\tau^{\not\to},\ \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2,\ \forall V\{\Sigma'\}.\,\mathcal{T}'\}$$

$$\mathcal{S} \in \{\sigma^{\not\to},\ \mathcal{S}_1 \xrightarrow{\varphi} \mathcal{S}_2,\ \forall V\{\Sigma'\}.\,\mathcal{S}'\}$$

then exactly one of the following is true:

- $\mathcal{S} = \top$;
- $\mathcal{T} = \tau^{\not\to}$, $\mathcal{S} = \sigma^{\not\to}$, *and* $\Sigma \vdash \tau \leq \sigma$;
- $\mathcal{T} = \mathcal{T}_1 \xrightarrow{\varphi_1} \mathcal{T}_2$, $\mathcal{S} = \mathcal{S}_1 \xrightarrow{\varphi_2} \mathcal{S}_2$, $\Sigma \vdash \mathcal{S}_1 \leq^{\forall} \mathcal{T}_1$, $\Sigma \vdash \mathcal{T}_2 \leq^{\forall} \mathcal{S}_2$, *and* $\Sigma \vdash \varphi_1 \leq \varphi_2$;
- $\mathcal{T} = \forall V\{\Sigma_1\}.\,\mathcal{T}'$, $\mathcal{S} = \forall V\{\Sigma_2\}.\,\mathcal{S}'$, $\Sigma_1 = \{\overline{\alpha_i \leq^{\pm_i} \tau_i}^{i}\}$, $\Sigma_2 = \{\overline{\alpha_i \leq^{\pm_i} \sigma_i}^{i}\}$, $\overline{\Sigma \vdash \sigma_i \leq^{\pm_i} \tau_i}^{i}$ $\Sigma\ \Sigma_2 \vdash \mathcal{T}' \leq^{\forall} \mathcal{S}'$, *and* $\Sigma \vdash \forall V\{\Sigma_2\}$ ***cons.***.

Proof. For the third case, suppose $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{S}_1$, and $\mathcal{S}_2$ are all monomorphic, we can conclude by Theorem D.31. Otherwise, we conclude by induction on each impossible form of subtyping derivation. For an impossible subtyping form $\Sigma \vdash \mathcal{T} \leq^{\forall} \mathcal{S}$, it can be derived by none of the subtyping rules. □

Lemma D.33. *If Σ **cons.**, $\Sigma \vdash \mathcal{T}_1 \leq^{\forall} \mathcal{S}$, and $\Sigma \vdash \mathcal{S} \leq^{\forall} \mathcal{T}_2$, then $\Sigma \vdash \mathcal{T}_1 \leq^{\forall} \mathcal{T}_2$.*

Proof. By induction on the first general subtyping derivations.

Case S-PTop. We conclude it immediately by Lemma D.32 and Theorem D.31.

Case S-PRefl. Immediately.

Case S-Mono. Then $\mathcal{S} = \sigma$ for some σ and $\mathcal{T}_1 = \tau_1$ for some τ_1. If $\mathcal{T}_2 = \top$ or $\sigma = \top$, it is the same as the case S-PTop. Otherwise, by Lemma D.32, and $\mathcal{T}_2 = \tau_2$ for some τ_2. We conclude by S-Trans.

Case S-Forall. Then $\mathcal{T}_1 = \forall V\{\Sigma_1\}.\mathcal{T}_1'$, $\mathcal{S} = \forall V\{\Sigma'\}.\mathcal{S}'$, $\Sigma_1 = \overline{\alpha_i \leq^{\pm_i} \tau_i}^{i}$, $\Sigma' = \overline{\alpha_i \leq^{\pm_i} \sigma_i}^{i}$, $\Sigma\,\Sigma' \vdash \mathcal{T}_1' \leq^{\forall} \mathcal{S}'$, $\Sigma \vdash V\{\Sigma'\}$ **cons.**, and $\overline{\Sigma \vdash \sigma_i \leq^{\pm_i} \tau_i}^{i}$. By Lemma D.32, $\mathcal{T}_2 = \forall V\{\Sigma_2\}.\mathcal{T}_2'$ for some Σ_2 and $\mathcal{T}_2'$, where $\Sigma_2 = \overline{\alpha_i \leq^{\pm_i} \tau_i'}^{i}$, $\Sigma\,\Sigma_2 \vdash \mathcal{S}' \leq^{\forall} \mathcal{T}_2'$, and $\overline{\Sigma \vdash \tau_i' \leq^{\pm_i} \sigma_i}^{i}$. By S-Trans, $\overline{\Sigma \vdash \tau_i' \leq^{\pm_i} \tau_i}^{i}$. Notice that $\Sigma\,\Sigma_2 \vDash \Sigma'$. Therefore, $\Sigma\,\Sigma_2 \vdash \mathcal{T}_1' \leq^{\forall} \mathcal{S}'$. By IH, $\Sigma\,\Sigma_2 \vdash \mathcal{T}_1' \leq^{\forall} \mathcal{T}_2'$. We conclude by S-Forall.

Case S-PFun. Similarly, we conclude by IH and Lemma D.32.

□

Lemma D.34. *If $\Xi \vdash \tau^+ \leq \sigma^+$ and $\Xi \vdash \sigma^- \leq \tau^-$, then $\Xi \vdash [a/\alpha^{\pm}]\mathcal{T} \leq^{\forall} [b/\alpha^{\pm}]\mathcal{T}$ and $\Xi \vdash [b/\alpha^{\mp}]\mathcal{T} \leq^{\forall} [a/\alpha^{\mp}]\mathcal{T}$, where $a = \textbf{in}\,\tau^-\ \textbf{out}\,\tau^+$ and $b = \textbf{in}\,\sigma^-\ \textbf{out}\,\sigma^+$.*

Proof. By induction on the shape of $\mathcal{T}$. If $\alpha \notin TV(\mathcal{T})$, we conclude by S-Refl and S-PRefl. □

D.3 Auxiliary Lemmas

Lemma D.35. *If $\Gamma \mid \Psi, \zeta \vdash \mathsf{r} : \tau$, then*

(1) $\Psi(\mathsf{r}) = \alpha$ and $sub(\Gamma) \vdash \mathsf{Region}[\textbf{out}\,\alpha] \leq \tau$, or

(2) $\mathsf{r} \notin dom(\Psi)$ and $sub(\Gamma) \vdash \mathsf{Region}[\textbf{out}\,\neg\zeta] \leq \tau$.

Proof. By induction on value typing derivations. It holds immediately by T-RegA for the first case and by T-RegD for the second case. For T-VSubs, it holds by IH and S-Trans. □

Lemma D.36. *If $\Gamma \mid \Psi, \zeta \vdash \ell_{\mathsf{r}} : \tau$, then there exists Γ', Ψ', τ_1, τ_2, and σ', such that $\Psi'(\ell_{\mathsf{r}}) = \tau_1$, $sub(\Gamma') \vdash \tau_1 \leq \tau_2$, $\Gamma \mid \Psi, \zeta \vdash \mathsf{r} : \mathsf{Region}[\textbf{out}\,\sigma']$, and $sub(\Gamma) \vdash \mathsf{Ref}[\tau_2, \textbf{out}\,\sigma'] \leq \tau$.*

Proof. By induction on the location typing judgment. It holds immediately by T-Loc. For T-VSubs, it holds by IH and S-Trans. □

Lemma D.37. *If $\Gamma \mid \Psi, \zeta \vdash \langle \lambda x.\, t, \gamma \rangle : \mathcal{T}$, then there exists Γ', where $\Gamma' \mid \Psi \vDash_{\zeta} \gamma \mid \psi$, $\Gamma'\ (x : \mathcal{T}_1), \zeta \vdash (t : \mathcal{T}_2) : \mathcal{T}_2 \mathbin{!} \varphi$, and $sub(\Gamma) \vdash \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2 \leq^{\forall} \mathcal{T}$.*

Proof. By induction on the closure typing judgment. It holds immediately by T-Clos. For T-VSubs, it holds by IH and Lemma D.33 when $\mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2$ is a higher-ranked polymorphic function type, and it holds by IH, S-Mono, and S-Trans when $\mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2$ is monomorphic. □

Lemma D.38. *If $\Gamma \mid \Psi, \zeta \vdash \mathsf{C}(\overline{v}) : \tau$, then there exist $\overline{a}$, such that $\Gamma \vdash construct_{\mathsf{C}} : \mathcal{T}_1 \to \ldots \to \mathcal{T}_n \to \mathsf{A}[\overline{a}]$, $\overline{\Gamma \mid \Psi, \zeta \vdash v_j : \mathcal{T}_j}^{j}$, and $sub(\Gamma) \vdash \mathsf{A}[\overline{a}] \leq \tau$.*

Proof. By induction on the class typing judgment. It holds immediately by T-Ctor. For T-VSubs, it holds by IH and S-Trans. □

Lemma D.39. *If $\Gamma \mid \Psi, \zeta \vdash v : \forall V\{\Sigma\}.\mathcal{T}$, then there exist $\mathcal{S}$ and Σ', such that $\Gamma \bullet V\,\Sigma' \mid \Psi, \zeta \vee \omega \vdash v : \mathcal{S}$, $\omega \in V$, $sub(\Gamma) \vdash \forall V\{\Sigma'\}$ **cons.**, and $sub(\Gamma) \vdash \forall V\{\Sigma'\}.\mathcal{S} \leq^{\forall} \forall V\{\Sigma\}.\mathcal{T}$.*

Proof. By induction on the forall typing judgment. It holds immediately by T-VGen. For T-VSubs, it holds by IH and Lemma D.33. □

Lemma D.40 (Context weakening). *If* $\Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}$, $\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi$ *for some* ζ, γ, ψ, $\mathbf{val}(\psi', \_) = \mathit{eval}\ \gamma\ \psi\ k\ t$ *for some* k *and* t, $\Gamma \mid \Psi' \vDash_{\zeta} \gamma \mid \psi'$, *and* $\Psi \subseteq \Psi'$, *then* $\Gamma \mid \Psi', \zeta \vdash v : \mathcal{T}$.

Proof. By induction on the value typing derivations. □

Definition D.41. We define $\Psi' = \mathit{clean}_{\Gamma,\zeta}(\Psi)$ to clean up type variables in Ψ, such that $\Gamma \vdash \Psi'$ ***wf***. Formally, $\mathit{clean}_{\Gamma,\zeta}(\Psi) \triangleq \rho(\Psi)$, where $\rho = \overline{[\neg\zeta/\alpha]}^{\alpha \in TV(\Psi), \alpha \notin \Gamma}$. Notice that such type variables can only be introduced by regions, since generalization requires the terms to be pure.

Lemma D.42. *If* $\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta) \mid \Psi,\ \zeta \vee \alpha \vdash v : \tau$, $\mathit{sub}(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash \tau \leq \tau'$, *where* $\Gamma \vdash \tau'$ $\boldsymbol{wf}^{+}$, *and* $\Psi' = \mathit{clean}_{\Gamma,\zeta}(\Psi)$, *then* $\Gamma \mid \Psi',\ \zeta \vdash v : \sigma$ *for some* σ *and* $\mathit{sub}(\Gamma) \vdash \sigma \leq \tau'$.

Proof. By induction on the value typing derivations. □

Lemma D.43. *If* Γ ***wf***, Γ ***cons.***, *and* $\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi$, *then* $\zeta = \bot \overline{\vee v}^{v \in V}$ *for some* V, *where* v *satisfies:*

(1) $v = \omega$ *for some* $\omega \in \Gamma$, *or,*

(2) $v = \alpha$ *for some* $\alpha \in \Gamma$, *where* $lb_{\mathit{sub}(\Gamma)}(\alpha) = \bot$, $ub_{\mathit{sub}(\Gamma)}(\alpha) = \neg\zeta'$, *where* $\zeta' = \bot\overline{\vee\alpha}^{\alpha \in V'}$, *for all* $\alpha' \in V'$, $\boldsymbol{lv}(\alpha', \Gamma) < \boldsymbol{lv}(\alpha, \Gamma)$, *and* $\Psi(\mathrm{r}) = \alpha$ *for some* r,

Proof. By induction on the conformance derivations. □

Lemma D.44. *If* Γ ***wf***, Γ ***cons.***, *and* $\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi$, *then* $\mathit{sub}(\Gamma) \vdash \neg\zeta \leq \zeta$ *is false.*

Proof. By induction on the conformance derivations.

Case CF-Empty. Then $\zeta = \bot$. $\mathit{sub}(\Gamma) \vdash \top \leq \bot$ contradicts Theorem D.31.

Case CF-Var, CF-Loc. By IH.

Case CF-Reg. Then $\zeta = \zeta' \vee \alpha$. By IH, $\mathit{sub}(\Gamma) \vdash \neg\zeta' \leq \zeta'$ is false. If $\mathit{sub}(\Gamma) \vdash \neg\zeta' \wedge \neg\alpha \leq \zeta' \vee \alpha$ holds, then $\mathit{sub}(\Gamma) \vdash \neg\zeta' \leq \zeta' \vee \alpha$ by Theorem D.1. By S-AndOrR−, $\mathit{sub}(\Gamma) \vdash \alpha \leq \zeta' \vee \alpha$. Then by S-AndOr−, $\mathit{sub}(\Gamma) \vdash \neg\zeta' \vee \alpha \leq \zeta' \vee \alpha$. By Theorem D.1, $\mathit{sub}(\Gamma) \vdash \neg\zeta' \leq \zeta' \wedge \neg\alpha$. Then by S-AndOrL+, $\mathit{sub}(\Gamma) \vdash \neg\zeta' \leq \zeta'$, which contradicts $\mathit{sub}(\Gamma) \vdash \neg\zeta' \leq \zeta'$ is false.

Case CF-Forall. Similar to case CF-Reg.

□

D.4 Declarative Soundness

Lemma D.45 (Effect Soundness). *Given* $\mathcal{D}$ ***wf***, Γ ***wf***, $\Gamma \vdash \Psi$ ***wf***, Γ ***cons.***, *and* $\Gamma \mid \Psi \vDash_{\zeta} \gamma \mid \psi$, *if* $\Gamma, \zeta \vdash t : \mathcal{T} \ !\ \varphi$ *and for all* k, $\mathbb{R} = \mathit{eval}\ \gamma\ \psi\ k\ t$ *and* $\mathbb{R} \neq$ **kill**, *then for all region* r *so that* $\Gamma \mid \Psi, \zeta \vdash \mathrm{r} : \mathsf{Region}[\mathbf{out}\,\tau]$, *if there is an operation on* r *in* $\mathit{eval}\ \gamma\ \psi\ k\ t$, *then* $\mathit{sub}(\Gamma) \vdash \tau \leq \varphi$.

Proof. By induction on the evaluation eval $\gamma\ \psi\ k\ t$. For the basic case where $k = 0$, we conclude immediately. For possitive k, by induction on the typing derivations (IH). We only show the interesting cases.

Case T-Region. Then $t =$ **region** x **in** t'. We have, $\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta) \mid \Psi\ (\mathrm{r} : \alpha) \vDash_{\zeta \vee \alpha} \gamma \mid \psi\ (\mathrm{r} \mapsto \epsilon)$ by CF-Reg. Let $\Gamma_1 = \Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)$, $\Psi_1 = \Psi\ (\mathrm{r} : \alpha)$, and $\psi_1 = \psi\ (\mathrm{r} \mapsto \epsilon)$. Then By CF-Var, $\Gamma_1\ (x : \mathsf{Region}[\alpha]) \mid \Psi_1 \vDash_{\zeta \vee \alpha} \gamma\ (x \mapsto \mathrm{r}) \mid \psi_1$. If eval $(\gamma\ (x : \mathsf{Region}[\mathbf{out}\,\alpha]))\ \psi_1\ (k-1)\ t' = \boldsymbol{err}$, then we conclude by IH. Otherwise, we have eval $(\gamma\ (x : \mathsf{Region}[\mathbf{out}\,\alpha]))\ \psi_1\ (k-1)\ t' = \mathbf{val}(\psi_2, v)$. By IH, for all region r' such that $\Gamma_1 \mid \Psi_1, \zeta \vee \alpha \vdash \mathrm{r}' : \mathsf{Region}[\mathbf{out}\,\tau]$, if there is an operation on r', then $\mathit{sub}(\Gamma_1) \vdash \tau \leq \varphi \vee \alpha$. If $\mathrm{r}' = \mathrm{r}$, then all operations on r' are in eval $(\gamma\ (x : \mathsf{Region}[\mathbf{out}\,\alpha]))\ \psi_1\ (k-1)\ t'$. Therefore, there is no operation on r' after $\psi_2 \setminus \mathrm{r}'$.

Then consider $r' \neq r$. By Theorem D.1, $sub(\Gamma_1) \vdash \tau \wedge \neg\alpha \leq \varphi$. Then by S-AndOr+ and S-Trans, $sub(\Gamma_1) \vdash \tau \wedge \zeta \leq \varphi$. If $r' \in dom(\psi_1)$, we have $sub(\Gamma_1) \vdash \tau \leq \zeta$ by CF-Reg. Therefore, $sub(\Gamma_1) \vdash \tau \leq \tau \wedge \zeta$ by S-AndOr+. Then we have $sub(\Gamma) \vdash \tau \leq \varphi$. Otherwise, $sub(\Gamma_1) \vdash \neg\zeta \leq \varphi \vee \alpha$. By S-AndOrR−, $sub(\Gamma_1) \vdash \alpha \leq \varphi \vee \alpha$. By S-AndOr−, $sub(\Gamma_1) \vdash \neg\zeta \vee \alpha \leq \varphi \vee \alpha$. Then by Theorem D.1, $sub(\Gamma_1) \vdash \neg\zeta \leq \varphi \wedge \neg\alpha$. Therefore, $sub(\Gamma_1) \vdash \neg\zeta \leq \varphi$ by S-AndOrL+ and S-Trans. Since $\alpha \notin \zeta$ and $\alpha \notin \varphi$, we have $sub(\Gamma) \vdash \neg\zeta \leq \varphi$.

