Type inference for functional and imperative dynamic languages

MICKAËL LAURENT, Charles University, Czech Republic
JAN VITEK, Czech Technical University, Czech Republic

In this paper, we formalize a type system based on set-theoretic types for dynamic languages that support both functional and imperative programming paradigms. We adapt prior work in the typing of overloaded and generic functions to support an impure λ-calculus, focusing on imperative features commonly found in dynamic languages such as JavaScript, Python, and Julia. We introduce a general notion of parametric opaque data types using set-theoretic types, enabling precise modeling of mutable data structures while promoting modularity, clarity, and readability. Finally, we compare our approach to existing work and evaluate our prototype implementation on a range of examples.

CCS Concepts: • **Theory of computation** → *Type structures*; *Program analysis*; • **Software and its engineering** → *Polymorphism.*

Additional Key Words and Phrases: dynamic language, imperative language, functional language, type inference, static typing, polymorphism, set-theoretic types, semantic subtyping

1 Introduction

Dynamic languages usually combine many features that make them challenging to statically type. In particular, containers like lists or dictionaries can contain heterogeneous data, and functions may exhibit different behaviors depending on the type of the data they are called on. Set-theoretic types are particularly suited to express the static behavior of programs written in such languages, as they provide:

- union, intersection, negation, products, extensible records, and recursive types to characterize heterogeneous collections,
- first class arrow types which, combined with the intersection, can express overloaded behaviors of functions (*ad-hoc polymorphism*),
- *parametric polymorphism* necessary to type generic containers and functions,
- a decidable semantic subtyping relation, well suited to structural typing (or *duck-typing*),
- a decidable constraint solving procedure (*tallying*) to instantiate type variables.

Some dynamic languages use ideas from set-theoretic types in their specification: this is the case, to some extent, of Erlang, Elixir, Python, or Ballerina. However, they do not provide a formalization of types and do not explain how to implement a static type-checker, even though some type-checkers have been developed or are in development (e.g. eqWAlizer and etylizer for Erlang, Mypy and Pyright for Python). This paper aims to provide a general formalism and a reference implementation for statically typing dynamic languages using set-theoretic types and semantic subtyping.

The typing of generic and overloaded functions in the context of a pure functional language has already been treated in [Castagna et al. 2024b]. For instance, consider the code below:

```
let filter (f:('a -> bool) & ('b -> false)) (l:[('a|'b)*]) =
  match l with
  | [] -> []
  | e::l -> if f e then e::(filter f l) else filter f l
  end
```

Authors' Contact Information: Mickaël Laurent, mickael.laurent@matfyz.cuni.cz, Charles University, Prague, Czech Republic; Jan Vitek, vitekj@me.com, Czech Technical University, Prague, Czech Republic.

```
let test_filter = filter (fun x -> (x is int)) [42 ; Null ; true ; 33]
```

It defines a function `filter` taking as input a predicate `f` and a list `l`, and returning a list that contains only the elements of `l` that satisfy `f`. For this function, the following type can be inferred:

$$\forall \alpha, \beta.\quad (\alpha \to \texttt{bool}) \wedge (\beta \to \texttt{false}) \to [(\alpha \vee \beta)*] \to [(\alpha \setminus \beta)*]$$

This type mixes both parametric polymorphism (by quantifying over the type variables α and β) and ad-hoc polymorphism (by using an intersection $\wedge$ for the type of the predicate `f`). It states that the parameter `f` is a function that must cover a domain α, for which it returns a boolean, and it must also cover a domain β for which we know it returns `false` (this domain β can be instantiated with the empty type if we don't have such guarantees about `f`). Moreover, the parameter `l` must be a list composed of any number of elements of type $\alpha \vee \beta$ (list types are defined using recursive types, and support the regular expression operators). In return, the function `filter` will return a list composed of any number of elements of type $\alpha \setminus \beta$. Note that, in order to be able to derive such a type, the type system must perform *type narrowing* (or *occurrence typing* [Tobin-Hochstadt and Felleisen 2010]) in order to deduce that the occurrence of `e` in the `then` branch has type $\alpha \setminus \beta$.

Deriving this type for the `filter` function is interesting because it captures the fact that the resulting list only contains elements from `l` that may satisfy `f`. When typing the definition `test_filter`, the type $(\texttt{int} \to \texttt{true}) \wedge (\neg\texttt{int} \to \texttt{false})$ can be inferred for the anonymous function `(fun x -> (x is int))`. Together with the precise typing of `filter`, this makes it possible to derive the type $[\texttt{int}*]$ for `test_filter`. This pattern, where a heterogeneous list is filtered to keep only elements of a given type, is quite idiomatic of dynamic languages (this happens, for instance, when filtering out `null` elements from a list). The type system of [Castagna et al. 2024b] is capable of deriving such types for these two definitions, but it crucially relies on the fact that the source language is pure. In particular, it requires that, if a source expression e has type $t_1 \vee t_2$, then all of its occurrences must reduce to values that all have type t_1 or all have type t_2.

Our paper adapts and extends this work to support impure languages, and focus on the imperative features that many dynamic languages have (e.g. JavaScript, Python, R, Julia), such as mutable variables. In those languages, the `filter` function, written above in a functional style, could also be written in an imperative style:

```
let filter_imp (f:('a -> bool) & ('b -> false)) (arr:array('a|'b)) =
  let res = array () in
  let mut i = 0 in
  while i < (len arr) do
    let e = arr[i] in
    if f e do push res e end ;
    i := i + 1
  end ;
  return res
```

Though this function achieves a similar goal as `filter`, it does it in a very different way: instead of building its result through a recursive call, `filter_imp` builds it by mutating the resizable array `res` successively using control flow (here, a `while` loop). In this code, `array` is an opaque data type used to model resizable arrays. It takes as parameter a type that represents the values that the array can contain. For this `filter_imp` function, our type system is able to derive the following type:

$$\forall \alpha, \beta, \gamma.\quad (\alpha \to \texttt{bool}) \wedge (\beta \to \texttt{false}) \to \texttt{array}(\alpha \vee \beta) \to \texttt{array}(\alpha \setminus \beta \vee \gamma)$$

As in the `filter` example, our type system is able to infer that the resulting array does not contain elements that do not satisfy the predicate `f`. Note the presence of an additional type variable γ in the type of the result, $\texttt{array}(\alpha \setminus \beta \vee \gamma)$. This union with γ is necessary for the type of the result to be as general as possible. Indeed, the parameter α of the type $\texttt{array}(\alpha)$ is invariant: $\texttt{array}(t)$ is a subtype of $\texttt{array}(s)$ if and only if t and s are equivalent. Consequently, assuming we have in our environment an array `arr` of type $\texttt{array}(\texttt{int})$, the expression `push arr true` is untypeable as `arr` cannot be converted to $\texttt{array}(\texttt{int} \vee \texttt{bool})$ by subtyping. However, as the function `filter_imp` returns a new array (distinct from the one given as input), we should be able to enlarge the type of its content to suit our needs: we expect `push (filter_imp f arr) true` (for some predicate `f`) to be typeable. This is possible with the type inferred for `filter_imp` by instantiating γ with `bool`.

Our paper is the first to formalize a polymorphic type system based on set-theoretic types for imperative languages. Our contributions are the following:

- A formal type system for a non-deterministic λ-calculus (Section 2) incorporating: type narrowing via union elimination, prenex parametric polymorphism *à la Hindley-Milner*, and ad-hoc polymorphism via intersection introduction. We first present it declaratively and prove type safety (Section 3.1), followed by an equivalent algorithmic version that introduces annotations (Section 3.2).
- A type inference capable of reconstructing the domain(s) of generic and overloaded functions using a tallying-based approach[1] (Section 3.3). A discussion about some practical aspects and key optimizations is also available in Appendix D.3.
- A notion of opaque data types, the associated subtyping relations, and their proofs of soundness. This allows us to represent mutable data structures such as references, arrays, and dictionaries (Section 4.1) thanks to the value restriction (Section 4.2).
- MLsem, a prototype implementation of our type system for a functional-imperative language with a type-case construct. It supports several extensions, such as pattern matching and recursive functions (Section 5), and performs program transformations in order to achieve flow-sensitive typing in the presence of mutable variables and imperative control flow primitives (`break`, `return`, etc.). This prototype serves as a proof of concept to demonstrate that: (i) even though our core calculus is mostly functional (for instance, it has no builtin notion of mutable variable), it can still be used to encode imperative features, and (ii) our type system is sufficiently powerful and general to type these encoded programs.

2 Types and language

2.1 Types

Our type system relies on the set-theoretic types theory, introduced by [Frisch 2004] and later extended with type variables [Castagna et al. 2014].

Definition 2.1 (Set-theoretic types). The set $\mathcal{T}$ of *set-theoretic types* is the set of regular and contractive terms coinductively defined by the following grammar:

$$\textbf{Types} \quad t \quad ::= \quad b \mid \alpha \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

where $b \in \mathcal{B}$ is a base type and $\alpha \in \mathcal{V}$ is a type variable. The notation $t_1 \setminus t_2$ is a syntactic sugar for $t_1 \wedge \neg t_2$. When writing a term, we use the following precedence (by decreasing priority): $\neg$, $\setminus$, $\wedge$, $\vee$, $\times$, $\to$.

[1] Tallying is similar to unification, but with subtyping constraints instead of syntactic equivalence [Castagna et al. 2014].

The set $\mathcal{B}$ of base types and the set $\mathcal{V}$ of type variables are fixed. Types are ranged over by meta-variables t and s. For each constant of the language, $\mathcal{B}$ contains the associated *singleton type*. It also contains some usual (non-singleton) types like `bool` and `int`.

As they are defined coinductively, types can be infinite trees, provided that they satisfy the constraints of regularity and contractivity explained below. This yields a definition of equirecursive types that does not require explicit binders for recursion.

A term is said *regular* if it only has a finite number of distinct subterms, and *contractive* if every infinite branch goes through an infinite number of arrows and products ($\rightarrow$ and $\times$). The contractivity constraint ensures that every type has a meaningful interpretation: for instance, it prevents from expressing types such as the one satisfying the equation $t = \neg t$. The regularity constraint ensures decidability of the subtyping relation.

The type $\mathbb{0}$ is a special type that is not inhabited by any value, and is the subtype of all types. Conversely, the type $\mathbb{1}$ is the supertype of all types.

The $\times$ constructor is used to type pairs of our language. Intuitively, it corresponds to the Cartesian product of two types. In particular, the product $\mathbb{1} \times \mathbb{1}$ is the supertype of all pairs: any well-typed pair can be typed with $\mathbb{1} \times \mathbb{1}$.

The $\rightarrow$ constructor is used to type functions (i.e., λ-abstractions). Intuitively, a λ-abstraction has type $t_1 \rightarrow t_2$ if and only if it accepts as argument a value of type t_1, in which case either it yields a value of type t_2 or it diverges. The type $\mathbb{0} \rightarrow \mathbb{1}$ is the supertype of all functions.