Case ref t_1 t_2. By inversion on the polymorphic type $\forall\alpha, \beta.\, \text{Region}[\textbf{out}\, \alpha] \to \beta \xrightarrow{\alpha} \text{Ref}[\beta, \textbf{out}\, \alpha]$, we have $\Gamma, \zeta \vdash t_1 : \text{Region}[\textbf{out}\, \sigma] \,!\, \varphi'$ for some σ and φ', and $\Gamma, \zeta \vdash t_2 : \tau \,!\, \varphi'$ for some τ. By inversion of value typing, the evaluation of t_1 yields a region r and the evaluation of t_2 yields some value v. For any region r', we do the case analysis on r'.

Case T-RegA. Then $\Psi(r') = \alpha$ for some α. If $r' = r$, then $sub(\Gamma) \vdash \text{Region}[\textbf{out}\, \alpha] \leq \text{Region}[\textbf{out}\, \sigma]$. By Theorem D.31, $sub(\Gamma) \vdash \alpha \equiv \sigma$. we conclude by S-AndOrR− with $\varphi = \varphi' \vee \sigma$ for r. For any other region $r' \neq r$, we have $sub(\Gamma) \vdash \alpha \leq \varphi'$ by IH. We conclude by S-AndOrL− and S-Trans.

Case T-RegD. Then $r' \notin dom(\Psi)$. If $r' = r$, then $sub(\Gamma) \vdash \text{Region}[\textbf{out}\, \neg\zeta] \leq \text{Region}[\textbf{out}\, \sigma]$. By Theorem D.31, $sub(\Gamma) \vdash \neg\zeta \equiv \sigma$. In this case, getreg will throw an ***err***. We conclude by S-AndOrR− with $\varphi = \varphi' \vee \sigma$ for r. For any other region $r' \neq r$, we have $sub(\Gamma) \vdash \neg\zeta \leq \varphi'$ by IH. We conclude by S-AndOrL− and S-Trans.

Case get t, set t_1 t_2. Similar to the case ref t_1 t_2.

□

Lemma D.46 (Purity). *If $\Gamma, \zeta \vdash t : \mathcal{T} \,!\, \bot$, $\Gamma \mid \Psi \vDash_\zeta \gamma \mid \psi$, and $\textbf{val}(\psi', v) = eval\ \gamma\ \psi\ k\ t$, then $\psi' = \psi$.*

Proof. By Lemma D.45 applied to all regions in Ψ, there are no operations on any region in Ψ. It is easy to see that this implies the new heap ψ' is the same as the old heap ψ. □

Lemma D.47 (General Soundness of the Declarative System). *Given $\mathcal{D}$ **wf**, Γ **wf**, $\Gamma \vdash \Psi$ **wf**, Γ **cons.**, and $\Gamma \mid \Psi \vDash_\zeta \gamma \mid \psi$, if $\Gamma, \zeta \vdash t : \mathcal{T} \,!\, \varphi$ and $sub(\Gamma) \vdash \varphi \leq \zeta$, then for all k, if $\mathbb{R} = eval\ \gamma\ \psi\ k\ t$ and $\mathbb{R} \neq$ **kill**, then $\mathbb{R} = \textbf{val}(\psi', v)$ and $\Gamma \mid \Psi', \zeta \vdash v : \mathcal{T}$ for some Ψ', where $\Gamma \mid \Psi' \vDash_\zeta \gamma \mid \psi'$, $\Gamma \vdash \Psi'$ **wf**, and $\Psi \subseteq \Psi'$.*

Proof. By induction on the evaluation eval $\gamma\ \psi\ k\ t$. For the basic case where $k = 0$, we conclude immediately. For possitive k, by induction on the typing derivations (IH).

Case T-Var. Then $\Gamma(x) = \mathcal{T}$ by the premise. Since $\Gamma \mid \Psi \vDash_\zeta \gamma \mid \psi$, there exists $(x \mapsto v) \in \gamma$, where $\Gamma \mid \Psi, \zeta \vdash v : \mathcal{T}$, by CF-Var. Therefore, $\gamma(x) = v$ and $\mathbb{R} = \textbf{val}(\psi, v)$.

Case T-Abs1. Then $t = \lambda x.\, t'$, and $\mathbb{R} = \textbf{val}(\psi, \langle \lambda x.\, t', \gamma \rangle)$. Since $\Gamma \mid \Psi \vDash_\zeta \gamma \mid \psi$, we conclude by T-Clos. For the effect, since t' is not executed immediately, the abstraction is pure.

Case T-Abs2. Similar to T-Abs1.

Case T-Asc. We conclude by IH.

Case T-App1. Then $t = t_1\ t_2$. By evaluations,

$$\textbf{val}(\psi_1, v_1) = \text{eval}\ \gamma\ \psi\ k-1\ t_1 \tag{1}$$
$$\textbf{val}(\psi_2, v_2) = \text{eval}\ \gamma\ \psi_1\ k-1\ t_2 \tag{2}$$

By IH on (1) and (2),

$$\Gamma \mid \Psi_1 \vDash_\zeta \gamma \mid \psi_1 \tag{3}$$
$$\Gamma \mid \Psi_2 \vDash_\zeta \gamma \mid \psi_2 \tag{4}$$

$$\Gamma \mid \Psi_1, \zeta \vdash \langle \lambda x.\, t', \gamma' \rangle : \tau_1 \xrightarrow{\varphi_1} \tau_2 \tag{5}$$
$$\Gamma \mid \Psi_2, \zeta \vdash v_2 : \tau_1 \tag{6}$$

for some Ψ_1 and Ψ_2, where $v_1 = \langle \lambda x.\, t',\, \gamma' \rangle$ and $\Psi \subseteq \Psi_1 \subseteq \Psi_2$. Notice that v_1 must be closure, or it contradicts the Theorem D.31. By Lemma D.40 on (5),

$$\Gamma \mid \Psi_2,\, \zeta \vdash \langle \lambda x.\, t',\, \gamma' \rangle : \tau_1 \xrightarrow{\varphi_1} \tau_2 \tag{7}$$

By Lemma D.37 on (7),

$$sub(\Gamma) \vdash \sigma_1 \xrightarrow{\varphi_2} \sigma_2 \leq \tau_1 \xrightarrow{\varphi_1} \tau_2 \tag{8}$$
$$\Gamma' \mid \Psi_2 \vDash_\zeta \gamma' \mid \psi_2 \tag{9}$$
$$\Gamma'\ (x : \sigma_1),\, \zeta' \vdash t' : \sigma_2 \mathrel{!} \varphi_2 \tag{10}$$

for some Γ', γ', σ_1, σ_2, and φ_2. By Theorem D.31 on (8),

$$sub(\Gamma) \vdash \tau_1 \leq \sigma_1 \tag{11}$$
$$sub(\Gamma) \vdash \sigma_2 \leq \tau_2 \tag{12}$$
$$sub(\Gamma) \vdash \varphi_2 \leq \varphi_1 \tag{13}$$

By T-VSubs on (6) and (11),

$$\Gamma \mid \Psi_2,\, \zeta \vdash v_1 : \sigma_1 \tag{14}$$

By CF-Var on (14),

$$\Gamma'\ (x : \sigma_1) \mid \Psi_2 \vDash_\zeta \gamma'\ (x \mapsto v_2) \mid \psi_2 \tag{15}$$

By S-Trans, $sub(\Gamma) \vdash \varphi_2 \leq \zeta$. Therefore, $\mathbf{val}(\psi', v) = \text{eval}\ \gamma'\ (x \mapsto v_2)\ \psi_2\ k-1\ t_1$, and $\Gamma \mid \Psi',\, \zeta \vdash v : \sigma_2$ holds by IH for some Ψ', where $\Gamma \mid \Psi' \vDash_\zeta \gamma \mid \psi'$ and $\Psi_2 \subseteq \Psi'$. Then $\Gamma \mid \Psi',\, \zeta \vdash v : \tau_2$ holds by T-VSubs.

Case T-App2. Similar to the case T-App1.

Case T-Subs1, T-Subs2. Immediately by IH and T-VSubs.

Case T-Let. Then $t = \mathbf{let}\, x = t_1\, \mathbf{in}\, t_2$. By the first evaluation, $\mathbf{val}(\psi_1, v_1) = \text{eval}\ \gamma\ \psi\ k-1\ t_1$. By IH, $\Gamma \mid \Psi_1,\, \zeta \vdash v_1 : \mathcal{T}_1$, $\Gamma \mid \Psi_1 \vDash_\zeta \gamma \mid \psi_1$ for some Ψ_1, where $\Psi \subseteq \Psi_1$. Then by CF-Var, $\Gamma\ (x : \mathcal{T}_1) \mid \Psi_1 \vDash_\zeta \gamma\ (x \mapsto v_1) \mid \psi_1$. We conclude by IH and the second evaluation.

Case T-Gen. By premises, Γ ***cons.*** and $sub(\Gamma) \vdash \forall V\{\Sigma\}$ ***cons.***, which implies $\Gamma\ V\ \Sigma$ ***cons.***. Then by CF-Forall, $\Gamma \bullet V\ \Sigma \mid \Psi \vDash_{\zeta \vee \omega} \gamma \mid \psi$, where $\omega \in V$. Therefore, by IH, $\mathbf{val}(\psi', v) = \text{eval}\ \gamma\ \psi\ k - 1\ (t : \mathcal{T})$, $\Gamma \bullet V\ \Sigma \vdash \Psi'$ ***wf***, $\Gamma \bullet V\ \Sigma \mid \Psi',\, \zeta \vee \omega \vdash v : \mathcal{T}$, and $\Gamma \mid \Psi' \vDash_{\zeta \vee \omega} \gamma \mid \psi'$. By Lemma D.46, $\psi' = \psi$, which implies $\Psi' = \Psi$. We conclude by T-VGen.

Case T-Inst. By IH,

$$\mathbf{val}(\psi', v) = \text{eval}\ \gamma\ \psi\ k\ t \tag{16}$$
$$\Gamma \mid \Psi' \vDash_\zeta \gamma \mid \psi' \tag{17}$$
$$\Gamma \mid \Psi',\, \zeta \vdash v : \forall V\{\Sigma\}.\mathcal{T} \tag{18}$$

for some Ψ', where $\Psi \subseteq \Psi'$. By Lemma D.39 on (18),

$$\Gamma\ V\ \Sigma' \mid \Psi',\, \zeta \vee \omega \vdash v : \mathcal{T}' \tag{19}$$
$$sub(\Gamma) \vdash \forall V\{\Sigma'\}\ \boldsymbol{cons.} \tag{20}$$
$$sub(\Gamma) \vdash \forall V\{\Sigma'\}.\,\mathcal{T}' \leq^{\forall} \forall V\{\Sigma\}.\,\mathcal{T} \tag{21}$$

for some Σ' and $\mathcal{T}'$, where $\omega \in V$. Then by Lemma D.32 and T-VGen on (21),

$$\Sigma' = \overline{\alpha_i \leq^{\pm_i} \tau_i}^{i} \tag{22}$$

$$\Sigma = \overline{\alpha_i \leq^{\pm_i} \sigma_i}^{i} \tag{23}$$

$$sub(\Gamma)\ \Sigma \vdash \mathcal{T}' \leq^{\forall} \mathcal{T} \tag{24}$$

$$sub(\Gamma) \vdash \forall V\{\Sigma\}\ \boldsymbol{cons.} \tag{25}$$

$$\overline{sub(\Gamma) \vdash \sigma_i \leq^{\pm_i} \tau_i}^{i} \tag{26}$$

By premise, $sub(\Gamma) \vDash \rho(\Sigma)$. Then by Corollary D.21 and Theorem D.15 on (24),

$$sub(\Gamma) \vdash \rho(\mathcal{T}') \leq^{\forall} \rho(\mathcal{T}) \tag{27}$$

Notice that $sub(\Gamma) \vDash \rho(\Sigma)$, (22), (23) and (26) imply $sub(\Gamma) \vDash \rho(\Sigma')$. By T-VSubs on (19) and (27),

$$\Gamma \mid \Psi', \zeta \vdash v : \rho(\mathcal{T}) \tag{28}$$

Notice that by Lemma D.46, $\rho(\Psi') = \Psi'$.

Case construct$_C$ $\overline{t_i}$. By induction on the number of parameters j with IH and T-Ctor.

Case T-Region. By CF-Reg, $\Gamma_1 \mid \Psi_1 \vDash_{\zeta \vee \alpha} \gamma \mid \psi_1$, where $\Gamma_1 = \Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)$, $\Psi_1 = \Psi\ (\mathsf{r} : \alpha)$, and $\psi_1 = \psi\ (\mathsf{r} \mapsto \epsilon)$. By CF-Var, $\Gamma_1\ (x : \mathrm{Region}[\alpha]) \mid \Psi_1 \vDash_{\zeta \vee \alpha} \gamma\ (x \mapsto \mathsf{r}) \mid \psi_1$. By IH on the evaluation, $\Gamma_1\ (x : \mathrm{Region}[\alpha]) \mid \Psi_2, \zeta \vee \alpha \vdash v : \tau$. for some Ψ_2, where $\Psi_1 \subseteq \Psi_2$ and $\Gamma_1 \mid \Psi_2 \vDash_{\zeta \vee \alpha} \gamma \mid \psi_2$. Let $\psi' = \psi_2 \setminus \mathsf{r}$ and $\Psi' = clean_{\Gamma,\zeta}(\Psi) \setminus \mathsf{r}\overline{\ell_{\mathsf{r}}}$. By Lemma D.42, $\Gamma \mid \Psi' \vDash_{\zeta} \gamma \mid \psi'$, $\Gamma \mid \Psi', \zeta \vdash v : \sigma$, and $sub(\Gamma) \vdash \sigma \leq \tau$. We conclude by T-VSubs.

Case ref $t_1\ t_2$. By inversion on the polymorphic type $\forall \alpha, \beta.\ \mathrm{Region}[\mathbf{out}\,\alpha] \to \beta \xrightarrow{\alpha} \mathrm{Ref}[\beta, \mathbf{out}\,\alpha]$, we have $\Gamma, \zeta \vdash t_1 : \mathrm{Region}[\mathbf{out}\,\sigma]\ !\ \varphi'$ for some σ and φ', and $\Gamma, \zeta \vdash t_2 : \tau\ !\ \varphi'$ for some τ. Then by evaluations,

$$\mathbf{val}(\psi_1, v_1) = \mathrm{eval}\ \gamma\ \psi\ k - 1\ t_1 \tag{29}$$

$$\mathbf{val}(\psi_2, v_2) = \mathrm{eval}\ \gamma\ \psi_1\ k - 1\ t_2 \tag{30}$$

By IH on (29) and (30),

$$\Gamma \mid \Psi_1 \vDash_{\zeta} \gamma \mid \psi_1 \tag{31}$$

$$\Gamma \mid \Psi_2 \vDash_{\zeta} \gamma \mid \psi_2 \tag{32}$$

$$\Gamma \mid \Psi_1, \zeta \vdash v_1 : \mathrm{Region}[\mathbf{out}\,\sigma] \tag{33}$$

$$\Gamma \mid \Psi_2, \zeta \vdash v_2 : \tau \tag{34}$$

for some Ψ_1 and Ψ_2, where

$$\Psi \subseteq \Psi_1 \subseteq \Psi_2 \tag{35}$$

By getreg, $v_1 = \mathsf{r}$ for some r. By Lemma D.40 on (33) and (35),

$$\Gamma \mid \Psi_2, \zeta \vdash \mathsf{r} : \mathrm{Region}[\mathbf{out}\,\sigma] \tag{36}$$

By Lemma D.35 on (36),

$$sub(\Gamma) \vdash \mathrm{Region}[\tau_0] \leq \mathrm{Region}[\sigma] \tag{37}$$

for some τ_0. Then by Lemma D.35 on (33), either $\mathsf{r} \notin dom(\Psi_1)$ or $\Psi_1(\mathsf{r}) = \alpha$ for some α. Notice that we never reintroduce a region, so $\mathsf{r} \notin dom(\Psi_2)$. If the first case holds, then $\Gamma \mid \Psi_2, \zeta \vdash \mathsf{r} : \mathrm{Region}[\mathbf{out}\,\neg\zeta]$. By Theorem D.31, we have $sub(\Gamma) \vdash \neg\zeta \leq \sigma$. Then $sub(\Gamma) \vdash \neg\zeta \leq \varphi$ by Lemma D.45 on the allocation. It contradicts $sub(\Gamma) \vdash \varphi \leq \zeta$ and Lemma D.44. Therefore,

$\Psi_1(\mathsf{r}) = \alpha$ for some α and $(\mathsf{r} \mapsto \mu) \in \psi_1$ for some μ by CF-Reg and CF-Loc. By Theorem D.31 on (37),

$$sub(\Gamma) \vdash \sigma \equiv \alpha \tag{38}$$

Let $\Psi' = \Psi_2\ (\ell_{\mathsf{r}} : \tau)$. Then $\Gamma \mid \Psi', \zeta \vdash \ell_{\mathsf{r}} : \mathsf{Ref}[\tau, \mathbf{out}\,\sigma]$ by T-VSubs and T-Loc, and $\Gamma \mid \Psi' \vDash \gamma \mid \psi'$ by CF-Loc with (38), where $\psi' = \psi_2\ (\mathsf{r} \mapsto (\mu\ (\ell_{\mathsf{r}} \mapsto v)))$.