Type variables α can be used by the type system to handle *parametric polymorphism*. However, at the level of the type algebra, type variables are not quantified: this will be handled by the type system. For any type t, we note $\text{vars}(t)$ the set of type variables occurring in t.

Definition 2.2 (Type substitution). A *type substitution* is a function $\phi : \mathcal{V} \rightarrow \mathcal{T}$ from type variables to types which is the identity everywhere except for a finite set of type variables, called its domain and denoted by $\text{dom}(\phi)$.

We use the symbol Φ to range over sets of substitutions.

Definition 2.3 (Application of a type substitution). The result of the application of a type substitution ϕ to a type t, noted $t\phi$, is the type satisfying these equations:

$$\begin{array}{lll} b\phi = b & (t_1 \times t_2)\phi = (t_1\phi) \times (t_2\phi) & (t_1 \vee t_2)\phi = (t_1\phi) \vee (t_2\phi) \\ \mathbb{1}\phi = \mathbb{1} & (t_1 \rightarrow t_2)\phi = (t_1\phi) \rightarrow (t_2\phi) & (t_1 \wedge t_2)\phi = (t_1\phi) \wedge (t_2\phi) \\ \mathbb{0}\phi = \mathbb{0} & \alpha\phi = \phi(\alpha) & (\neg t)\phi = \neg(t\phi) \end{array}$$

Note that this system of equations has a unique solution, and this solution is a type as defined by Definition 2.1 (in particular, it is contractive and regular).

A decidable subtyping relation $\leq$ (usually referred to as *semantic subtyping*) can be defined for set-theoretic types (cf. Appendix A for more details). For this presentation, it suffices to consider that ground types (i.e., types with no variables) are interpreted as sets of values that have that type in an interpretation domain $\mathcal{D}$, and that subtyping is set containment. Type connectives (i.e., union, intersection, negation) are interpreted as the corresponding set-theoretic operators. For non-ground types, the subtyping relation is preserved by type-substitutions: if $t_1 \leq t_2$, then $t_1\phi \leq t_2\phi$ for every type-substitution ϕ. We note $\simeq$ the semantic equivalence: $t_1 \simeq t_2$ if and only if $t_1 \leq t_2$ and $t_2 \leq t_1$.

2.2 Language syntax

Definition 2.4 (Syntax of the core language). The *expressions* and *values* of our core language are the finite terms produced by the following grammar:

$$\begin{array}{llll} \textbf{Expression} & e & ::= & c \mid x \mid \lambda x.e \mid e\ e \mid (e,e) \mid \pi_i e \mid (e{\in}\tau)\ ?\ e : e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e \oplus e \\ \textbf{Value} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array}$$

where τ is a test type, that is, a ground type that does not feature any arrow except $\mathbb{0} \to \mathbb{1}$.

Definition 2.5 (Test type). A *test type* is a type produced by the following grammar:

$$\textbf{Test Type} \quad \tau \quad ::= \quad b \mid \mathbb{0} \to \mathbb{1} \mid \tau \times \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg\tau \mid \mathbb{0} \mid \mathbb{1}$$

Expressions of our language are λ-expressions with constants c, variables x, λ-abstractions $\lambda x.e$, applications $e\ e$, pairs (e, e), pair projections $\pi_i e$, type-cases $(e{\in}\tau)\ ?\ e : e$, let-bindings $\texttt{let}\ x = e\ \texttt{in}\ e$, and non-deterministic choices $e \oplus e$.

A type-case $(e_0{\in}\tau)\ ?\ e_1 : e_2$ is a dynamic type test that evaluates either e_1 or e_2 depending on the type of the value resulting from the evaluation of e_0. Type-cases cannot test arbitrary types but just ground types (i.e., types without type variables occurring in them) where the only arrow type that can occur is $\mathbb{0} \to \mathbb{1}$ (the supertype of all functions). This means that type-cases cannot distinguish functions that have a type $s \to t$ from those that have not. This restriction is necessary in order to give our language a proper semantics: having full type tests of the form $v \in t$ would entail that we must be able to check at run-time the types of λ-abstractions. While it is possible in languages such as CDuce, where λ-abstractions are monomorphic and are decorated with their static type, the problem is more complex for us as we consider unannotated λ-abstractions that may be typed in many different ways. We believe this restriction to be acceptable since the dynamic languages we want to model can test at run-time whether a value is a function but cannot have more precise information.

The role of let-bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ in our core calculus is twofold (cf. Section 3): (i) they will be used to implement let-polymorphism [Milner 1978], and (ii) they will play a central role in our implementation of type narrowing.

Finally, non-deterministic choices $e \oplus e$ are added to our calculus in order to model some behaviors of imperative languages, such as IO. It addresses limitations of prior work [Castagna et al. 2024b], where the soundness of the type system crucially relies on the purity of the language. Our type system does not: Theorem 3.3 (type safety) shows that our system is safe even in the presence of non-deterministic behaviors.

2.3 Language dynamic semantics

The reduction semantics for our expressions is the usual call-by-value reduction semantics, together with the context rules that implement a leftmost-outermost reduction strategy. Our reduction semantics is non-deterministic (several reduction steps may be applicable) because of choice expressions $e \oplus e$. This semantics is formalized in Figure 1.

The relation $v \in \tau$ determines whether a *value* is of a given type or not. Note that $\mathrm{typeof}(v)$ maps every λ-abstraction to $\mathbb{0} \to \mathbb{1}$ and, thus, dynamic type tests do not depend on static type inference, nor do they require subtyping to be performed at runtime. This approximation is allowed by the restriction on arrow types in the types τ used in type-cases.

Given a reduction step relation $\rightsquigarrow$, we write $e \rightsquigarrow^* e'$ when there exists a sequence of $\rightsquigarrow$ steps of any length from e to e' (we say that e can reduce to e'). We write $e \rightsquigarrow^\infty$ when there exists a sequence of $\rightsquigarrow$ steps starting from e and that can be prolonged indefinitely (we say that e can diverge).

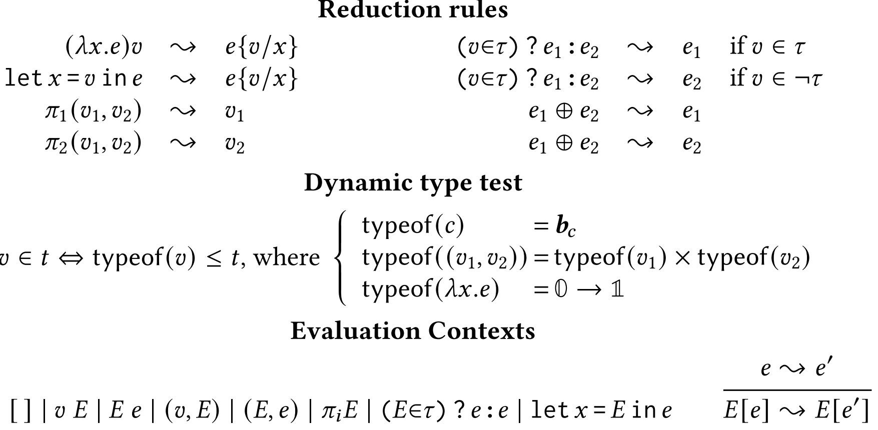

Fig. 1. Semantics of the source language

3 Static semantics

3.1 Declarative type system

In order to implement parametric polymorphism, we reuse the notion of type schemes used in Hindley-Milner type systems. Intuitively, the type scheme $\forall\vec{\alpha}.t$, where $\vec{\alpha}$ is a set of type variables, is a syntactic object that denotes the infinite set of types $t\phi$ for all type substitutions ϕ such that $\mathrm{dom}(\phi) \subseteq \vec{\alpha}$. Note that these types $t\phi$ may still contain (unquantified) type variables. We use the symbol σ to range over type schemes.

Definition 3.1 (Type environment). A *type environment* Σ is a finite mapping from variables of our language to type schemes. We note $(\Sigma, x : \sigma)$ the extension of Σ that maps x to σ, with the condition that x is not already in the domain of Σ.

Definition 3.2 (Free type variables). The *free type variables* $\mathrm{vars}(\sigma)$ of a type scheme σ and the *free type variables* $\mathrm{vars}(\Sigma)$ of a type environment Σ are defined as follows:

$$\mathrm{vars}(\forall\vec{\alpha}.t) \stackrel{\text{def}}{=} \mathrm{vars}(t) \setminus \vec{\alpha}$$
$$\mathrm{vars}(\Sigma) \stackrel{\text{def}}{=} \bigcup_{(x:\sigma)\in\Sigma} \mathrm{vars}(\sigma)$$

The disjointness between two sets of type variables $\vec{\alpha}_1$ and $\vec{\alpha}_2$ is noted $\vec{\alpha}_1 \# \vec{\alpha}_2$.

Our type system is formalized in full in Figure 2. Judgments are of the form $\Sigma \vdash e : t$. Note that a judgment derives a type t and not a type scheme σ (type schemes only appear in type environments).

A variable x is typed by the rule [Var] that retrieves the type scheme associated to x in the environment Σ. As said before, a type scheme $\forall\vec{\alpha}.t$ intuitively denotes the infinite set of types $t\phi$ (with $\mathrm{dom}(\phi) \subseteq \vec{\alpha}$), and the [Var] rule can derive any of those instances.

A non-deterministic choice $e_1 \oplus e_2$ is typed by typing both e_1 and e_2 and returning (the union of) their type. Note that we do not need to write an explicit union in the [Choose] rule as unions can already be introduced by subtyping ([$\leq$] rule): a type t_1 can be turned into $t_1 \vee t_2$ as $t_1 \leq t_1 \vee t_2$. Likewise, we do not need an intersection elimination rule: a type $t_1 \wedge t_2$ can be turned into t_1 (or into t_2) using the [$\leq$] rule.

$$[\textsc{Var}]\ \frac{\Sigma(x) = \forall\vec{\alpha}.t}{\Sigma \vdash x : t\phi}\ \mathrm{dom}(\phi) \subseteq \vec{\alpha} \qquad [\textsc{Const}]\ \frac{}{\Sigma \vdash c : \boldsymbol{b_c}} \qquad [\textsc{Choose}]\ \frac{\Sigma \vdash e_1 : t \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 \oplus e_2 : t}$$

$$[\textsc{Let}]\ \frac{\Sigma \vdash e_1 : s \qquad (\forall i \in I) \quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t}{\Sigma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \ : t}\ \begin{array}{l} s \leq \bigvee_{i \in I} s_i \\ \vec{\alpha}\#\mathrm{vars}(\Sigma), \quad \forall i \in I.\ \vec{\alpha}\#\mathrm{vars}(s_i) \end{array}$$