Case get t. By inversion on the polymorphic type $\forall \alpha, \beta.\, \mathsf{Ref}[\mathbf{out}\,\beta, \mathbf{out}\,\alpha] \xrightarrow{\alpha} \beta$, we have $\Gamma, \zeta \vdash t : \mathsf{Ref}[\mathbf{out}\,\tau, \mathbf{out}\,\sigma] \mathrel{!} \varphi'$ for some τ, σ, and φ'. Then $\mathbf{val}(\psi', v_1) = \mathsf{eval}\ \gamma\ \psi\ k-1\ t$. By IH, $\Gamma \mid \Psi', \zeta \vdash v_1 : \mathsf{Ref}[\mathbf{out}\,\tau, \mathbf{out}\,\sigma]$ and $\Gamma \mid \Psi' \vDash_\zeta \gamma \mid \psi'$ for some Ψ', where $\Psi \subseteq \Psi'$. By getloc, $v_1 = \ell_{\mathsf{r}}$ for some r. By Lemma D.36, $\Psi''(\ell_{\mathsf{r}}) = \tau_1$, $sub(\Gamma') \vdash \tau_1 \leq \tau_2$, $\Gamma \mid \Psi', \zeta \vdash \mathsf{r} : \mathsf{Region}[\mathbf{out}\,\sigma']$, and $sub(\Gamma) \vdash \mathsf{Ref}[\tau_2, \mathbf{out}\,\sigma'] \leq \mathsf{Ref}[\mathbf{out}\,\tau, \mathbf{out}\,\sigma]$ for some Γ', Ψ'', τ_1, τ_2, and σ'. Then by a similar reasoning in ref $t_1\ t_2$, $\Gamma \mid \Psi, \zeta \vdash \mathsf{r} : \mathsf{Region}[\mathbf{out}\,\sigma']$, $sub(\Gamma) \vdash \sigma' \equiv \alpha$ for some α. which implies $\Psi' = \Psi''$, $\Gamma = \Gamma'$, and $\tau_1 = \tau_2$. By CF-Loc, $(\ell_{\mathsf{r}} \mapsto v) \in \mu$, $\psi'(\mathsf{r}) = \mu$ for some μ, and $\Gamma \mid \Psi', \zeta \vdash v : \tau_2$. Therefore $sub(\Gamma) \vdash \tau_2 \leq \tau$ and $sub(\Gamma) \vdash \sigma' \leq \sigma$ by Theorem D.31. Then $\Gamma \mid \Psi', \zeta \vdash v : \tau$ by read and T-VSubs.

Case set $t_1\ t_2$. Similar reasoning to the case of ref $t_1\ t_2$ and deref t.

Case match$_\mathsf{A}$ $t\ \overline{\lambda x_{i1}.\ \ldots \lambda x_{in}.\, t_i}$. By inversion on the polymorphic type, we have $\Gamma, \zeta \vdash t : \mathsf{A}[\overline{a_i}] \mathrel{!} \varphi'$ for some A, a_i, and φ'; and for each pattern branch, we have polymorphic function types $\forall \overline{b}.\, \mathcal{T}_{i1} \to \ldots \to \mathcal{T}_{in} \xrightarrow{\gamma} \beta$. By the first evaluation,

$$\mathbf{val}(\psi_1, v) = \mathsf{eval}\ \gamma\ \psi\ k-1\ t \tag{39}$$

By IH on (39),

$$\Gamma \mid \Psi_1 \vDash_\zeta \gamma \mid \psi_1 \tag{40}$$

$$\Gamma \mid \Psi_1, \zeta \vdash v : \mathsf{A}[\overline{a}] \tag{41}$$

By getctor, $v = \mathsf{C}(\overline{v_j}^{j})$ for one of Cs of A. By Lemma D.38 on (41),

$$\Gamma \vdash \mathsf{construct}_\mathsf{C} : \mathcal{T}_1 \to \cdots \to \mathcal{T}_n \to \mathsf{A}[\overline{a'}] \tag{42}$$

$$\Gamma \mid \Psi, \zeta \vdash \overline{v_j : \mathcal{T}_j}^{j \in 1 \ldots n} \tag{43}$$

$$sub(\Gamma) \vdash \mathsf{A}[\overline{a'}] \leq \mathsf{A}[\overline{a}] \tag{44}$$

for some a's. By Theorem D.31 on (44),

$$sub(\Gamma) \vdash \overline{\alpha_j \leq \gamma_j} \tag{45}$$

$$sub(\Gamma) \vdash \overline{\delta_j \leq \beta_j} \tag{46}$$

where $\overline{a_i = \mathbf{in}\,\alpha_i\ \mathbf{out}\,\beta_i}^{i}$ and $\overline{a'_i = \mathbf{in}\,\gamma_i\ \mathbf{out}\,\delta_i}$. By Lemma D.34 on (42), (45), and (46),

$$\Gamma \vdash \mathsf{construct}_\mathsf{C} : \mathcal{S}_1 \to \cdots \to \mathcal{S}_n \to \mathsf{A}[\overline{a}] \tag{47}$$

$$sub(\Gamma) \vdash \overline{\mathcal{T}_j \leq^\forall \mathcal{S}_j} \tag{48}$$

Therefore, $\Gamma\ \overline{(x_j : \mathcal{S}_j)} \mid \Psi_1 \vDash_\zeta \gamma\ \overline{(x_j \mapsto v_j)} \mid \psi_1$ by CF-Var and T-VSubs. Then we conclude by IH on the corresponding evaluation.

Case pmatch$_\mathsf{A}$ $t\ \overline{\lambda x_{i1}.\ \ldots \lambda x_{in}.\, t_i}$. Similar reasoning to match$_\mathsf{A}$ $t\ \overline{\lambda x_{i1}.\ \ldots \lambda x_{in}.\, t_i}$.

□

Theorem D.48 (Soundness of the Declarative System). *Given $\mathcal{D}$ **wf**, if $\vdash t : \mathcal{T} \mathrel{!} \bot$, for all k, if $\mathbb{R} = \mathsf{eval}\ \epsilon\ \epsilon\ k\ t$ and $\mathbb{R} \neq$ **kill**, then $\mathbb{R} = \mathbf{val}(\epsilon, v)$ and $\vdash v : \mathcal{T}$.*

Proof. By Lemma D.47. □

D.5 Soundness of Type Inference

Lemma D.49 (Soundness of Extrusion). *If* $(\hat{\Sigma}, \tau) \overset{(\pm,m)}{\rightsquigarrow} (\Sigma', \sigma)$, *then* $\hat{\Sigma}\,\Sigma' \vdash \tau \leq^{\pm} \sigma$.

Proof. By induction on the extrusion derivations.

Case X-AndOr◇. Then by IH, $\hat{\Sigma}\,\Sigma_1 \vdash \tau_1 \leq^{\pm} \tau_1'$ and $\hat{\Sigma}\,\Sigma_2 \vdash \tau_2 \leq^{\pm} \tau_2'$. By Theorem D.15, $\hat{\Sigma}\,\Sigma' \vdash \tau_1 \leq^{\pm} \tau_1'$ and $\hat{\Sigma}\,\Sigma' \vdash \tau_2 \leq^{\pm} \tau_2'$, where $\Sigma' = \Sigma_1\,\Sigma_2$. Then we discuss the case when $\pm = +$. The proof is symmetric when $\pm = -$. By case analysis on $\diamond$.

Case $\diamond = +$. By S-Trans, S-AndOrL−, and S-AndOrR−, $\hat{\Sigma}\,\Sigma' \vdash \tau_1 \leq \tau_1' \vee \tau_2'$ and $\hat{\Sigma}\,\Sigma' \vdash \tau_2 \leq \tau_1' \vee \tau_2'$. By S-AndOr−, $\hat{\Sigma}\,\Sigma' \vdash \tau_1 \vee \tau_2 \leq \tau_1' \vee \tau_2'$.

Case $\diamond = -$. By S-Trans, S-AndOrL+, and S-AndOrR+, $\hat{\Sigma}\,\Sigma' \vdash \tau_1 \wedge \tau_2 \leq \tau_1'$ and $\hat{\Sigma}\,\Sigma' \vdash \tau_1 \wedge \tau_2 \leq \tau_2'$. By S-AndOr+, $\hat{\Sigma}\,\Sigma' \vdash \tau_1 \wedge \tau_2 \leq \tau_1' \wedge \tau_2'$.

Case X-Neg. Then by IH, $\hat{\Sigma}\,\Sigma' \vdash \tau \leq^{\mp} \tau'$. By Theorem D.1, $\hat{\Sigma}\,\Sigma' \vdash \tau \wedge \neg\tau' \leq^{\mp} \bot$. By Theorem D.1 again, $\hat{\Sigma}\,\Sigma' \vdash \neg\tau' \leq^{\mp} \neg\tau$. i.e., $\hat{\Sigma}\,\Sigma' \vdash \neg\tau \leq^{\pm} \neg\tau'$.

Case X-Fun, X-Ctor. Similar to the case X-AndOr◇.

Case X-Var. Immediately by S-Hyp.

Case X-Skolem. By S-Hyp and S-AndOr±, $\hat{\Sigma} \vdash \hat{\alpha} \leq^{\pm} ub^{\pm}_{\hat{\Sigma}}(\hat{\alpha})$. By S-Hyp, $\hat{\Sigma}\,\Sigma' \vdash ub^{\pm}_{\hat{\Sigma}}(\hat{\alpha}) \leq^{\pm} \beta^{n}$. We conclude by Lemma D.27 and S-Trans.

Case X-Skip. Immediately by S-Refl.

□

Lemma D.50 (Sufficiency of constraining). *Given subtyping context* Ξ *and polymorphic quantifier bounds* $\hat{\Sigma}$:

(1) If $\Xi, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma$ *and* **err** $\notin \Sigma$, *then* $\Sigma\,\Xi\,\hat{\Sigma} \vdash \tau \leq \sigma$.

(2) If $\Xi, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma$ *and* **err** $\notin \Sigma$, *then* $\Sigma\,\Xi\,\hat{\Sigma} \vdash \mathbb{D} \leq \bot$.

Proof. By induction on the constraining derivations.

Case C-Hyp. Immediately by S-Hyp.

Case C-Assum. Then

$$\Xi \rhd (\tau_1 \leq \tau_2), \hat{\Sigma} \vdash dnf \circ \varrho_{\hat{\Sigma}}(\tau_1 \wedge \neg\tau_2) \Rightarrow \Sigma \tag{1}$$

By IH, Lemma D.13, and Lemma C.4 on (1),

$$\Sigma\,\Xi\,\hat{\Sigma} \rhd (\tau_1 \leq \tau_2) \vdash \tau_1 \wedge \neg\tau_2 \leq \bot \tag{2}$$

By Theorem D.1 on (2),

$$\Sigma\,\Xi\,\hat{\Sigma} \rhd (\tau_1 \leq \tau_2) \vdash \tau_1 \leq \tau_2 \tag{3}$$

We conclude by S-Assum on (3).

Case C-Or. Then $\mathbb{D} = \mathbb{D}' \vee \mathbb{C}$. By IH,

$$\Sigma_0\,\Xi\,\hat{\Sigma} \vdash \mathbb{D}' \leq \bot \tag{4}$$

$$\Sigma'\,\Sigma_0\,\Xi\,\hat{\Sigma} \vdash \mathbb{C} \leq \bot \tag{5}$$

By Lemma D.16,

$$\Sigma'\,\Sigma_0\,\Xi\,\hat{\Sigma} \vDash \Sigma_0\,\Xi\,\hat{\Sigma} \tag{6}$$

By Theorem D.15 on (4) and (6),

$$\Sigma'\,\Sigma_0\,\Xi\,\hat{\Sigma} \vdash \mathbb{D}' \leq \bot \tag{7}$$

We conclude by S-AndOr− on (5) and (7).

Case C-Bot. Immediately by S-Refl.

Case C-NotBot, C-Fun2, C-Fun3. Contradiction.

Case C-Ctor1. Then $\mathbb{D} = \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i1}\,\textbf{out}\,\mathbb{D}_{i3}}^{i}] \wedge \neg(U \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i2}\,\textbf{out}\,\mathbb{D}_{i4}}^{i}])$. By premises

$$\overline{\lhd\Xi\,\overline{\Sigma_j}^{j\in1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i2} \ll \mathbb{D}_{i1} \Rightarrow \Sigma_i}^{i} \tag{8}$$

$$\overline{\lhd\Xi\,\overline{\Sigma_j}^{j}\,\overline{\Sigma'_k}^{k\in1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i3} \ll \mathbb{D}_{i4} \Rightarrow \Sigma'_i}^{i} \tag{9}$$

By IH on (8) and (9),

$$\overline{\lhd\Xi\,\hat{\Sigma}\,\overline{\Sigma_j}^{j\in1\ldots i} \vdash \mathbb{D}_{i2} \le \mathbb{D}_{i1}}^{i} \tag{10}$$

$$\overline{\lhd\Xi\,\hat{\Sigma}\,\overline{\Sigma_j}^{j}\,\overline{\Sigma'_k}^{k\in1\ldots i} \vdash \mathbb{D}_{i3} \le \mathbb{D}_{i4}}^{i} \tag{11}$$

By Lemma D.16 and Theorem D.15 on (10), (11),

$$\overline{\lhd\Xi\,\hat{\Sigma}\,\Sigma \vdash \mathbb{D}_{i2} \le \mathbb{D}_{i1}}^{i} \tag{12}$$

$$\overline{\lhd\Xi\,\hat{\Sigma}\,\Sigma \vdash \mathbb{D}_{i3} \le \mathbb{D}_{i4}}^{i} \tag{13}$$

where $\Sigma = \overline{\Sigma_i}^{i}\,\overline{\Sigma'_i}^{i}$. By S-Ctor on (12) and (13),

$$\Xi\,\Sigma\,\hat{\Sigma} \vdash \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i1}\,\textbf{out}\,\mathbb{D}_{i3}}] \le \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i2}\,\textbf{out}\,\mathbb{D}_{i4}}] \tag{14}$$

By S-AndOrR− and S-Trans on (14),

$$\Xi\,\Sigma\,\hat{\Sigma} \vdash \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i1}\,\textbf{out}\,\mathbb{D}_{i3}}] \le U \vee \mathsf{A}[\overline{\textbf{in}\,\mathbb{D}_{i2}\,\textbf{out}\,\mathbb{D}_{i4}}] \tag{15}$$

We conclude by Theorem D.1 on (15).

Case C-Ctor2. Then $\mathbb{D} = \mathsf{A}_1[\overline{\textbf{in}\,\mathbb{D}_{i1}\,\textbf{out}\,\mathbb{D}_{i3}}^{i}] \wedge \neg(U \vee \mathsf{A}_2[\overline{\textbf{in}\,\mathbb{D}_{j2}\,\textbf{out}\,\mathbb{D}_{j4}}^{j}])$. By IH,

$$\Sigma\,\Xi\,\hat{\Sigma} \vdash \mathsf{A}_1[\overline{\textbf{in}\,\mathbb{D}_{i1}\,\textbf{out}\,\mathbb{D}_{i3}}^{i}] \le U \tag{16}$$

By S-AndOrL− and S-Trans on (16),

$$\Sigma\,\Xi\,\hat{\Sigma} \vdash \mathsf{A}_1[\overline{\textbf{in}\,\mathbb{D}_{i1}\,\textbf{out}\,\mathbb{D}_{i3}}^{i}] \le U \vee \mathsf{A}_2[\overline{\textbf{in}\,\mathbb{D}_{j2}\,\textbf{out}\,\mathbb{D}_{j4}}^{j}] \tag{17}$$

We conclude by Theorem D.1 on (17).

Case C-Ctor3, C-Ctor4. Similar to C-Ctor2.

Case C-Fun1. Then $\mathbb{D} = \mathbb{D}_1 \xrightarrow{\mathbb{D}_5} \mathbb{D}_2 \wedge \neg(\mathbb{D}_3 \xrightarrow{\mathbb{D}_6} \mathbb{D}_4)$. By IH,

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma_0 \vdash \mathbb{D}_3 \le \mathbb{D}_1 \tag{18}$$

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma_0\,\Sigma' \vdash \mathbb{D}_2 \le \mathbb{D}_4 \tag{19}$$

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma_0\,\Sigma'\,\Sigma'' \vdash \mathbb{D}_5 \le \mathbb{D}_6 \tag{20}$$

Let $\Sigma = \Sigma_0\,\Sigma'\,\Sigma''$, by Lemma D.16 and Theorem D.15 on (18) and (19),

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma \vdash \mathbb{D}_3 \le \mathbb{D}_1 \tag{21}$$

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma \vdash \mathbb{D}_2 \le \mathbb{D}_4 \tag{22}$$

By S-Fun on (20), (21), and (22),

$$\Xi\,\hat{\Sigma}\,\Sigma \vdash \mathbb{D}_1 \xrightarrow{\mathbb{D}_5} \mathbb{D}_2 \le \mathbb{D}_3 \xrightarrow{\mathbb{D}_6} \mathbb{D}_4 \tag{23}$$

We conclude by Theorem D.1 on (23).

Case C-Sk. We prove the case where $\mathbb{D} = \mathbb{C} \land \neg\alpha$. For $\mathbb{D} = \mathbb{C} \land \alpha$, the proof is symmetric. By IH on the premise, $\Xi\,\hat{\Sigma}\,\Sigma \vdash \mathbb{C} \leq \bot$. Then by S-Top−, $\Xi\,\hat{\Sigma}\,\Sigma \vdash \bot \leq \alpha$. We conclude by S-Trans and Theorem D.1.