$$[\to\mathrm{I}]\ \frac{\Sigma, x : s \vdash e : t}{\Sigma \vdash \lambda x.e : s \to t} \qquad [\to\mathrm{E}]\ \frac{\Sigma \vdash e_1 : t_1 \to t_2 \qquad \Sigma \vdash e_2 : t_1}{\Sigma \vdash e_1 e_2 : t_2}$$

$$[\times\mathrm{I}]\ \frac{\Sigma \vdash e_1 : t_1 \qquad \Sigma \vdash e_2 : t_2}{\Sigma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\mathrm{E}_1]\ \frac{\Sigma \vdash e : t_1 \times t_2}{\Sigma \vdash \pi_1 e : t_1} \qquad [\times\mathrm{E}_2]\ \frac{\Sigma \vdash e : t_1 \times t_2}{\Sigma \vdash \pi_2 e : t_2}$$

$$[\in]\ \frac{\Sigma \vdash e : s \qquad s \wedge \tau \not\simeq \mathbb{0} \Rightarrow \Sigma \vdash e_1 : t \qquad s \setminus \tau \not\simeq \mathbb{0} \Rightarrow \Sigma \vdash e_2 : t}{\Sigma \vdash (e {\in} \tau)\, ?\, e_1 : e_2 : t}$$

$$[\wedge]\ \frac{\Sigma \vdash e : t_1 \qquad \Sigma \vdash e : t_2}{\Sigma \vdash e : t_1 \wedge t_2} \qquad [\leq]\ \frac{\Sigma \vdash e : t}{\Sigma \vdash e : t'}\ t \leq t'$$

Fig. 2. Declarative type system

The arrow and product constructors have introduction and elimination rules. Note that, in the case of $[\to\mathrm{I}]$, the variable x must not already be in $\mathrm{dom}(\Sigma)$ in order for $\Sigma, x : s$ to be defined, but α-renaming can be applied implicitly on expressions whenever needed.

Type-cases are handled by the rule $[\in]$, which can avoid typing a branch if this branch cannot possibly be taken, that is, if the type of the tested expression e is disjoint with the test type captured by the branch (τ for the first branch, $\neg\tau$ for the second branch).

The rules for intersection ($[\wedge]$) and subtyping ($[\leq]$) are the classical ones.

The most interesting rule is the rule for let-bindings, [Let]. A first feature of this rule is that it can decompose the type s of e_1 into a disjunction $\bigvee_{i \in I} s \wedge s_i$ (the side-condition $s \leq \bigvee_{i \in I} s_i$ ensures that this decomposition covers s), and then type e_2 under the environment $\Sigma, x : s \wedge s_i$ for each $i \in I$ independently. This is a restricted form of the *union elimination rule* [MacQueen et al. 1986], used to implement *type narrowing* as it is done in [Castagna et al. 2024b], but that only applies on let-definitions. For instance, when typing $\mathtt{let}\, x = f\ 42\ \mathtt{in}\ (x {\in} \mathtt{int})\, ?\, x + 1 : x$ with $f : \mathbb{1} \to \mathbb{1}$, one can choose to decompose the type of x into $\mathtt{int}$ and $\neg\mathtt{int}$. Each case will be considered independently: the type-case $(x {\in} \mathtt{int})\, ?\, x + 1 : x$ will first be typed under the hypothesis $x : \mathtt{int}$, in which case only the first branch is typed (successfully, as $x : \mathtt{int}$), and a second time under the hypothesis $x : \neg\mathtt{int}$, in which case only the second branch is typed. The restriction of union-elimination on let-definitions, in addition to making the implementation of the type system simpler, is justified by the presence of non-determinism in our language. Indeed, in a pure language, two occurrences of the same expression will reduce to the same value: thus it is sound to derive the type $\mathtt{true} \times \mathtt{true} \vee \mathtt{false} \times \mathtt{false}$ for the expression $(f\ 42, f\ 42)$ under the environment $f : \mathbb{1} \to \mathtt{bool}$. However, this is not true anymore in our setting, as the function f could be non-deterministic (for instance it could be defined as $\lambda x.\mathtt{true} \oplus \mathtt{false}$). In order to regain soundness, we restrict the union elimination rule to only decompose the type of variables (which is sound as two occurrences of the same variable hold the same value), instead of arbitrary expressions.

A second aspect of the [Let] rule is that it can generalize some type variables $\vec{\alpha}$ of s. There are two restrictions on $\vec{\alpha}$: (i) type variables in $\vec{\alpha}$ must not be bound to the environment Σ, and (ii) type variables in $\vec{\alpha}$ must not appear in any of the s_i used to decompose the type s. The first restriction is standard and prevents the generalization of a type variable that may be bound to the parameter of a λ-abstraction we are currently typing. For instance, when typing the identity $\lambda x.\mathtt{let}\, y = x\, \mathtt{in}\, y$ for the domain $x : \alpha$, it would be unsound to associate the type scheme $\forall\alpha.\alpha$ to y as it would allow deriving the type $\alpha \to \mathbb{0}$ for this function. The second restriction prevents unsound type decompositions. For instance, it would be unsound to decompose the type $\mathbb{1}$ into the two types schemes $\forall\alpha.\alpha$ and $\forall\alpha.\neg\alpha$.