Case C-Var1. Then $\mathbb{D} = \mathbb{C} \land \alpha^m$. By premises, $\boldsymbol{err} \notin \Sigma$. By S-Hyp, $\Sigma\,\Xi\,(\alpha^m \leq \neg\mathbb{C})\,\hat{\Sigma} \vdash \alpha^m \leq \neg\mathbb{C}$. We conclude by Theorem D.1.

Case C-Var2. Similar to C-Var1.

Case C-Var3. Then $\mathbb{D} = \mathbb{C} \land \alpha^m$. By premises,

$$(\hat{\Sigma}, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma', \tau) \tag{24}$$

$$\Xi, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma'' \tag{25}$$

$$\Xi\,\Sigma''\,(\alpha^m \leq \tau), \hat{\Sigma} \vdash \mathsf{lb}_{\Xi}(\alpha^m) \ll \tau \Rightarrow \Sigma_0 \tag{26}$$

By IH on (25),

$$\Xi\,\Sigma''\,\hat{\Sigma} \vDash \Sigma' \tag{27}$$

By Lemma D.49 on (24),

$$\Xi\,\Sigma'\,\hat{\Sigma} \vdash \tau \leq \neg\mathbb{C} \tag{28}$$

By Theorem D.15 on (27) and (28),

$$\Xi\,\Sigma''\,\hat{\Sigma} \vdash \tau \leq \neg\mathbb{C} \tag{29}$$

Therefore, by S-Hyp and S-Trans on (29),

$$\Xi\,\Sigma''\,\hat{\Sigma}\,(\alpha^m \leq \tau) \vdash \alpha^m \leq \neg\mathbb{C} \tag{30}$$

By premises, $\boldsymbol{err} \notin \Sigma$, where $\Sigma = \Sigma_0\,\Sigma''$. By Lemma D.16,

$$\Xi\,\hat{\Sigma}\,\Sigma\,(\alpha^m \leq \tau) \vDash \Xi\,\hat{\Sigma}\,\Sigma''\,(\alpha^m \leq \tau) \tag{31}$$

We conclude by Theorem D.15 on (30) and (31), and then Theorem D.1.

Case C-Var4. Similar to C-Var3.

□

In the rest of paper, we adopt the reformulated constraining rules presented in Figure 22. We always start derivations with an empty Ξ that only maintains constraints, a $\hat{\Sigma}$ containing all skolem bounds, and a Σ containing all general bounds. These rules are equivalent to those defined in Figure 8.

Lemma D.51 (Consistency of constraining). *Given* $\lhd\Xi \vdash \Sigma\,\hat{\Sigma}$ ***cons.***,

(1) If $\Xi, \Sigma, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma'$, *and* $\boldsymbol{err} \notin \Sigma'$, *then* $\lhd\Xi \vdash \Sigma'\,\Sigma\,\hat{\Sigma}$ ***cons.***.

(2) If $\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma'$ *and* $\boldsymbol{err} \notin \Sigma'$, *then* $\lhd\Xi \vdash \Sigma'\,\Sigma\,\hat{\Sigma}$ ***cons.***.

Proof. By induction on the constraining derivations.

Case C-Hyp, C-Bot. Immediately since $\Sigma' = \epsilon$.

Case C-NotBot, C-Fun2, C-Fun3. Contradiction.

Case C-Assum. Then

$$\Xi \rhd (\tau \leq \sigma), \Sigma, \hat{\Sigma} \vdash \mathit{dnf} \circ \varrho_{\hat{\Sigma}}(\tau \land \neg\sigma) \Rightarrow \Sigma' \tag{32}$$

$$\lhd\Xi \vdash \Sigma\,\hat{\Sigma}\ \boldsymbol{cons.} \tag{33}$$

By Lemma D.16,

$$\lhd(\Xi\,\hat{\Sigma} \rhd (\tau \leq \sigma))\,\Sigma \vDash \lhd\Xi \tag{34}$$

$$\boxed{\Xi, \Sigma, \hat{\Sigma} \vdash \Xi \Rightarrow \Sigma} \qquad \textsc{C-Done}\ \frac{}{\Xi, \Sigma, \hat{\Sigma} \vdash \epsilon \Rightarrow \epsilon} \qquad \textsc{C-NotDone}\ \frac{\Xi, \Sigma, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma_1 \qquad \Xi, \Sigma\ \Sigma_1, \hat{\Sigma} \vdash \Xi_0 \Rightarrow \Sigma_2}{\Xi, \Sigma, \hat{\Sigma} \vdash \Xi_0\ (\tau_1 \leq \tau_2) \Rightarrow \Sigma_1\ \Sigma_2}$$

$$\boxed{\Xi, \Sigma, \hat{\Sigma} \vdash \tau \ll \tau \Rightarrow \Sigma} \qquad \textsc{C-Hyp}\ \frac{(\tau_1 \leq \tau_2) \in \Xi\ \Sigma\ \hat{\Sigma}}{\Xi, \Sigma, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \epsilon} \qquad \textsc{C-Assum}\ \frac{\begin{array}{c}(\tau_1 \leq \tau_2) \notin \Xi\ \Sigma\ \hat{\Sigma} \\ \Xi \rhd (\tau_1 \leq \tau_2), \Sigma, \hat{\Sigma} \vdash dnf \circ \rho_{\hat{\Sigma}}(\tau_1 \wedge \neg\tau_2) \Rightarrow \Sigma'\end{array}}{\Xi, \Sigma, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma'}$$

$$\boxed{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma} \qquad \textsc{C-Or}\ \frac{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{D} \Rightarrow \Sigma_1 \qquad \Xi, \Sigma\ \Sigma_1, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma_2}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{D} \vee \mathbb{C} \Rightarrow \Sigma_1\ \Sigma_2} \qquad \textsc{C-NotBot}\ \frac{}{\Xi, \Sigma, \hat{\Sigma} \vdash I \wedge \neg\bot \Rightarrow \boldsymbol{err}} \qquad \textsc{C-Bot}\ \frac{}{\Xi, \Sigma, \hat{\Sigma} \vdash \bot \Rightarrow \epsilon}$$

$$\textsc{C-Ctor1}\ \frac{\overline{\lhd\Xi, \Sigma\ \overline{\Sigma_j}^{j \in 1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i2} \ll \mathbb{D}_{i1} \Rightarrow \Sigma_i}^{i} \qquad \overline{\lhd\Xi, \Sigma\ \overline{\Sigma_j}^{j}\ \overline{\Sigma'_k}^{k \in 1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i3} \ll \mathbb{D}_{i4} \Rightarrow \Sigma'_i}^{i}}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i1}\,\mathbf{out}\,\mathbb{D}_{i3}}] \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i2}\,\mathbf{out}\,\mathbb{D}_{i4}}]) \Rightarrow \overline{\Sigma_i}\ \overline{\Sigma'_i}}$$

$$\textsc{C-Ctor2}\ \frac{\Xi, \Sigma, \hat{\Sigma} \vdash \mathsf{A}_1[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}] \wedge \neg U \Rightarrow \Sigma' \qquad \mathsf{A}_1 \neq \mathsf{A}_2}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathsf{A}_1[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}] \wedge \neg(U \vee \mathsf{A}_2[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}]) \Rightarrow \Sigma'} \qquad \textsc{C-Ctor3}\ \frac{\Xi, \Sigma, \hat{\Sigma} \vdash (\mathbb{D} \xrightarrow{\mathbb{D}} \mathbb{D}) \wedge \neg U \Rightarrow \Sigma'}{\Xi, \Sigma, \hat{\Sigma} \vdash (\mathbb{D} \xrightarrow{\mathbb{D}} \mathbb{D}) \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}]) \Rightarrow \Sigma'}$$

$$\textsc{C-Ctor4}\ \frac{\Xi, \Sigma, \hat{\Sigma} \vdash \top \wedge \neg U \Rightarrow \Sigma'}{\Xi, \Sigma, \hat{\Sigma} \vdash \top \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}]) \Rightarrow \Sigma'} \qquad \textsc{C-Fun1}\ \frac{\begin{array}{c}\lhd\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{D}_3 \ll \mathbb{D}_1 \Rightarrow \Sigma_1 \qquad \lhd\Xi, \Sigma\ \Sigma_1, \hat{\Sigma} \vdash \mathbb{D}_2 \ll \mathbb{D}_4 \Rightarrow \Sigma_2 \\ \lhd\Xi, \Sigma\ \Sigma_1\ \Sigma_2, \hat{\Sigma} \vdash \mathbb{D}_5 \ll \mathbb{D}_6 \Rightarrow \Sigma_3\end{array}}{\Xi, \Sigma, \hat{\Sigma} \vdash (\mathbb{D}_1 \xrightarrow{\mathbb{D}_5} \mathbb{D}_2) \wedge \neg(\mathbb{D}_3 \xrightarrow{\mathbb{D}_6} \mathbb{D}_4) \Rightarrow \Sigma_1\ \Sigma_2\ \Sigma_3}$$

$$\textsc{C-Fun2}\ \frac{}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}\,\mathbf{out}\,\mathbb{D}'}] \wedge \neg(\mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2) \Rightarrow \boldsymbol{err}} \qquad \textsc{C-Fun3}\ \frac{}{\Xi, \Sigma, \hat{\Sigma} \vdash \top \wedge \neg(\mathbb{D}_1 \xrightarrow{\mathbb{D}_3} \mathbb{D}_2) \Rightarrow \boldsymbol{err}} \qquad \textsc{C-Sk}\ \frac{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma'}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{C} \wedge \neg^{\pm}\hat{\alpha} \Rightarrow \Sigma'}$$

$$\textsc{C-Var1}\ \frac{\begin{array}{c}\boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \leq m \\ \Xi, \Sigma\ (\alpha^m \leq \neg\mathbb{C}), \hat{\Sigma} \vdash \mathsf{lb}_{\Sigma}(\alpha^m) \ll \neg\mathbb{C} \Rightarrow \Sigma'\end{array}}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{C} \wedge \alpha^m \Rightarrow \Sigma'\ (\alpha^m \leq \neg\mathbb{C})} \qquad \textsc{C-Var2}\ \frac{\begin{array}{c}\boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \leq m \\ \Xi, \Sigma\ (\mathbb{C} \leq \alpha^m), \hat{\Sigma} \vdash \mathbb{C} \ll \mathsf{ub}_{\Sigma}(\alpha^m) \Rightarrow \Sigma'\end{array}}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{C} \wedge \neg\alpha^m \Rightarrow \Sigma'\ (\mathbb{C} \leq \alpha^m)}$$

$$\textsc{C-Var3}\ \frac{m < \boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \qquad (\hat{\Sigma}, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma', \tau) \qquad \Xi, \Sigma, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma_1 \qquad \Xi, \Sigma\ \Sigma_1\ (\alpha^m \leq \tau), \hat{\Sigma} \vdash \mathsf{lb}_{\Sigma\ \Sigma_1}(\alpha^m) \ll \tau \Rightarrow \Sigma_2}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{C} \wedge \alpha^m \Rightarrow \Sigma_1\ \Sigma_2\ (\alpha^m \leq \tau)}$$

$$\textsc{C-Var4}\ \frac{m < \boldsymbol{lv}(\mathbb{C}, \hat{\Sigma}) \qquad (\hat{\Sigma}, \mathbb{C}) \overset{(+,m)}{\rightsquigarrow} (\Sigma', \tau) \qquad \Xi, \Sigma, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma_1 \qquad \Xi, \Sigma\ \Sigma_1\ (\tau \leq \alpha^m), \hat{\Sigma} \vdash \tau \ll \mathsf{ub}_{\Sigma\ \Sigma_1}(\alpha^m) \Rightarrow \Sigma_2}{\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{C} \wedge \neg\alpha^m \Rightarrow \Sigma_1\ \Sigma_2\ (\tau \leq \alpha^m)}$$

Fig. 22. Reformulated Normal form constraining rules.

By Lemma D.30 on (33) and (34),

$$\lhd(\Xi \rhd (\tau \leq \sigma)) \vdash \Sigma\ \hat{\Sigma}\ \boldsymbol{cons.} \tag{35}$$

By IH on (32) and (35),

$$\lhd\Xi\ (\tau \leq \sigma) \vdash \Sigma\ \Sigma'\ \hat{\Sigma}\ \boldsymbol{cons.} \tag{36}$$

Then by Lemma D.50 on (32),

$$\Xi\,\hat{\Sigma} \rhd (\tau \leq \sigma)\,\Sigma\,\Sigma' \vdash \mathit{dnf} \circ \rho_{\hat{\Sigma}}(\tau \wedge \neg\sigma) \leq \bot \tag{37}$$

By Lemma C.4, Lemma D.13, Theorem D.1, and S-Assum on (37),

$$\Xi\,\hat{\Sigma}\,\Sigma\,\Sigma' \vdash \tau \leq \sigma \tag{38}$$

Therefore, by Lemma D.16 and S-Cons on (38),

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma\,\Sigma' \models \lhd\Xi\,\hat{\Sigma}\,(\tau \leq \sigma) \tag{39}$$

We conclude by Lemma D.30 on (36) and (39).

Case C-Or. Then by premises,

$$\Xi, \Sigma, \hat{\Sigma} \vdash \mathbb{D}' \Rightarrow \Sigma_0 \tag{40}$$

$$\Xi, \Sigma\,\Sigma_0, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma_1 \tag{41}$$

By IH on (40),

$$\lhd\Xi \vdash \Sigma\,\Sigma_0\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{42}$$

By IH on (41) and (42),

$$\lhd\Xi \vdash \Sigma\,\Sigma'\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{43}$$

where $\Sigma' = \Sigma_0\,\Sigma_1$.

Case C-Ctor1, C-Fun1. Similar to case C-Or,

Case C-Ctor2, C-Ctor3, C-Ctor4, C-Sk. Immediately by IH.

Case C-Var1. Then

$$\Xi, \Sigma\,(\alpha^m \leq \neg\mathbb{C}), \hat{\Sigma} \vdash \mathit{lb}_{\Sigma}(\alpha^m) \ll \neg\mathbb{C} \Rightarrow \Sigma_0 \tag{44}$$

$$\lhd\Xi \vdash \Sigma\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{45}$$

By Lemma D.16,

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma\,(\alpha^m \leq \neg\mathbb{C}) \models \lhd\Xi \tag{46}$$

By Lemma D.30 on (45) and (46),

$$\lhd\Xi\,(\alpha^m \leq \neg\mathbb{C}) \vdash \Sigma\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{47}$$

for some substitution ρ. Therefore, by the definition of consistency,

$$\rhd(\Sigma\,\hat{\Sigma})\ \rho(\lhd\Xi\,(\alpha^m \leq \neg\mathbb{C})) \models \rho(\Sigma\,\hat{\Sigma}) \tag{48}$$

By S-Hyp and S-Cons on (48),

$$\rhd(\Sigma\,\hat{\Sigma})\ \rho(\lhd\Xi\,(\alpha^m \leq \neg\mathbb{C})) \models \rho(\Sigma\,\hat{\Sigma}\,(\alpha^m \leq \neg\mathbb{C})) \tag{49}$$

(49) implies,

$$\lhd\Xi\,(\alpha^m \leq \neg\mathbb{C}) \vdash \Sigma\,\hat{\Sigma}\,(\alpha^m \leq \neg\mathbb{C})\ \textbf{\textit{cons.}} \tag{50}$$

By Lemma D.16,

$$\lhd\Xi\,\hat{\Sigma}\,\Sigma\,(\alpha^m \leq \neg\mathbb{C}) \models \lhd\Xi\,(\alpha^m \leq \neg\mathbb{C}) \tag{51}$$

Then by Lemma D.30 on (50) and (51),

$$\lhd\Xi \vdash \Sigma\,\hat{\Sigma}\,(\alpha^m \leq \neg\mathbb{C})\ \textbf{\textit{cons.}} \tag{52}$$

We conclude by IH on (44) and (52).

Case C-Var2. Similar to C-Var1.

Case C-Var3. Then

$$\Xi, \Sigma\,\Sigma''\,(\alpha^m \le \tau), \hat{\Sigma} \vdash lb_{\Sigma}(\alpha^m) \ll \tau \Rightarrow \Sigma_1 \tag{53}$$

$$\lhd\Xi \vdash \Sigma\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{54}$$

where

$$(\hat{\Sigma}, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma_0, \tau) \tag{55}$$

$$\Xi, \Sigma, \hat{\Sigma} \vdash \Sigma_0 \Rightarrow \Sigma'' \tag{56}$$

By IH on (54) and (56),

$$\lhd\Xi \vdash \Sigma\,\Sigma''\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{57}$$

By Lemma D.16,

$$\lhd\Xi\,\Sigma''\,\hat{\Sigma}\,\Sigma\,(\alpha^m \le \tau) \vDash \lhd\Xi \tag{58}$$

By Lemma D.30 on (57) and (58),

$$\lhd\Xi\,(\alpha^m \le \tau) \vdash \Sigma\,\Sigma''\,\hat{\Sigma}\ \textbf{\textit{cons.}} \tag{59}$$

for some substitution ρ. Therefore, by the definition of consistency,

$$\rhd(\Sigma\,\Sigma''\,\hat{\Sigma})\,\rho(\lhd\Xi\,(\alpha^m \le \tau)) \vDash \rho(\Sigma\,\Sigma''\,\hat{\Sigma}) \tag{60}$$

By S-Refl and S-Cons on (60),

$$\rhd(\Sigma\,\Sigma''\,\hat{\Sigma})\,\rho(\lhd\Xi\,(\alpha^m \le \tau)) \vDash \rho(\Sigma\,\Sigma''\,\hat{\Sigma}\,(\alpha^m \le \tau)) \tag{61}$$

By Lemma D.18 on (61) and $\rhd(\{\alpha^m \le \tau\}) \vDash \epsilon$,

$$\lhd\Xi\,(\alpha^m \le \tau) \vdash \Sigma\,\Sigma''\,\hat{\Sigma}\,(\alpha^m \le \tau)\ \textbf{\textit{cons.}} \tag{62}$$

By Lemma D.16,

$$\lhd\Xi\,\Sigma''\,\hat{\Sigma}\,\Sigma\,(\alpha^m \le \tau) \vDash \lhd\Xi\,(\alpha^m \le \tau) \tag{63}$$

Then by Lemma D.30 on (62) and (63),

$$\lhd\Xi \vdash \Sigma\,\Sigma''\,\hat{\Sigma}\,(\alpha^m \le \tau)\ \textbf{\textit{cons.}} \tag{64}$$

We conclude by IH on (53) and (64).