Theorem 3.3 (Type safety). *For every expression e and test type τ, if $\varnothing \vdash e : \tau$, then for any e' such that $e \rightsquigarrow^* e'$, we have either $e' \rightsquigarrow^* v$ for some $v \in \tau$ or $e' \rightsquigarrow^\infty$.*

This theorem states that if an expression is well-typed (of test type τ), then all possible reduction sequences either diverge or reduce to a value of type τ. Notice that this type safety theorem holds for a test type τ instead of an arbitrary type t. Indeed, only test types are guaranteed to be preserved by reduction: in particular, because of the type decomposition performed in the [Let] rule, a negative arrow type (e.g. $\neg(\mathtt{int} \to \mathtt{int})$) may be derivable for an expression before reduction, but not after. An example is given in Appendix B, followed by the proof of Theorem 3.3. Even though type preservation holds only for test types, we still retain strong guarantees about function behavior. While we cannot in general ensure that the type of a function f itself is preserved, we can guarantee preservation for every application of f. For example, if f has type $\mathtt{int} \to \mathtt{int}$, then f 42 has type `int`, which is a test type and is thus preserved by reduction.

3.2 Algorithmic type system

The declarative type system defined in Section 3.1 is not algorithmic for two reasons: (i) the [$\leq$] and [$\wedge$] rules are not *syntax directed* (they can apply on any expression), and (ii) the rules [Var], [$\to$I], [Let] and [$\leq$] are not *analytic*, meaning that some inputs of their premises cannot be determined by the inputs of their conclusion.

In order to make the type system algorithmic, we apply two ideas from prior work. First, we get rid of the [$\leq$] rule by inlining it in other rules whenever needed. This is standard fares when using set-theoretic types [Castagna et al. 2015; Frisch 2004]. Second, we give an additional input to the type system, consisting in an annotation tree that specifies (i) when to apply the [$\wedge$] rule (thus making the system syntax-directed, or rather, annotation-driven), and (ii) for each rule, the information needed to make it analytic: the expected result type for [$\to$E], [$\times$E$_1$] and [$\times$E$_2$], which substitution to use for [Var], which domain to use for [$\to$I], and which type decomposition to perform for [Let]. Note that this annotation tree is not meant to be provided by the programmer, but to be inferred from an expression using a reconstruction algorithm inspired by algorithm $\mathcal{W}$ [Damas and Milner 1982] (cf. Section 3.3).

$$\begin{array}{lll} \textbf{Annotations} & a ::= & \varnothing \mid \mathtt{var}(\phi) \mid \mathtt{let}\ (a, \{(t,a),\dots,(t,a)\}) \mid @(a,a,t) \mid \pi(a,t) \mid \times(a,a) \\ & & \mid \in(a,b,b) \mid \lambda(t,a) \mid \oplus(a,a) \mid \bigwedge(\{a,\dots,a\}) \\ \textbf{Branch annotations} & b ::= & \mathtt{type}(a) \mid \mathtt{skip} \end{array}$$

The use of separate annotations, instead of annotations directly inserted in the expression to type, is justified by the tree structure of our derivations: for instance, when deriving an intersection type for an overloaded function, the same λ-abstraction will be typed multiple times but with different domains (and thus, different annotations). This idea of using a separate annotation tree is taken from [Castagna et al. 2024b], though the shape of the annotations have been adapted to our setting. In particular, the algorithmic type system from this prior work applies to expressions in a *Maximal*

Sharing Canonical form, while ours directly applies on the source language. Additionally, their type system can derive types that contain polymorphic (i.e. universally quantified) type variables, while ours can only derive monomorphic types (type schemes only appear in environments). This is necessary to implement the value restriction (cf. Section 4.2) that only allows generalizing (and therefore instantiating) values. A consequence is that our algorithmic type system must perform instantiations directly on variables, while [Castagna et al. 2024b] instantiates polymorphic type variables when typing destructors (applications, projections, etc.).

The full algorithmic type system is formalized in Figure 3. Judgments have the form $\Sigma \vdash_{\mathscr{A}} [e \mid a] : t$, where Σ is the current type environment, e is the expression to type, a is the associated annotation tree, and t is the derived type.

$$[\textsc{Const-A}]\ \frac{}{\Sigma \vdash_{\mathscr{A}} [c \mid \varnothing] : \boldsymbol{b}_c} \qquad [\textsc{Var-A}]\ \frac{\Sigma(x) = \forall\vec{\alpha}.t}{\Sigma \vdash_{\mathscr{A}} [x \mid \mathtt{var}(\phi)] : t\phi}\ \mathrm{dom}(\phi) \subseteq \vec{\alpha}$$

$$[\textsc{Let-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e_1 \mid a] : s \qquad \vec{\alpha} = \mathrm{vars}(s) \setminus (\mathrm{vars}(\Sigma) \cup \bigcup_{i\in I} \mathrm{vars}(s_i)) \qquad (\forall i \in I)\quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash_{\mathscr{A}} [e_2 \mid a_i] : t_i}{\Sigma \vdash_{\mathscr{A}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \mathtt{let}\ (a, \{(s_i, a_i)\}_{i\in I})] : \bigvee_{i\in I} t_i}\ s \leq \bigvee_{i\in I} s_i$$

$$[{\to}\text{I-A}]\ \frac{\Sigma, x : s \vdash_{\mathscr{A}} [e \mid a] : t}{\Sigma \vdash_{\mathscr{A}} [\lambda x.e \mid \lambda(s, a)] : s \to t} \qquad [{\to}\text{E-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e_1 \mid a_1] : t_1 \quad \Sigma \vdash_{\mathscr{A}} [e_2 \mid a_2] : t_2}{\Sigma \vdash_{\mathscr{A}} [e_1 e_2 \mid @(a_1, a_2, t)] : t}\ t_1 \leq t_2 \to t$$