Case C-Var4. Similar to C-Var3.

□

Lemma D.52 (Soundness of constraining). *If $\Sigma\,\hat{\Sigma}$ **cons.**, $\Sigma, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma'$, and **err** $\notin \Sigma'$, then $\Sigma\,\hat{\Sigma}\,\Sigma'$ **cons.** and $\Sigma\,\hat{\Sigma}\,\Sigma' \vdash \tau \le \sigma$.*

Proof. By Lemma D.50 and Lemma D.51. □

Lemma D.53 (General soundness of type inference). *Given definitions $\mathcal{D}$ **wf** and $\Gamma\,\Sigma_0$ **cons.**, if $\Gamma, \zeta \vdash t : \mathcal{T} \,!\, \varphi \Rightarrow \Xi$, $\epsilon, \Sigma_0, sub(\Gamma) \vdash \Xi \Rightarrow \Sigma$, and **err** $\notin \Sigma$, then $\Gamma\,\Sigma_0\,\Sigma, \zeta \vdash t : \mathcal{T} \,!\, \varphi$, and $\Gamma\,\Sigma\,\Sigma_0$ **cons.**.*

Proof. By induction on the type inference derivations.

Case I-Var. Since $\Xi = \epsilon$, we conclude by T-Var.

Case I-Abs1. Then $t = \lambda x.\, t'$. By premises, $\Gamma\,(x : \alpha^n), \zeta \vdash t' : \mathcal{T}' \,!\, \varphi' \Rightarrow \Xi$. Since $sub(\Gamma) = sub(\Gamma\,(x : \alpha^n))$, $\Gamma\,\Sigma_0\,(x : \alpha^n)\,\Sigma, \zeta \vdash t' : \mathcal{T}' \,!\, \varphi'$ and $\Gamma\,\Sigma_0\,\Sigma$ **cons.** by IH. Then by T-Abs1, $\Gamma\,\Sigma_0\,\Sigma, \zeta \vdash t : \alpha^n \xrightarrow{\varphi'} \mathcal{T}' \,!\, \bot$.

Case I-Abs2. Similar to thecase I-Abs1. For the effect constraint $\varphi' \leq \varphi$, we can conclude by Lemma D.52.

Case I-Let. Then $t = \textbf{let}\, x \;=\; t_1\; \textbf{in}\, t_2$. By premises and IH,

$$\Gamma, \zeta \vdash t_1 : \mathcal{T}_1 \,!\, \varphi_1 \Rightarrow \Xi_1 \tag{65}$$
$$\epsilon, \Sigma_0, sub(\Gamma) \vdash \Xi_1 \Rightarrow \Sigma_1 \tag{66}$$
$$\Gamma\, \Sigma_0\, \Sigma_1, \zeta \vdash t_1 : \mathcal{T}_1 \,!\, \varphi_1 \tag{67}$$
$$\Gamma\, \Sigma_0\, \Sigma_1\; \textbf{\textit{cons.}} \tag{68}$$

Then by the definition of Γ consistency and (68),

$$\Gamma\, \Sigma_0\, \Sigma_1\, (x : \mathcal{T}_1)\; \textbf{\textit{cons.}} \tag{69}$$

Therefore, by IH on the premises and (69),

$$\Gamma\, (x : \mathcal{T}_1), \zeta \vdash t_2 :^{\varphi_2} \mathcal{T}_2 \Rightarrow \Xi_2 \tag{70}$$
$$\epsilon, \Sigma_0\, \Sigma_1, sub(\Gamma) \vdash \Xi_2 \Rightarrow \Sigma_2 \tag{71}$$
$$\Gamma\, (x : \mathcal{T}_1)\, \Sigma', \zeta \vdash t_2 : \mathcal{T}_2 \,!\, \varphi_2 \tag{72}$$
$$\Gamma\, (x : \mathcal{T}_1)\, \Sigma'\; \textbf{\textit{cons.}} \tag{73}$$

where $\Sigma' = \Sigma_1\, \Sigma_2$. By Corollary D.28 on (67),

$$\Gamma\, \Sigma', \zeta \vdash t_1 : \mathcal{T}_1 \,!\, \varphi_1 \tag{74}$$

By T-Subs2 on (72) and (74),

$$\Gamma\, (x : \mathcal{T}_1)\, \Sigma', \zeta \vdash t_2 : \mathcal{T}_2 \,!\, \varphi \tag{75}$$
$$\Gamma\, \Sigma', \zeta \vdash t_1 : \mathcal{T}_1 \,!\, \varphi \tag{76}$$

where $\varphi = \varphi_1 \vee \varphi_2$. We conclude by T-Let on (75) and (76).

Case I-Asc1, I-Asc2. Then $t = t' : \tau$. By premises and IH,

$$\Gamma, \zeta \vdash t' : \sigma \,!\, \varphi \Rightarrow \Xi' \tag{77}$$
$$\epsilon, \Sigma_0, sub(\Gamma) \vdash \Xi' \Rightarrow \Sigma' \tag{78}$$
$$\Gamma\, \Sigma_0\, \Sigma', \zeta \vdash t' : \sigma \,!\, \varphi \tag{79}$$
$$\Gamma\, \Sigma_0\, \Sigma'\; \textbf{\textit{cons.}} \tag{80}$$

Let,

$$\epsilon, \Sigma_0\, \Sigma', sub(\Gamma) \vdash \sigma \leq \tau \Rightarrow \Sigma'' \tag{81}$$

By Lemma D.52 on (80) and (81),

$$\Gamma\, \Sigma\; \textbf{\textit{cons.}} \tag{82}$$
$$sub(\Gamma)\, \Sigma \vdash \sigma \leq \tau \tag{83}$$

where $\Sigma = \Sigma'\, \Sigma_0\, \Sigma''$. By Corollary D.28 on (79),

$$\Gamma\, \Sigma, \zeta \vdash t' : \sigma \,!\, \varphi \tag{84}$$

We conclude by T-Subs1 on (84).

Case I-App1. Then $t = t_1\ t_2$. By premises and IH,

$$\Gamma,\ \zeta \vdash t_1 : \tau_1\ !\ \varphi_1 \Rightarrow \Xi_1 \tag{85}$$
$$\epsilon, \Sigma_0, sub(\Gamma) \vdash \Xi_1 \Rightarrow \Sigma_1 \tag{86}$$
$$\Gamma\ \Sigma_0\ \Sigma_1,\ \zeta \vdash t_1 : \tau_1\ !\ \varphi_1 \tag{87}$$
$$\Gamma\ \Sigma_0\ \Sigma_1\ \boldsymbol{cons.} \tag{88}$$

Therefore, by IH on premises and (88),

$$\Gamma,\ \zeta \vdash t_2 : \tau_2\ !\ \varphi_2 \Rightarrow \Xi_2 \tag{89}$$
$$\epsilon, \Sigma_0\ \Sigma_1, sub(\Gamma) \vdash \Xi_2 \Rightarrow \Sigma_2 \tag{90}$$
$$\Gamma\ \Sigma_0\ \Sigma_1\ \Sigma_2,\ \zeta \vdash t_2 : \tau_2\ !\ \varphi_2 \tag{91}$$
$$\Gamma\ \Sigma_0\ \Sigma_1\ \Sigma_2\ \boldsymbol{cons.} \tag{92}$$

Assume that

$$sub(\Gamma)\ \Sigma_0\ \Sigma_1\ \Sigma_2 \vdash \tau_1 \ll \tau_2 \xrightarrow{\gamma^n} \beta^n \Rightarrow \Sigma_3 \tag{93}$$

where $n = \boldsymbol{lv}(\Gamma)$. Let $\Xi = \Xi_1\ \Xi_2\ (\tau_1 \leq \tau_2 \xrightarrow{\gamma^n} \beta^n)$ and $\Sigma = \Sigma_1\ \Sigma_2\ \Sigma_3\ \Sigma_0$. Since $\boldsymbol{err} \notin \Sigma$, by Lemma D.52 on (93),

$$\Gamma\ \Sigma\ \boldsymbol{cons.} \tag{94}$$

$$sub(\Gamma)\ \Sigma \vdash \tau_1 \leq \tau_2 \xrightarrow{\gamma^n} \beta^n \tag{95}$$

By Corollary D.28 on (87), (91),

$$\Gamma\ \Sigma,\ \zeta \vdash t_1 : \tau_1\ !\ \varphi_1 \tag{96}$$
$$\Gamma\ \Sigma,\ \zeta \vdash t_2 : \tau_2\ !\ \varphi_2 \tag{97}$$

By T-Subs1 and T-Subs2 on (95), (96), and (97),

$$\Gamma\ \Sigma,\ \zeta \vdash t_1 : \tau_2 \xrightarrow{\gamma^n} \beta^n\ !\ \varphi \tag{98}$$
$$\Gamma\ \Sigma,\ \zeta \vdash t_2 : \tau_2\ !\ \varphi \tag{99}$$

where $\varphi = \gamma^n \vee \varphi_1 \vee \varphi_2$. We conclude by T-App.

Case I-App2. Similar to the case I-App1 but no additional constraints are required.

Case I-Gen. Then $t = t' : \forall V\{\Sigma_1\}.\mathcal{S}$. By premises,

$$\epsilon, \epsilon, sub(\Gamma) \vdash \Sigma_1 \Rightarrow \Sigma_2 \tag{100}$$
$$\Gamma \bullet V\ \Sigma_1,\ \zeta \vee \omega \vdash (t' : \mathcal{S}) : \mathcal{S}\ !\ \varphi' \Rightarrow \Xi' \tag{101}$$
$$\epsilon, \epsilon, sub(\Gamma \bullet V\ \Sigma_1) \vdash \Xi'\ (\varphi' \leq \bot) \Rightarrow \Sigma' \tag{102}$$

where $\omega \in V$. Since $\boldsymbol{err} \notin \Sigma_2$, by Lemma D.52 on Γ $\boldsymbol{cons.}$ and (100),

$$\Gamma \bullet V\ \Sigma_1\ \boldsymbol{cons.} \tag{103}$$

Notice that ρ will not break the consistency. By IH on (101), (102), and (103),

$$\Gamma\ \Sigma' \bullet V\ \Sigma_1,\ \zeta \vee \omega \vdash (t' : \mathcal{S}) : \mathcal{S}\ !\ \varphi' \tag{104}$$
$$\Gamma\ \Sigma' \bullet V\ \Sigma_1\ \boldsymbol{cons.} \tag{105}$$

By Lemma D.50 on (102),

$$sub(\Gamma \bullet V\ \Sigma_1)\ \Sigma' \vdash \varphi' \leq \bot \tag{106}$$

By T-Subs2 on (104) and (106),

$$\Gamma\,\Sigma' \bullet V\,\Sigma_1,\ \zeta \vee \omega \vdash (t : \mathcal{S}) : \mathcal{S} \mathbin{!} \bot \tag{107}$$

Assume that

$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Sigma' \Rightarrow \Sigma \tag{108}$$

Since $\boldsymbol{err} \notin \Sigma$, by Lemma D.52 on (108),

$$\Gamma\,\Sigma_0\,\Sigma\ \boldsymbol{cons.} \tag{109}$$

We conclude by T-Gen on (103) and (107).

Case I-Inst. By premises,

$$\Gamma,\ \zeta \vdash t : \forall V\{\Sigma'\}.\,\mathcal{S} \mathbin{!} \varphi \Rightarrow \Xi \tag{110}$$

$$\overline{\beta^n_\alpha\ \boldsymbol{fresh}}^{\alpha^n \in V \setminus \omega} \tag{111}$$

$$\rho = [\overline{\beta^n_\alpha / \alpha^n}^{\alpha^n \in V \setminus \omega}]\,[\zeta/\omega] \tag{112}$$

where $n = \boldsymbol{lv}(\Gamma)$ and $\omega \in V$. By IH on (110),

$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Xi \Rightarrow \Sigma \tag{113}$$

$$\Gamma\,\Sigma_0\,\Sigma,\ \zeta \vdash t : \forall V\{\Sigma'\}.\,\mathcal{S} \mathbin{!} \varphi \tag{114}$$

$$\Gamma\,\Sigma_0\,\Sigma\ \boldsymbol{cons.} \tag{115}$$

Assume that

$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \rho(\Sigma') \Rightarrow \Sigma'' \tag{116}$$

Then by Lemma D.50 on (116),

$$\mathit{sub}(\Gamma)\,\Sigma_0\,\Sigma'' \vDash \rho(\Sigma') \tag{117}$$

By Lemma D.18 on (117) and $\Sigma \vDash \epsilon$,

$$\mathit{sub}(\Gamma)\,\Sigma_0\,\Sigma\,\Sigma'' \vDash \rho(\Sigma') \tag{118}$$

We conclude by T-Inst on (112), (114), and (118).

Case I-Region. Then $t = \textbf{region}\,x\,\textbf{in}\,t'$. Since α is fresh, by premises and IH,

$$\Gamma \bullet \alpha\ (x : \alpha)\ (\alpha \leq \neg\zeta),\ \zeta \vee \alpha \vdash t' : \tau \mathbin{!} \varphi' \Rightarrow \Xi' \tag{119}$$

$$\epsilon, \epsilon, \mathit{sub}(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash \Xi' \Rightarrow \Sigma' \tag{120}$$

$$\Gamma\,\Sigma' \bullet \alpha\ (x : \alpha)\ (\alpha \leq \neg\zeta),\ \zeta \vee \alpha \vdash t' : \tau \mathbin{!} \varphi' \tag{121}$$

$$\Gamma\,\Sigma'\ \boldsymbol{cons.} \tag{122}$$

Assume that

$$\epsilon, \epsilon, \mathit{sub}(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash (\varphi' \leq \gamma^n \vee \alpha)\ (\tau \leq \beta^n) \Rightarrow \Sigma'' \tag{123}$$

Then by Lemma D.50 on (122) and (123),

$$\mathit{sub}(\Gamma)\ (\alpha \leq \neg\zeta)\ \Sigma'\,\Sigma'' \vdash \varphi' \leq \gamma \vee \alpha \tag{124}$$

$$\mathit{sub}(\Gamma)\ (\alpha \leq \neg\zeta)\ \Sigma'\,\Sigma'' \vdash \tau \leq \beta \tag{125}$$

By Corollary D.28, T-Subs1, and T-Subs2 on (121), (124), and (125),

$$\Gamma\,\Sigma'\,\Sigma'' \bullet \alpha\ (x : \alpha)\ (\alpha \leq \neg\zeta),\ \zeta \vee \alpha \vdash t : \beta \mathbin{!} \gamma \vee \alpha \tag{126}$$

Assume that

$$\epsilon, \Sigma_0, sub(\Gamma) \vdash \Sigma' \, \Sigma'' \Rightarrow \Sigma \tag{127}$$

By Lemma D.50 on (127),

$$sub(\Gamma) \, \Sigma_0 \, \Sigma \vDash \Sigma' \, \Sigma'' \tag{128}$$

By Theorem D.15 on (128) and (126),

$$\Gamma \, \Sigma_0 \, \Sigma \bullet \alpha \; (x : \alpha) \; (\alpha \leq \neg\zeta), \; \zeta \vee \alpha \vdash t : \beta \; ! \; \gamma \vee \alpha \tag{129}$$

We conclude by T-Region on (129).

□

Theorem D.54 (Soundness of type inference). *Given definitions $\mathcal{D}$ **wf**, if $\vdash t : \mathcal{T} \Rightarrow \Xi$, $\vdash \Xi \Rightarrow \Sigma$, and **err** $\notin \Sigma$, then $\Sigma \vdash t : \mathcal{T}$ and Σ **cons**.*.