$$[{\times}\text{I-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e_1 \mid a_1] : t_1 \qquad \Sigma \vdash_{\mathscr{A}} [e_2 \mid a_2] : t_2}{\Sigma \vdash_{\mathscr{A}} [(e_1, e_2) \mid \times(a_1, a_2)] : t_1 \times t_2} \qquad [{\times}\text{E}_1\text{-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e \mid a] : s}{\Sigma \vdash_{\mathscr{A}} [\pi_1 e \mid \pi(a, t)] : t}\ t \leq (s \times \mathbb{1})$$

$$[{\in}\text{-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e \mid a] : s \qquad b_1 = \mathtt{skip} \Rightarrow s \leq \neg\tau \qquad b_2 = \mathtt{skip} \Rightarrow s \leq \tau \qquad \Sigma \vdash_{\mathscr{A}} [e_1 \mid b_1] : t_1 \qquad \Sigma \vdash_{\mathscr{A}} [e_2 \mid b_2] : t_2}{\Sigma \vdash_{\mathscr{A}} [(e{\in}\tau)\, ?\, e_1 : e_2 \mid \in(a, b_1, b_2)] : t_1 \vee t_2}$$

$$[\textsc{Choose-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e_1 \mid a_1] : t_1 \quad \Sigma \vdash_{\mathscr{A}} [e_2 \mid a_2] : t_2}{\Sigma \vdash_{\mathscr{A}} [e_1 \oplus e_2 \mid \oplus(a_1, a_2)] : t_1 \vee t_2} \qquad [\wedge\text{-A}]\ \frac{(\forall i \in I)\quad \Sigma \vdash_{\mathscr{A}} [e \mid a_i] : t_i}{\Sigma \vdash_{\mathscr{A}} [e \mid \bigwedge(\{a_i\}_{i\in I})] : \bigwedge_{i\in I} t_i}\ I \neq \emptyset$$

$$[\textsc{Skip-A}]\ \frac{}{\Sigma \vdash_{\mathscr{A}} [e \mid \mathtt{skip}] : \mathbb{0}} \qquad [\textsc{Type-A}]\ \frac{\Sigma \vdash_{\mathscr{A}} [e \mid a] : t}{\Sigma \vdash_{\mathscr{A}} [e \mid \mathtt{type}(a)] : t}$$

Fig. 3. Algorithmic type system

Branches of type-cases are typed with the rules [Skip-A] and [Type-A], which derive judgments of the form $\Sigma \vdash_{\mathscr{A}} [e \mid b] : t$. A branch annotated with `skip` is given the type $\mathbb{0}$ without even looking the associated expression, and the rule for type-cases [∈-A] is responsible for checking that only unreachable branches are annotated with `skip`. The [Let-A] rule generalizes as many type variables as possible. The [$\times$E$_2$] rule is omitted, but is similar to the [$\times$E$_1$] one.

This algorithmic type system is equivalent to the declarative one, meaning that an expression is typeable with the declarative type system if and only if there exists an annotation tree that makes it typeable with the algorithmic type system.

Theorem 3.4 (Equivalence with the declarative type system). *Let e be an expression, Σ a type environment, and t a type.*

$$\Sigma \vdash e : t \quad \Leftrightarrow \quad \exists a, t'. \ \Sigma \vdash_{\mathcal{A}} [e \mid a] : t' \text{ and } t' \leq t$$

A proof of this theorem can be found in Appendix C.

3.3 Annotation tree reconstruction

The annotation tree used by the algorithmic type system is too complex to be provided directly by the programmer, thus it must be automatically reconstructed (though the programmer may help this process by annotating some functions with their domain). In this section, we propose an algorithm that reconstructs an annotation tree a from an expression e. This algorithm is inspired by Hindley-Milner systems, in particular algorithm $\mathcal{W}$ [Damas and Milner 1982] and its inference by unification. In our setting, however, we have semantic subtyping, and the syntactic constraints used in unification must be replaced by subtyping constraints: this is what *tallying* [Castagna et al. 2015] does.

3.3.1 Tallying. While unification consists, for two types s and t, in finding the type substitutions ϕ such that $s\phi$ and $t\phi$ are syntactically equivalent, tallying consists in finding the type substitutions ϕ such that $s\phi \leq t\phi$.

Definition 3.5 (Tallying problem). Let C be a finite set of constraints $\{(s_i \mathrel{\dot{\leq}} t_i)\}_{i \in I}$. A type substitution ϕ is a solution to the *tallying problem* C, noted $C \Vdash \phi$, if and only if $\forall i \in I.\ s_i\phi \leq t_i\phi$.

Note that tallying is usually defined with an additional input Δ that specifies the set of *monomorphic* type variables (i.e. the type variables that must not be substituted), but this is unnecessary in our setting. Solutions to a tallying problem are characterized by a principal finite set of type substitutions. We use the notation $C \Vdash \Phi$ to state that the finite set of substitutions Φ is a principal solution of the tallying instance C.

Proposition 3.6 (Principality). *For every tallying problem C, the set of all solutions can be characterized by a finite set of type substitutions Φ such that:*

$$\forall \phi \in \Phi.\ C \Vdash \phi \qquad \textit{(soundness)}$$

$$\forall \phi''.\ C \Vdash \phi'' \quad \Rightarrow \quad \exists \phi \in \Phi.\ \exists \phi'.\ \phi'' \simeq \phi' \circ \phi \qquad \textit{(completeness)}$$

An algorithm that computes a principal set of substitutions for any tallying problem is described in [Castagna et al. 2015].