Proof. A special case of Lemma D.53. □

D.6 Termination of Constraining Algorithm

Theorem D.55 (Constraining termination). *For all $\mathcal{D}$ **wf**, $\Sigma \, \hat{\Sigma}$ **wf**, τ, and σ, $\epsilon, \Sigma, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma'$ for some Σ'.*

Proof. We define T_i as the set of type pairs constrained at recursive depth i of the constraining algorithm with the subtyping hypotheses. If we start from $\epsilon, \Sigma, \hat{\Sigma} \vdash \tau \ll \sigma \Rightarrow \Sigma'$, then $T_0 = \{\tau \leq \sigma\} \cup \Sigma \cup \hat{\Sigma}$. We define T_i as follows:

$$
\begin{aligned}
T_0 &= \{\tau \le \sigma\} \cup \Sigma \cup \hat{\Sigma} \\
T_{i+1} &= \overline{\{\mathbb{D}_{j2} \le \mathbb{D}_{j1} \;\mid\; \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{j1}\ \mathbf{out}\,\mathbb{D}_{j3}}^{j}] \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{j2}\ \mathbf{out}\,\mathbb{D}_{j4}}^{j}]) \in S_i\}}^{j} \\
&\quad \cup \overline{\{\mathbb{D}_{j3} \le \mathbb{D}_{j4} \;\mid\; \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{j1}\ \mathbf{out}\,\mathbb{D}_{j3}}^{j}] \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{j2}\ \mathbf{out}\,\mathbb{D}_{j4}}^{j}]) \in S_i\}}^{j} \\
&\quad \cup \{\mathbb{D}_2 \le \mathbb{D}_1 \;\mid\; (\mathbb{D}_1 \overset{\mathbb{D}_5}{\rightarrow} \mathbb{D}_3) \wedge \neg(\mathbb{D}_2 \overset{\mathbb{D}_6}{\rightarrow} \mathbb{D}_4) \in S_i\} \\
&\quad \cup \{\mathbb{D}_3 \le \mathbb{D}_4 \;\mid\; (\mathbb{D}_1 \overset{\mathbb{D}_5}{\rightarrow} \mathbb{D}_3) \wedge \neg(\mathbb{D}_2 \overset{\mathbb{D}_6}{\rightarrow} \mathbb{D}_4) \in S_i\} \\
&\quad \cup \{\mathbb{D}_5 \le \mathbb{D}_6 \;\mid\; (\mathbb{D}_1 \overset{\mathbb{D}_5}{\rightarrow} \mathbb{D}_3) \wedge \neg(\mathbb{D}_2 \overset{\mathbb{D}_6}{\rightarrow} \mathbb{D}_4) \in S_i\} \\
&\quad \cup \{\bigvee_{\tau \in S} \tau \le \neg\mathbb{C} \;\mid\; \mathbb{C} \wedge \alpha^m \in S_i, S = P(\{\sigma \;\mid\; (\sigma \le \alpha^m) \in \bigcup_{j \le i} T_j\}), \boldsymbol{lv}(\hat{\Sigma}, \mathbb{C}) \le m\} \\
&\quad \cup \{\bigvee_{\tau \in S} \tau \le \neg\tau', \overline{\beta_j \le \gamma_j}^{j} \;\mid\; \mathbb{C} \wedge \alpha^m \in S_i, S = P(\{\sigma \;\mid\; (\sigma \le \alpha^m) \in \bigcup_{j \le i} T_j\}), \\
&\qquad\quad m < \boldsymbol{lv}(\hat{\Sigma}, \mathbb{C}), (\bigcup_{j \le i} T_j, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma, \tau'), \overline{\beta_j \le \gamma_j \in \Sigma}^{j}\} \\
&\quad \cup \{\mathbb{C} \le \bigwedge_{\tau \in S} \tau \;\mid\; \mathbb{C} \wedge \neg\alpha^m \in S_i, S = P(\{\sigma \;\mid\; (\alpha^m \le \sigma) \in \bigcup_{j \le i} T_j\}), \boldsymbol{lv}(\hat{\Sigma}, \mathbb{C}) \le m\} \\
&\quad \cup \{\tau' \le \bigwedge_{\tau \in S} \tau, \overline{\beta_j \le \gamma_j}^{j} \;\mid\; \mathbb{C} \wedge \tau \in S_i, S = P(\{\sigma \;\mid\; (\alpha^m \le \sigma) \in \bigcup_{j \le i} T_j\}), \\
&\qquad\quad m < \boldsymbol{lv}(\hat{\Sigma}, \mathbb{C}), (\bigcup_{j \le i} T_j, \mathbb{C}) \overset{(+,m)}{\rightsquigarrow} (\Sigma, \tau'), \overline{\beta_j \le \gamma_j \in \Sigma}^{j}\} \\
&\quad \cup \{\alpha^m \le \neg\mathbb{C} \;\mid\; \alpha^m \wedge \mathbb{C} \in S_i\} \\
&\quad \cup \{\mathbb{C} \le \alpha^m \;\mid\; \mathbb{C} \wedge \neg\alpha^m \in S_i\} \\
S_i &= \{\mathbb{C} \;\mid\; (\tau \le \sigma) \in T_i, \mathit{dnf} \circ \varrho_{\hat{\Sigma}}(\tau \wedge \neg\sigma) = \bigvee_j \mathbb{C}_j, \mathbb{C} \in \{\overline{\mathbb{C}_j}^{j}\}\}
\end{aligned}
$$

In the above definition, S_i invokes the *dnf* function and skolem bound inlining function to translate the given constraint into RNDF, which is done by C-Assum. For T_{i+1}, the first five components are provided by C-Ctor1 and C-Fun1. The remaining components contain the premises from C-Var1, C-Var2, C-Var3, C-Var4, and C-Hyp respectively. Note that some hypotheses created by C-Assum may not appear as a bound of a type variable at the end. We simply overapproximate by using the power set function P to consider all the possibilities. Therefore, $(\bigcup_i T_i) \cup \{\boldsymbol{err}\}$ covers all constraints that are reachable by $\Xi, \Sigma, \hat{\Sigma} \vdash \tau \ll \sigma$ and $\Xi \cup \Sigma \cup \hat{\Sigma}$ itself, where $\Xi = \epsilon$ initially. Since C-Hyp prevents generating duplicate constraints and the recursive algorithm always increases the size of $\Xi \cup \Sigma$, So it suffices to show that $\bigcup_i T_i$ is finite.

If $\mathcal{D}$ ***wf***, then for all τ, the number of ADTs appearing in τ is finite, and $TV(\tau)$ is also finite. The maximum type constructor depth is bound by some number tc, which is equivalent to that of types in T_0. Notice that $\mathit{dnf} \circ \varrho_{\hat{\Sigma}}$ will not introduce new ADTs or type variables. *dnf* will not increase the type constructor depth. Althogh $\varrho_{\hat{\Sigma}}$ might increase the type constructor depth, we merely inline the bounds for skolems at the first level, which means the whole type constructor depth is still bound by tc. For all $\mathbb{C} \in S_i$, it can be constructed by using ADTs and existing type variables, plus extruded variables, within the constructor depth tc. Notice that ***X-fresh*** makes sure each type variable can be extruded to a level only once. Since there is a finite number of reachable polymorphic levels from a given set of constraints, the number of extruded variables is also finite. Therefore, the size of S_i is bound. Derived from S_i, T_{i+1} is then finite. Thus, the size of $\bigcup_i T_i$ is bound. □

D.7 Completeness of Type Inference

Definition D.56. We write $\mathit{fresh}(D)$ to denote all the type variables that are taken as fresh in the given derivation D and are not skolems.

Definition D.57. A fresh type variable substitution ρ is well-formed with context Σ (written as (Σ, ρ) ***wf***), if for all $\alpha^m \in \mathit{dom}(\rho)$, we have $\boldsymbol{lv}(\rho(\alpha^m), \Sigma) \leq m$.

Definition D.58. $\mathbb{D} = \bigvee \mathbb{C}_i$ is well-formed, written as $(\Sigma, \mathbb{D})$ ***wf***, if for all $\mathbb{C}_i$, the following conditions are false:

(1) $\mathbb{C}_i = (\mathbb{D}_1 \xrightarrow{\mathbb{D}_2} \mathbb{D}_3) \land \neg((\mathbb{D}_4 \xrightarrow{\mathbb{D}_5} \mathbb{D}_6) \lor \ldots) \land \ldots \land \neg^{\diamond}\alpha^m \land \ldots$, where $m < \boldsymbol{lv}(\mathbb{D}_1 \xrightarrow{\mathbb{D}_2} \mathbb{D}_3, \Sigma)$ and $m < \boldsymbol{lv}(\mathbb{D}_4 \xrightarrow{\mathbb{D}_5} \mathbb{D}_6, \Sigma)$,

(2) $\mathbb{C}_i = \mathsf{A}_1[\overline{\mathbf{in}\,\mathbb{D}_{i1}\ \mathbf{out}\,\mathbb{D}'_{i1}}] \land \neg(\mathsf{A}_2[\overline{\mathbf{in}\,\mathbb{D}_{i2}\ \mathbf{out}\,\mathbb{D}'_{i2}}] \lor U) \land \ldots \land \neg^{\diamond}\alpha^m \land \ldots$, where $\mathsf{A}_1 = \mathsf{A}_2$, $m < \boldsymbol{lv}(\mathsf{A}_1[\overline{\mathbf{in}\,\mathbb{D}_{i1}\ \mathbf{out}\,\mathbb{D}'_{i1}}], \Sigma)$ and $m < \boldsymbol{lv}(\mathsf{A}_2[\overline{\mathbf{in}\,\mathbb{D}_{i2}\ \mathbf{out}\,\mathbb{D}'_{i2}}], \Sigma)$.

Lemma D.59 (Completeness of Extrusion). *If* $\Sigma\,\hat{\Sigma}$ ***cons.***, $(\hat{\Sigma}, \rho)$ ***wf***, $\Sigma\,\hat{\Sigma} \vDash \rho(\Sigma_0)$, $m < \boldsymbol{lv}(\mathbb{D}, \hat{\Sigma})$, $\boldsymbol{lv}(\tau, \hat{\Sigma}) \leq m$, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq^{\pm} \rho(\mathbb{D})$, $(\hat{\Sigma}, \mathit{dnf}(\neg^{\mp}(\tau \land^{\pm} \neg\mathbb{D})))$ ***wf***, $(\hat{\Sigma}, \mathbb{D}) \overset{(\mp,m)}{\rightsquigarrow} (\Sigma', \sigma)$ *(denoted as D), then* $\Sigma\,\hat{\Sigma} \vdash \rho'(\tau) \leq^{\pm} \rho'(\sigma)$ *and* $\Sigma\,\hat{\Sigma} \vDash \rho'(\Sigma')$, *where* ρ' *extends* ρ, $(\hat{\Sigma}, \rho')$ ***wf***, *and* $\mathit{dom}(\rho') \setminus \mathit{dom}(\rho) = \mathit{fresh}(D)$.

Proof. We prove the case where $\pm = +$. For $\pm = -$, the proof is symmetric.
By induction on the syntax of $\mathbb{D}$.

Case $\mathbb{D} = \bot$, $\mathbb{D} = \top$**.** Impossible since $\boldsymbol{lv}(\mathbb{D}, \hat{\Sigma}) = 0$.

Case $\mathbb{D} = \hat{\alpha}$**.** Then $m < \boldsymbol{lv}(\hat{\alpha}, \hat{\Sigma})$. The premise $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \rho(\hat{\alpha})$ implies $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \sigma$ for some σ, where $\Sigma\,\hat{\Sigma} \vdash \sigma \leq \rho(lb_{\hat{\Sigma}}(\hat{\alpha}))$ and $\boldsymbol{lv}(\sigma, \hat{\Sigma}) \leq m$. Consider $(\hat{\alpha}, -, \beta^m)$ ***X-fresh***. We conclude by $\rho' = [\sigma/\beta^m] \circ \rho$.

Case $\mathbb{D} = \alpha^n$**.** Then $m < n$. $\Sigma\,\hat{\Sigma} \vDash \rho(\Sigma_0)$ implies $\Sigma\,\hat{\Sigma} \vdash \rho(lb_{\Sigma_0}(\alpha^n)) \leq \rho(\alpha^n)$. The premise $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \rho(\alpha^n)$ implies $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \sigma$ for some σ, where $\Sigma\,\hat{\Sigma} \vdash \sigma \leq \rho(lb_{\Sigma_0}(\alpha^n))$ and $\boldsymbol{lv}(\sigma, \hat{\Sigma}) \leq m$. Consider $(\alpha^n, -, \beta^m)$ ***X-fresh***. We conclude by $\rho' = [\sigma/\beta^m] \circ \rho$.

Case $\mathbb{D} = \mathbb{D}_1 \xrightarrow{\mathbb{D}_2} \mathbb{D}_3$**.** By case analysis on the subtyping judgment $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \rho(\mathbb{D})$.

Case S-Top−. Then $\Sigma\,\hat{\Sigma} \vdash \bot \leq \rho(\mathbb{D})$. We conclude by IH.

Case S-Refl, S-AndOrL+, S-AndOrR−, S-AndOr−, S-FunMrg+. Impossible due to the level requirement.

Case S-Fun. Then $\rho(\tau) = \rho(\tau_1 \xrightarrow{\varphi} \tau_2)$ for some τ_1, τ_2, and φ. By Theorem D.31, $\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}_1) \leq \rho(\tau_1)$, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_2) \leq \rho(\mathbb{D}_3)$, and $\Sigma\,\hat{\Sigma} \vdash \rho(\varphi) \leq \rho(\mathbb{D}_2)$. We conclude by IH and S-Fun.

Case $\mathbb{D} = \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_i\ \mathbf{out}\,\mathbb{D}'_i}]$**.** Similar to the case $\mathbb{D} = \mathbb{D}_1 \xrightarrow{\mathbb{D}_2} \mathbb{D}_3$.

Case $\mathbb{D} = \neg U$**.** Then $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \neg\rho(U)$. If $\tau = \neg\sigma$, then by Theorem D.1, $\Sigma\,\hat{\Sigma} \vdash \rho(U) \leq \rho(\tau)$. For each component τ_U of U, we can directly rewrite it into normal form $\mathbb{D}'$. By S-AndOr−, $\Sigma\,\hat{\Sigma} \vdash \tau_U \leq \rho(\tau)$. We conclude by IH. Otherwise, by Theorem D.1, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau \land U) \leq \bot$. The only possible subtyping derivation is either S-CtorBot or S-CFBot. We conclude by IH. Notice that S-Compl− is impossible due to the level requirement.

Case $\mathbb{D} = I \wedge \neg U$**.** By premise $(\hat{\Sigma}, dnf(\tau \wedge \neg\mathbb{D}))$ ***wf***, the subtyping judgment cannot be derived from S-Compl±, S-FunMrg±, or S-CtorMrg±. Then by S-AndOr+, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \rho(I)$ and $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \neg\rho(U)$. We conclude by IH and S-AndOr+.

Case $\mathbb{D} = I \wedge \neg U \wedge \neg^{\diamond} v$**.** Similar to the case $\mathbb{D} = I \wedge \neg U$.

Case $\mathbb{D} = \mathbb{D}' \vee \mathbb{C}$**.** By S-AndOrL− and S-AndOrR−, either $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \rho(\mathbb{D}')$ or $\Sigma\,\hat{\Sigma} \vdash \rho(\tau) \leq \rho(\mathbb{C})$. We conclude by IH.

□

Lemma D.60. *Given* $\hat{\Sigma}$,

(1) If $\tau_1 \wedge \neg\tau_2$ ***wf****, then* $(\hat{\Sigma}, dnf(\tau_1 \wedge \neg\tau_2))$ ***wf****.*

(2) If $\mathsf{A}[\overline{\textbf{in}\ \tau_i\ \textbf{out}\ \sigma_i}] \wedge \neg\mathsf{A}[\overline{\textbf{in}\ \tau_i'\ \textbf{out}\ \sigma_i'}]$ ***wf****, then* $\overline{(\hat{\Sigma}, dnf(\tau_i' \wedge \neg\tau_i))}$ ***wf*** *and* $\overline{(\hat{\Sigma}, dnf(\sigma_i \wedge \neg\sigma_i'))}$ ***wf****.*

(3) If $(\tau_1 \xrightarrow{\varphi} \tau_2) \wedge \neg(\sigma_1 \xrightarrow{\varphi'} \sigma_2)$ ***wf****, then* $(\hat{\Sigma}, \sigma_1 \wedge \neg\tau_1)$ ***wf****,* $(\hat{\Sigma}, \tau_2 \wedge \neg\sigma_2)$ ***wf****, and* $(\hat{\Sigma}, \varphi \wedge \neg\varphi')$ ***wf****.*

Proof. By induction on the syntax of $\tau_1 \wedge \neg\tau_2$. □

Lemma D.61 (Necessity of constraining). *Given* $\Sigma\,\hat{\Sigma}$ ***cons.*** *and* $(\hat{\Sigma}, \rho)$ ***wf****,*

(1) If $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_1) \leq \rho(\tau_2)$*,* $(\hat{\Sigma}, dnf(\tau_1 \wedge \neg\tau_2))$ ***wf****,* $\Sigma\,\hat{\Sigma} \vDash \rho(\Sigma_0)$*, and* $\Xi, \Sigma_0, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma'$ *(denoted as D), then* $\Sigma\,\hat{\Sigma} \vDash \rho'(\Sigma')$*.*

(2) If $\Sigma\,\hat{\Sigma} \vdash \rho(\bigvee_i \mathbb{C}_i) \leq \bot$*,* $(\hat{\Sigma}, \bigvee_i \mathbb{C}_i)$ ***wf****,* $\Sigma\,\hat{\Sigma} \vDash \rho(\Sigma_0)$*, and* $\Xi, \Sigma_0, \hat{\Sigma} \vdash \bigvee_i \mathbb{C}_i \Rightarrow \Sigma'$ *(denoted as D), then* $\Sigma\,\hat{\Sigma} \vDash \rho'(\Sigma')$*.*

where ρ' *extends* ρ*,* $(\hat{\Sigma}, \rho')$ ***wf****, and* $dom(\rho') \setminus dom(\rho) = fresh(D)$*.*

Proof. By induction on the constraining derivations.

Case C-Hyp, C-Bot. Immediately by S-Empty.

Case C-Assum. Then $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_1) \leq \rho(\tau_2)$. By the premise, $\Xi \rhd (\tau_1 \leq \tau_2), \Sigma_0, \hat{\Sigma} \vdash dnf \circ \varrho_{\hat{\Sigma}}(\tau_1 \wedge \neg\tau_2) \Rightarrow \Sigma'$. By Theorem D.1, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_1) \wedge \neg\rho(\tau_2) \leq \bot$. Therefore, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_1 \wedge \neg\tau_2) \leq \bot$. By Lemma C.4 and Lemma D.13, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_1 \wedge \neg\tau_2) \equiv \rho(dnf \circ \varrho_{\hat{\Sigma}}(\tau_1 \wedge \neg\tau_2))$. We conclude by S-Trans and IH.

Case C-Or. Then $\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}' \vee \mathbb{C}) \leq \bot$. By premises,

$$\Xi, \Sigma_0, \hat{\Sigma} \vdash \mathbb{D}' \Rightarrow \Sigma_1 \tag{1}$$

$$\Xi, \Sigma_0\,\Sigma_1, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma_2 \tag{2}$$

Let $\Sigma' = \Sigma_1\,\Sigma_2$. By S-AndOrL− and S-AndOrR−,

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}') \leq \rho(\mathbb{D}' \vee \mathbb{C}) \tag{3}$$

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{C}) \leq \rho(\mathbb{D}' \vee \mathbb{C}) \tag{4}$$

Then by S-Trans on (3) and (4),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}') \leq \bot \tag{5}$$

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{C}) \leq \bot \tag{6}$$

By IH on (1) and (5),

$$\Sigma\,\hat{\Sigma} \vDash \rho_1(\Sigma_1) \tag{7}$$

where ρ_1 extends ρ. By Lemma D.18 on (7) and the premise $\hat{\Sigma} \vDash \rho(\Sigma_0)$,

$$\Sigma\,\hat{\Sigma} \vDash \rho_1(\Sigma_1\,\Sigma_0) \tag{8}$$

By IH on (4), (6), and (8),

$$\Sigma\,\hat{\Sigma} \vDash \rho'(\Sigma_2) \tag{9}$$

where ρ' extends ρ_1. We conclude by Lemma D.18 on (8) and (9).

Case C-NotBot. Then $\Sigma\,\hat{\Sigma} \vdash \rho(I) \wedge \neg\bot \leq \bot$. By Theorem D.1, $\Sigma\,\hat{\Sigma} \vdash \rho(I) \leq \bot$. No matter which case is picked, it contradicts Theorem D.31.

Case C-Fun2, C-Fun3. Similar to C-NotBot.

Case C-Ctor1. Then

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i1}\,\mathbf{out}\,\mathbb{D}_{i3}}] \wedge \neg(U \vee \mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i2}\,\mathbf{out}\,\mathbb{D}_{i4}}])) \leq \bot \tag{10}$$

By premises,

$$\overline{\lhd\Xi, \Sigma_0\,\overline{\Sigma_j}^{j\in 1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i2} \ll \mathbb{D}_{i1} \Rightarrow \Sigma_i}^{i} \tag{11}$$

$$\overline{\lhd\Xi, \Sigma_0\,\overline{\Sigma_j}^{j}\,\overline{\Sigma'_k}^{k\in 1\ldots i-1}, \hat{\Sigma} \vdash \mathbb{D}_{i3} \ll \mathbb{D}_{i4} \Rightarrow \Sigma'_i}^{i} \tag{12}$$

By Theorem D.3 and then Theorem D.1 on (10),

$$\Sigma\,\hat{\Sigma} \vdash \neg\rho(U) \leq \neg\rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i1}\,\mathbf{out}\,\mathbb{D}_{i3}}]) \vee \rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i2}\,\mathbf{out}\,\mathbb{D}_{i4}}]) \tag{13}$$

By S-Top− and S-Trans on (13),

$$\Sigma\,\hat{\Sigma} \vdash \bot \leq \neg\rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i1}\,\mathbf{out}\,\mathbb{D}_{i3}}]) \vee \rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i2}\,\mathbf{out}\,\mathbb{D}_{i4}}]) \tag{14}$$

By Theorem D.1 on (14),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i1}\,\mathbf{out}\,\mathbb{D}_{i3}}]) \leq \rho(\mathsf{A}[\overline{\mathbf{in}\,\mathbb{D}_{i2}\,\mathbf{out}\,\mathbb{D}_{i4}}]) \tag{15}$$

By Theorem D.31 on (15),

$$\overline{\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}_{i2}) \leq \rho(\mathbb{D}_{i1})}^{i} \tag{16}$$

$$\overline{\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}_{i3}) \leq \rho(\mathbb{D}_{i4})}^{i} \tag{17}$$

Let $\Sigma' = \overline{\Sigma_i}\,\overline{\Sigma'_i}$, we conclude by IH and Lemma D.18 and Lemma D.60 on (11), (12), (16), and (17).

Case C-Ctor2, C-Ctor3, C-Ctor4. Similar to C-Ctor1.

Case C-Fun1. Then

$$\Sigma\,\hat{\Sigma} \vdash \rho((\mathbb{D}_1 \xrightarrow{\mathbb{D}_5} \mathbb{D}_2) \wedge \neg(\mathbb{D}_3 \xrightarrow{\mathbb{D}_6} \mathbb{D}_4)) \leq \bot \tag{18}$$

By premises,

$$\lhd\Xi, \Sigma_0, \hat{\Sigma} \vdash \mathbb{D}_3 \ll \mathbb{D}_1 \Rightarrow \Sigma_1 \tag{19}$$

$$\lhd\Xi, \Sigma_0\,\Sigma_1, \hat{\Sigma} \vdash \mathbb{D}_2 \ll \mathbb{D}_4 \Rightarrow \Sigma_2 \tag{20}$$

$$\lhd\Xi, \Sigma_0\,\Sigma_1\,\Sigma_2, \hat{\Sigma} \vdash \mathbb{D}_5 \ll \mathbb{D}_6 \Rightarrow \Sigma_3 \tag{21}$$

By Theorem D.1 on (18),

$$\Sigma\,\hat{\Sigma} \vdash \rho((\mathbb{D}_1 \xrightarrow{\mathbb{D}_5} \mathbb{D}_2) \leq \rho(\mathbb{D}_3 \xrightarrow{\mathbb{D}_6} \mathbb{D}_4)) \tag{22}$$

By Theorem D.31 on (22),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}_3) \leq \rho(\mathbb{D}_1) \tag{23}$$

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}_2) \leq \rho(\mathbb{D}_4) \tag{24}$$

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{D}_5) \leq \rho(\mathbb{D}_6) \tag{25}$$

Then we conclude by IH and Lemma D.60 on (19), (20), (21), (23), (24), and (25).

Case C-Sk. We prove the case where $\pm = -$. The proof for $\pm = +$ is symmetric.

Then

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{C} \wedge \alpha) \leq \bot \tag{26}$$

By premises,

$$\Xi, \Sigma_0, \hat{\Sigma} \vdash \mathbb{C} \Rightarrow \Sigma' \tag{27}$$

By Theorem D.1 on (26),

$$\Sigma\,\hat{\Sigma} \vdash \alpha \leq \neg\rho(\mathbb{C}) \tag{28}$$

Notice that skolems cannot be further bound and all bounds are already inlined by $\varrho_{\hat{\Sigma}}$. Therefore, (14) implies

$$\Sigma\,\hat{\Sigma} \vdash \top \leq \neg\rho(\mathbb{C}) \tag{29}$$

Then by Theorem D.1 on (29),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{C}) \leq \bot \tag{30}$$

We conclude by IH on (30).

Case C-Var1. Then

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{C} \wedge \alpha^m) \leq \bot \tag{31}$$

By premises,

$$\Xi, \Sigma_0\ (\alpha^m \leq \neg\mathbb{C}), \hat{\Sigma} \vdash \mathsf{lb}_{\Sigma_0}(\alpha^m) \ll \neg\mathbb{C} \Rightarrow \Sigma_1 \tag{32}$$

By Theorem D.1 on (31),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\alpha^m) \leq \neg\rho(\mathbb{C}) \tag{33}$$

By S-AndOr− and S-Hyp,

$$\Sigma_0 \vdash \mathsf{lb}_{\Sigma_0}(\alpha^m) \leq \alpha^m \tag{34}$$

Then by S-Hyp and S-Trans on (34),

$$\Sigma_0\ (\alpha^m \leq \neg\mathbb{C}) \vdash \mathsf{lb}_{\Sigma_0}(\alpha^m) \leq \neg\mathbb{C} \tag{35}$$

Therefore, by Corollary D.21 on (35),

$$\rho(\Sigma_0\ (\alpha^m \leq \neg\mathbb{C})) \vdash \rho(\mathsf{lb}_{\Sigma_0}(\alpha^m)) \leq \rho(\neg\mathbb{C}) \tag{36}$$

Then by S-Cons on (33) and the premise $\Sigma\,\hat{\Sigma} \models \rho(\Sigma_0)$,

$$\Sigma\,\hat{\Sigma} \models \rho(\Sigma_0\ (\alpha^m \leq \neg\mathbb{C})) \tag{37}$$

By Theorem D.15 on (36) and (37),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathsf{lb}_{\Sigma_0}(\alpha^m)) \leq \rho(\neg\mathbb{C}) \tag{38}$$

By IH on (38), (37), and (32),

$$\Sigma\,\hat{\Sigma} \models \rho'(\Sigma_1) \tag{39}$$

where ρ' extends ρ. Let $\Sigma' = \Sigma_1\ (\alpha^m \leq \neg\mathbb{C})$. We conclude by S-Cons on (38) and (39).

Case C-Var2. Similar to C-Var1.

Case C-Var3. Then

$$\Sigma\,\hat{\Sigma} \vdash \rho(\mathbb{C} \wedge \alpha^m) \leq \bot \tag{40}$$

By premises,

$$(\hat{\Sigma}, \neg\mathbb{C}) \overset{(-,m)}{\rightsquigarrow} (\Sigma', \tau) \tag{41}$$

$$\Xi, \Sigma_0, \hat{\Sigma} \vdash \Sigma' \Rightarrow \Sigma'' \tag{42}$$

$$\Xi, \Sigma_0\, \Sigma''\ (\alpha^m \leq \tau), \hat{\Sigma} \vdash \mathsf{lb}_{\Sigma_0}(\alpha^m) \ll \tau \Rightarrow \Sigma_1 \tag{43}$$

By Theorem D.1 on (40),

$$\Sigma\,\hat{\Sigma} \vdash \rho(\alpha^m) \leq \neg\rho(\mathbb{C}) \tag{44}$$

By Lemma D.59 on (41), (44), and $(\hat{\Sigma}, \alpha^m \wedge \mathbb{C})$ ***wf***,

$$\Sigma\,\hat{\Sigma} \vdash \rho_0(\alpha^m) \leq \rho_0(\tau) \tag{45}$$

$$\Sigma\,\hat{\Sigma} \vDash \rho_0(\Sigma') \tag{46}$$

where ρ_0 extends ρ. By IH on (42) and (46),

$$\Sigma\,\hat{\Sigma} \vDash \rho_1(\Sigma'') \tag{47}$$

where ρ_1 extends ρ_0. Then by S-Cons on (45), (46), and (47), and the premise $\Sigma\,\hat{\Sigma} \vDash \rho(\Sigma_0)$,

$$\Sigma\,\hat{\Sigma} \vDash \rho_1(\Sigma_0\, \Sigma''\ (\alpha^m \leq \tau)) \tag{48}$$

By S-AndOr− and S-Hyp,

$$\Sigma_0 \vdash \mathsf{lb}_{\Sigma_0}(\alpha^m) \leq \alpha^m \tag{49}$$

Then by S-Hyp and S-Trans on (49),

$$\Sigma_0\ (\alpha^m \leq \tau) \vdash \mathsf{lb}_{\Sigma_0}(\alpha^m) \leq \tau \tag{50}$$

Therefore, by Corollary D.21 on (50),

$$\rho_1(\Sigma_0\ (\alpha^m \leq \tau)) \vdash \rho_1(\mathsf{lb}_{\Sigma_0}(\alpha^m)) \leq \rho_1(\tau) \tag{51}$$

By Theorem D.15 on (48) and (51),

$$\Sigma\,\hat{\Sigma} \vdash \rho_1(\mathsf{lb}_{\Sigma_0}(\alpha^m)) \leq \rho_1(\tau) \tag{52}$$

By IH on (43), (46), and (52),

$$\Sigma\,\hat{\Sigma} \vDash \rho'(\Sigma_1) \tag{53}$$

where ρ' extends ρ_0. We conclude by S-Cons on (48) and (53).

Case C-Var4. Similar to C-Var3.

□

Lemma D.62 (Completeness of constraining). *If $\Sigma\,\hat{\Sigma}$ **cons.**, $\Sigma\,\hat{\Sigma} \vdash \rho(\tau_1) \leq \rho(\tau_2)$ for some type variable substitution ρ, $(\hat{\Sigma}, \rho)$ **wf**, $\Sigma\,\hat{\Sigma} \vDash \rho(\Sigma_0)$, then $\epsilon, \Sigma_0, \hat{\Sigma} \vdash \tau_1 \ll \tau_2 \Rightarrow \Sigma_1$ (denoted as D), where **err** $\notin \Sigma_1$, $\Sigma\,\hat{\Sigma} \vDash \rho'(\Sigma_1)$, ρ' extends ρ, $(\hat{\Sigma}, \rho')$ **wf**, and $dom(\rho') \setminus dom(\rho) = fresh(D)$.*

Proof. By Theorem D.55, Lemma D.60, and Lemma D.61. □

Lemma D.63. *If $\Gamma\ (x : \tau) \vdash t : \mathcal{T}\ !\ \varphi$, then $\Gamma\ (x : \alpha)\ (\alpha \leq \tau) \vdash t : \mathcal{S}\ !\ \varphi'$, $sub(\Gamma) \vdash [\tau/\alpha]\mathcal{S} \leq^{\forall} \mathcal{T}$, and $sub(\Gamma) \vdash [\tau/\alpha]\varphi' \leq \varphi$ for any α **fresh** and some $\mathcal{S}$ and φ'.*

Proof. By induction on the typing derivations. □

Lemma D.64 (General completeness of type inference). *Given definitions $\mathcal{D}$ **wf**, if $\Gamma\,\Sigma$ **cons.**, where $sub(\Gamma)$ only maintains skolem bounds, $\Gamma\,\Sigma,\ \zeta \vdash t : \mathcal{T} \mathbin{!} \varphi$, $sub(\Gamma)\,\Sigma \vDash \rho(\Sigma_0)$ for some substitution ρ, and $(\hat{\Sigma}, \rho)$ **wf**, then $\Gamma,\ \zeta \vdash t : \mathcal{S} \mathbin{!} \varphi' \Rightarrow \Xi$ (denoted as D_1), $\epsilon, \Sigma_0, sub(\Gamma) \vdash \Xi \Rightarrow \Sigma'$ (denoted as D_2), $sub(\Gamma)\,\Sigma \vdash \rho'(\mathcal{S}) \leq^{\forall} \mathcal{T}$, $sub(\Gamma)\,\Sigma \vdash \rho'(\varphi') \leq \varphi$, and $sub(\Gamma)\,\Sigma \vDash \rho'(\Sigma_0\,\Sigma')$, where ρ' extends ρ, $(\hat{\Sigma}, \rho')$ **wf**, and $dom(\rho') \setminus dom(\rho) = fresh(D_1) \cup fresh(D_2)$.*

Proof. By induction on the typing derivations (IH).

Case T-Var. Then $t = x$. By premise, $\Gamma(x) = \mathcal{T}$. By I-Var, $\Gamma,\ \zeta \vdash t : \mathcal{T} \mathbin{!} \bot \Rightarrow \epsilon$. We conclude by S-PRefl.

Case T-Abs1. Then $t = \lambda x.\, t'$. By the premise,

$$\Gamma\,\Sigma\,(x : \tau),\ \zeta \vdash t' : \mathcal{S} \mathbin{!} \varphi_1 \tag{54}$$

By Lemma D.63 on (54),

$$\Gamma\,\Sigma\,(\alpha \leq \tau)\,(x : \alpha),\ \zeta \vdash t' : \mathcal{S}' \mathbin{!} \varphi_2 \tag{55}$$

$$sub(\Gamma)\,\Sigma \vdash [\tau/\alpha]\mathcal{S}' \leq^{\forall} \mathcal{S} \tag{56}$$

$$sub(\Gamma)\,\Sigma \vdash [\tau/\alpha]\varphi_2' \leq \varphi_1 \tag{57}$$

By IH on (55),

$$\Gamma\,(x : \alpha^n),\ \zeta \vdash t' : \mathcal{S}'' \mathbin{!} \varphi_3 \Rightarrow \Xi \tag{58}$$

$$\epsilon, \Sigma_0, sub(\Gamma) \vdash \Xi \Rightarrow \Sigma' \tag{59}$$

$$sub(\Gamma)\,\Sigma\,(\alpha \leq \tau) \vdash \rho_1(\mathcal{S}'') \leq^{\forall} \mathcal{S}' \tag{60}$$

$$sub(\Gamma)\,\Sigma\,(\alpha \leq \tau) \vdash \rho_1(\varphi_3) \leq \varphi_2 \tag{61}$$

$$sub(\Gamma)\,\Sigma\,(\alpha \leq \tau) \vDash \rho_1(\Sigma_0\,\Sigma') \tag{62}$$

for some ρ_1 and n, where ρ_1 extends ρ and $n = \boldsymbol{lv}(\Gamma)$. By I-Abs1 on (58),

$$\Gamma\,(x : \alpha^n),\ \zeta \vdash (\lambda x.\, t') : \alpha \xrightarrow{\varphi_3} \mathcal{S}'' \mathbin{!} \bot \Rightarrow \Xi \tag{63}$$

By Corollary D.21 and S-Refl on (60), (61), and (62),

$$sub(\Gamma)\,\Sigma \vdash \rho(\mathcal{S}'') \leq^{\forall} [\tau/\alpha]\mathcal{S}' \tag{64}$$

$$sub(\Gamma)\,\Sigma \vdash \rho(\varphi_3) \leq [\tau/\alpha]\varphi_2 \tag{65}$$

$$sub(\Gamma)\,\Sigma \vDash \rho(\Sigma_0\,\Sigma') \tag{66}$$

where $\rho = [\tau/\alpha] \circ \rho_1$. Notice that α is fresh, so $\alpha \notin dom(\rho_1) \cup TV(\Sigma) \cup TV(\Sigma_0) \cup TV(\Gamma)$. By Lemma D.33 on (64), (65), (56), and (57),

$$sub(\Gamma)\,\Sigma \vdash \rho(\mathcal{S}'') \leq^{\forall} \mathcal{S} \tag{67}$$

$$sub(\Gamma)\,\Sigma \vdash \rho(\varphi_3) \leq \varphi_1 \tag{68}$$

We conclude by S-PFun on (67), (68) and $sub(\Gamma)\,\Sigma \vdash \rho(\alpha) \leq \tau$.

Case T-Abs2. Then $t = (\lambda x.\, t') : \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2$. By premises,

$$\Gamma\,\Sigma\,(x : \mathcal{T}_1),\ \zeta \vdash (t' : \mathcal{T}_2) : \mathcal{T}_2 \mathbin{!} \varphi \tag{69}$$

By IH on (69),

$$\Gamma\ (x : \mathcal{T}_1),\ \zeta \vdash (t' : \mathcal{T}_2) : \mathcal{T}_2\ !\ \varphi_1 \Rightarrow \Xi_1 \tag{70}$$
$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Xi_1 \Rightarrow \Sigma_1 \tag{71}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vdash \rho_1(\varphi_1) \leq \varphi \tag{72}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vDash \rho_1(\Sigma_0\ \Sigma_1) \tag{73}$$

where ρ_1 extends ρ. Assume that $\epsilon, \Sigma_0\ \Sigma_1, \mathit{sub}(\Gamma) \vdash \varphi_1 \leq \varphi \Rightarrow \Sigma_2$. By Lemma D.62 and Lemma D.60 on (73),

$$\mathit{sub}(\Gamma)\ \Sigma \vDash \rho_2(\Sigma_2) \tag{74}$$

where $\boldsymbol{\mathit{err}} \notin \Sigma_2$ and ρ_2 extends ρ_1. We can rewrite (72) and (73),

$$\mathit{sub}(\Gamma)\ \Sigma \vdash \rho_2(\varphi_1) \leq \varphi \tag{75}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vDash \rho_2(\Sigma_0\ \Sigma_1) \tag{76}$$

Let $\Sigma' = \Sigma_1\ \Sigma_2$. By Lemma D.18 on (74) and (76),

$$\mathit{sub}(\Gamma)\ \Sigma \vDash \rho_2(\Sigma_0\ \Sigma') \tag{77}$$

We conclude by I-Asb2 on (70) and S-PFun on (75).

Case T-App. Then $t = t_1\ t_2$. By premises,

$$\Gamma\ \Sigma,\ \zeta \vdash t_1 : \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2\ !\ \varphi \tag{78}$$
$$\Gamma\ \Sigma,\ \zeta \vdash t_2 : \mathcal{T}_1\ !\ \varphi \tag{79}$$

By IH on (78),

$$\Gamma,\ \zeta \vdash t_1 : \mathcal{S}\ !\ \varphi_1 \Rightarrow \Xi_1 \tag{80}$$
$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Xi_1 \Rightarrow \Sigma_1 \tag{81}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vdash \rho_1(\mathcal{S}) \leq \mathcal{T}_1 \xrightarrow{\varphi} \mathcal{T}_2 \tag{82}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vdash \rho_1(\varphi_1) \leq \varphi \tag{83}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vDash \rho_1(\Sigma_0\ \Sigma_1) \tag{84}$$

where ρ_1 extends ρ. We prove the case where $\mathcal{T}_1 = \tau_1$, $\mathcal{T}_2 = \tau_2$, and $\mathcal{S} = \tau^{\nrightarrow}$ for some $\tau^{\nrightarrow}$. The proof for higher-ranked functions and $\mathcal{S} = \mathcal{S}_1 \xrightarrow{\varphi'} \mathcal{S}_2$ is similar but requires no further constraints. Similarly, by IH on (79),

$$\Gamma,\ \zeta \vdash t_2 : \tau_1'\ !\ \varphi_2 \Rightarrow \Xi_2 \tag{85}$$
$$\epsilon, \Sigma_0\ \Sigma_1, \mathit{sub}(\Gamma) \vdash \Xi_2 \Rightarrow \Sigma_2 \tag{86}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vdash \rho_2(\tau_1') \leq \tau_1 \tag{87}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vdash \rho_2(\varphi_2) \leq \varphi \tag{88}$$
$$\mathit{sub}(\Gamma)\ \Sigma \vDash \rho_2(\Sigma_0\ \Sigma_1\ \Sigma_2) \tag{89}$$

where ρ_2 extends ρ_1. We introduce fresh type variables β and γ. By I-App1 on (80) and (85),

$$\Gamma,\ \zeta \vdash t_1\ t_2 : \beta\ !\ \varphi_1 \vee \varphi_2 \vee \gamma \Rightarrow \Xi \tag{90}$$

where $\Xi = \Xi_1\, \Xi_2\, (\tau^{\not\leftrightarrow} \leq \tau_1' \xrightarrow{\gamma} \beta)$. Let $\rho_3 = [\tau_2/\beta,\ \varphi/\gamma] \circ \rho_2$. Therefore, we can rewrite (82), (83), (84), (87), (88), and (89) by Corollary D.21,

$$sub(\Gamma)\,\Sigma \vdash \rho_3(\tau^{\not\leftrightarrow}) \leq \tau_1 \xrightarrow{\varphi} \tau_2 \tag{91}$$

$$sub(\Gamma)\,\Sigma \vdash \rho_3(\varphi_1) \leq \varphi \tag{92}$$

$$sub(\Gamma)\,\Sigma \vdash \rho_3(\tau_1') \leq \tau_1 \tag{93}$$

$$sub(\Gamma)\,\Sigma \vdash \rho_3(\varphi_2) \leq \varphi \tag{94}$$

$$sub(\Gamma)\,\Sigma \vDash \rho_3(\Sigma_0\,\Sigma_1\,\Sigma_2) \tag{95}$$

By S-Trans and S-Fun on (91) and (93)

$$sub(\Gamma)\,\Sigma \vdash \rho_3(\tau^{\not\leftrightarrow}) \leq \rho_3(\tau_1' \xrightarrow{\gamma} \beta) \tag{96}$$

By Lemma D.62 and Lemma D.60 on (95), (96) and the constraint $\epsilon, \Sigma_0\,\Sigma_1\,\Sigma_2, sub(\Gamma) \vdash \tau^{\not\leftrightarrow} \ll \tau_1' \xrightarrow{\gamma} \beta \Rightarrow \Sigma_3$,

$$sub(\Gamma)\,\Sigma \vDash \rho(\Sigma_3) \tag{97}$$

for some ρ', where ρ' extends ρ_3. We conclude by Lemma D.18 on (95) and (97).

Case T-Gen. Then $t = t' : \forall V\{\Sigma''\}.\,\mathcal{S}$. By premises,

$$\Gamma\,\Sigma \bullet V\,\Sigma'',\ \zeta \vee \omega \vdash (t' : \mathcal{S}) : \mathcal{S}\ !\ \bot \tag{98}$$

$$sub(\Gamma)\,\Sigma \vdash \forall V\{\Sigma''\}\ \boldsymbol{cons.} \tag{99}$$

where $\omega \in V$. Then by IH on (98) and (99),

$$\Gamma \bullet V\,\Sigma'',\ \zeta \vee \omega \vdash (t' : \rho'(\mathcal{S})) : \rho'(\mathcal{S})\ !\ \varphi \Rightarrow \Xi' \tag{100}$$

$$\epsilon, \epsilon, sub(\Gamma \bullet V\,\Sigma'') \vdash \Xi' \Rightarrow \Sigma' \tag{101}$$

$$sub(\Gamma)\,\Sigma\,\Sigma'' \vdash \rho_1(\varphi) \leq \bot \tag{102}$$

$$sub(\Gamma)\,\Sigma\,\Sigma'' \vDash \rho_1(\Sigma') \tag{103}$$

where ρ_1 extends ρ. Assume that $\epsilon, \Sigma', sub(\Gamma \bullet V\,\Sigma'') \vdash \varphi \leq \bot \Rightarrow \Sigma_1$. By Lemma D.62 and Lemma D.60 on (102) and (103),

$$sub(\Gamma)\,\Sigma\,\Sigma'' \vDash \rho_2(\Sigma'\,\Sigma_1) \tag{104}$$

where ρ_2 extends ρ_1. Since (99) implies $\epsilon, \epsilon, sub(\Gamma) \vdash \Sigma'' \Rightarrow \Sigma'''$ for some Σ''' and $\boldsymbol{err} \notin \Sigma'''$, by I-Gen on (100) and (102),

$$\Gamma,\ \zeta \vdash (t : \forall V\{\Sigma''\}.\,\mathcal{S}) : \forall V\{\Sigma''\}.\,\mathcal{S}\ !\ \bot \Rightarrow \Sigma'\,\Sigma_1 \tag{105}$$

Assume that

$$\epsilon, \Sigma_0, sub(\Gamma) \vdash \Sigma'\,\Sigma_1 \Rightarrow \Sigma \tag{106}$$

Since for all α at level $n+1$, it is fresh. Then $\alpha^{n+1} \in dom(\rho_2)$. Therefore, (104) implies

$$sub(\Gamma)\,\Sigma \vDash \rho_2(\Sigma'\,\Sigma_1) \tag{107}$$

We conclude by Lemma D.62 and Lemma D.60 on (106) and (107).

Case T-Inst. By premises,

$$\Gamma\,\Sigma,\ \zeta \vdash t : \forall V\{\Sigma'\}.\,\mathcal{T}'\ !\ \varphi \tag{108}$$

$$sub(\Gamma)\,\Sigma \vDash \rho'(\Sigma') \tag{109}$$

$$dom(\rho') = V \tag{110}$$

for some ρ', where $\omega \in V$ and $\rho'(\omega) = \zeta$. By IH on (108),

$$\Gamma, \zeta \vdash t : \mathcal{S}' \mathbin{!} \varphi_1 \Rightarrow \Xi' \tag{111}$$

$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Xi' \Rightarrow \Sigma'' \tag{112}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \rho_1(\mathcal{S}') \leq^{\forall} \forall V\{\Sigma'\}.\mathcal{T}' \tag{113}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \rho_1(\varphi_1) \leq \varphi \tag{114}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vDash \rho_1(\Sigma_0\,\Sigma'') \tag{115}$$

where ρ_1 extends ρ. By Lemma D.32 on (113),

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \rho_1(\forall V\{\Sigma_1\}.\mathcal{S}'') \leq^{\forall} \forall V\{\Sigma'\}.\mathcal{T}' \tag{116}$$

$$\mathit{sub}(\Gamma)\,\Sigma\,\rho_1(\Sigma') \vdash \rho_1(\mathcal{S}'') \leq^{\forall} \mathcal{T}' \tag{117}$$

For some $\mathcal{S}''$ and $\mathit{sub}(\Gamma) \vdash \forall V\{\Sigma_1\}$ ***cons.***, where $\Sigma_1 = \{\overline{\alpha_i \leq^{\pm_i} \tau_i}^{\,i}\}$, $\Sigma' = \{\overline{\alpha_i \leq^{\pm_i} \sigma_i}^{\,i}\}$, and $\overline{\Sigma \vdash \sigma_i \leq^{\pm_i} \tau_i}^{\,i}$. Let $\rho_2 = V$ ***fresh*** and $\rho_3 = \rho_2 \circ \rho_1$. By I-Inst on (111),

$$\Gamma, \zeta \vdash t : \rho_3(\mathcal{S}'') \mathbin{!} \varphi_1 \Rightarrow \Xi'\,\rho_3(\Sigma_1) \tag{118}$$

Assume that

$$\epsilon, \Sigma_0\,\Sigma'', \mathit{sub}(\Gamma) \vdash \rho_3(\Sigma_1) \Rightarrow \Sigma''' \tag{119}$$

By Lemma D.62 and Lemma D.60 on (111) and (119),

$$\mathit{sub}(\Gamma)\,\Sigma \vDash \rho_4(\Sigma_0\,\Sigma''\,\Sigma''') \tag{120}$$

where ρ_4 extends ρ_3. Noice that $\mathit{dom}(\rho_4) \cap \mathit{dom}(\rho_2) = \emptyset$. Let $\rho_5 = \rho'' \circ \rho_4$, where ρ'' maps V ***fresh*** to corresponding types in ρ'. Notice that (109) implies $\mathit{sub}(\Gamma)\,\Sigma \vDash \rho''(\Sigma')$. By Corollary D.21 on (117),

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \rho_5(\mathcal{S}'') \leq^{\forall} \rho'' \circ \rho_2(\mathcal{T}') \tag{121}$$

Since $\rho'' \circ \rho_2$ is equivalent to ρ', (121) implies $\mathit{sub}(\Gamma)\,\Sigma \vdash \rho_5(\mathcal{S}'') \leq^{\forall} \rho'(\mathcal{T}')$.

Case T-Subs1. By premises,

$$\Gamma\,\Sigma, \zeta \vdash t : \tau_1 \mathbin{!} \varphi \tag{122}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \tau_1 \leq \tau_2 \tag{123}$$

By IH on (122),

$$\Gamma, \zeta \vdash t : \sigma \mathbin{!} \varphi_0 \Rightarrow \Xi \tag{124}$$

$$\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Xi \Rightarrow \Sigma' \tag{125}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \rho(\sigma) \leq \tau_1 \tag{126}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vdash \rho(\varphi) \leq \varphi_1 \tag{127}$$

$$\mathit{sub}(\Gamma)\,\Sigma \vDash \rho(\Sigma_0\,\Sigma') \tag{128}$$

where ρ extends ρ_0. We conclude by S-Trans on (126) and (123).

Case T-Subs2, T-Asc. Similar to the case T-Subs1.

Case T-Let. Similar to case T-Abs2.

Case T-Region. Then $t = \textbf{region}\; x \;\textbf{in}\; t'$. By the premise,

$$\Gamma\,\Sigma \bullet \alpha\;(\alpha \leq \neg\zeta)\;(x : \mathrm{Region}[\textbf{out}\;\alpha]), \zeta \vee \alpha \vdash t' : \tau \mathbin{!} \varphi \vee \alpha \tag{129}$$

Then by IH on (129),

$$\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)\ (x : \text{Region}[\textbf{out}\ \alpha]),\ \zeta \vee \alpha \vdash t' : \sigma\ !\ \varphi' \Rightarrow \Xi_1 \tag{130}$$

$$\epsilon, \epsilon, \mathit{sub}(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash \Xi_1 \Rightarrow \Sigma_1 \tag{131}$$

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vdash \rho_1(\sigma) \leq \tau \tag{132}$$

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vdash \rho_1(\varphi') \leq \varphi \vee \alpha \tag{133}$$

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vDash \rho_1(\Sigma_1) \tag{134}$$

where ρ_1 extends ρ. Assume that

$$\epsilon, \Sigma_1, \mathit{sub}(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash \varphi' \leq \gamma^n \vee \alpha \Rightarrow \Sigma_2 \tag{135}$$

$$\epsilon, \Sigma_1\ \Sigma_2, \mathit{sub}(\Gamma \bullet \alpha\ (\alpha \leq \neg\zeta)) \vdash \sigma \leq \beta^n \Rightarrow \Sigma_3 \tag{136}$$

Let $\rho_2 = [\varphi/\gamma^n, \tau/\beta^n] \circ \rho_1$, where $n = \boldsymbol{lv}(\Gamma)$. By Corollary D.21 on (132) and (133),

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vdash \rho_2(\sigma) \leq \rho_2(\beta) \tag{137}$$

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vdash \rho_2(\varphi') \leq \rho_2(\gamma \vee \alpha) \tag{138}$$

By Lemma D.62 and Lemma D.60 on (135), (138), and (134),

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vDash \rho_3(\Sigma_1\ \Sigma_2) \tag{139}$$

where ρ_3 extends ρ_2. Similarly, by Lemma D.62 and Lemma D.60 on (136), (137), and (139),

$$\mathit{sub}(\Gamma)\ \Sigma\ (\alpha \leq \neg\zeta) \vDash \rho_4(\Sigma_1\ \Sigma_2\ \Sigma_3) \tag{140}$$

where ρ_4 extends ρ_3. By I-Region on (130), (131), (135), and (136),

$$\Gamma,\ \zeta \vdash \textbf{region}\ x\ \textbf{in}\ t' :^{\gamma^n} \beta^n \Rightarrow \Sigma_1\ \Sigma_2\ \Sigma_3 \tag{141}$$

Let $\rho' = [\neg\zeta/\alpha] \circ \rho_4$. (140) implies $\mathit{sub}(\Gamma)\ \Sigma \vDash \rho'(\Sigma_1\ \Sigma_2\ \Sigma_3)$. Assume that $\epsilon, \Sigma_0, \mathit{sub}(\Gamma) \vdash \Sigma_1\ \Sigma_2\ \Sigma_3 \Rightarrow \Sigma'$, we conclude by Lemma D.62 and Lemma D.60.

□

Theorem D.65 (Completeness of type inference). *Given definitions $\mathcal{D}$ **wf**, if $\vdash t : \mathcal{T}\ !\ \bot$, then $\vdash t : \mathcal{S}\ !\ \varphi \Rightarrow \Xi$, $\vdash \Xi \Rightarrow \Sigma$, and there exists some type variable substitution ρ, such that $\epsilon \vDash \rho(\Sigma)$, $\vdash \rho(\mathcal{S}) \leq^{\forall} \mathcal{T}$, and $\vdash \rho(\varphi) \leq \bot$.*

Proof. A special case of Lemma D.64. □

Received 2025-03-25; accepted 2025-08-12