3.3.2 Reconstruction algorithm. The type inference algorithm proceeds in two steps. First, an initial (and incomplete) annotation tree is generated for the expression (cf. *partial annotations* below). Then, this annotation tree is refined in a recursive process until it becomes a complete annotation tree for the algorithmic type system, or the reconstruction fails if some sub-expressions cannot be typed.

To refine an annotation tree, our reconstruction algorithm proceeds similarly to the algorithm $\mathcal{W}$ [Damas and Milner 1982]: when it enters a λ-abstraction, its parameter is typed according to the initial annotation (either with a fresh type variable, or a type annotated by the user). Then, when this parameter is used in the body (in an application or projection), a constraint is generated and solved using tallying. The difficulty in our setting stems from the nature of the tallying procedure, which returns a set of (possibly incompatible) substitutions. Contrary to algorithm $\mathcal{W}$, where unification returns at most one principal substitution and can thus be used to accumulate constraints on type variables by only going forward, our algorithm needs, when given a set of substitutions by the

tallying procedure, to backtrack "high enough" in the term (at the place where the type variables are introduced) and retries typing several times, using the newly found substitutions.

The intermediate annotations $\bar{a}$ refined by the reconstruction algorithm follow this grammar:

$$
\begin{array}{lll}
\textbf{Partial annotations} & \bar{a} ::= & a \mid \mathtt{untyp} \mid \bar{\mathtt{var}}(\phi) \mid \bar{@}(\bar{a},\bar{a},t) \mid \bar{\pi}(\bar{a},t) \mid \bar{\times}(\bar{a},\bar{a}) \\
 & & \mid \bar{\in}(\bar{a},\bar{b},\bar{b}) \mid \bar{\lambda}(t,\bar{a}) \mid \bar{\mathtt{let}}\ (\bar{a},\bar{U}) \mid \bar{\bigwedge}(\bar{I}) \\
\textbf{Branch partial annotations} & \bar{b} ::= & \bar{\mathtt{maybe}}(\bar{a}) \mid \bar{\mathtt{type}}(\bar{a}) \mid b \\
\textbf{Inter partial annotations} & \bar{I} ::= & \{\bar{a}, \dots, \bar{a}\} \\
\textbf{Union partial annotations} & \bar{U} ::= & \{(t,\bar{a}), \dots, (t,\bar{a})\}
\end{array}
$$

The initial partial annotation for an expression e is generated as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{init}(c) & = & \varnothing & \mathsf{init}(\lambda x.e) & = & \bar{\lambda}(\alpha, \mathsf{init}(e)) \text{ with } \alpha \text{ fresh} \\
\mathsf{init}(x) & = & \bar{\mathtt{var}}(\phi) \text{ with } \phi \text{ fresh} & \mathsf{init}(e_1 e_2) & = & \bar{@}(\mathsf{init}(e_1), \mathsf{init}(e_2), \alpha) \text{ with } \alpha \text{ fresh} \\
\mathsf{init}((e_1, e_2)) & = & \bar{\times}(\mathsf{init}(e_1), \mathsf{init}(e_2)) & \mathsf{init}(\pi_i e) & = & \bar{\pi}(\mathsf{init}(e), \alpha) \text{ with } \alpha \text{ fresh}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{init}((e_0 \in \tau)\ ?\ e_1 : e_2) & = & \bar{\in}(\mathsf{init}(e_0), \bar{\mathtt{maybe}}(\mathsf{init}(e_1)), \bar{\mathtt{maybe}}(\mathsf{init}(e_2))) \\
\mathsf{init}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) & = & \bar{\mathtt{let}}\ (\mathsf{init}(e_1), \{(s_i, \mathsf{init}(e_2))\}_{i \in I}) \text{ where } \{s_i\}_{i\in I} = \mathsf{decomposition}(x)
\end{array}
$$

Each sub-expression x is initially annotated with a fresh renaming ϕ mapping each type variable to a fresh one (strictly speaking, such a renaming is not a valid substitution as its domain is infinite, but only a finite subdomain will be used in the typing rules so it can be implemented in a lazy way). This initial renaming will be used to decorrelate polymorphic type variables in the type of x. Abstractions $\lambda x.e$ are initially annotated with a fresh domain α, as in algorithm $\mathcal{W}$. Likewise, results of applications and projections are initially captured by a fresh type variable. Branches of a type-case are given a $\bar{\mathtt{maybe}}(.)$ annotation, meaning that it is not known yet whether the branch should be typed or skipped.

Finally, the initial annotation for the body of a let-binding relies on an oracle $\mathsf{decomposition}(x)$ that returns a type decomposition $\{s_i\}_{i\in I}$ that must be performed on the type of x. This type decomposition can be computed by an independent algorithm. Several choices are possible, depending on how pervasive and precise we want type narrowing to be (at the expense of performance and added complexity to the reconstructed types). It could be the trivial oracle $\mathsf{decomposition}(x) = \{\mathbb{1}\}$, which would result in a type system that does not perform type narrowing. Or, it could decompose the type of x by analyzing the type-cases that involve x in the body e_2, similarly to what is done in [Castagna et al. 2024b, Section H.3]. For instance, if e_2 contains a type-case $(f\ x \in \mathtt{true})\ ?\ \dots : \dots$ and our context Σ contains a binding $(f : (\mathbb{1} \to \mathbb{1}) \wedge (\alpha \to \mathtt{false}))$, then we can deduce that x has the type $\neg\alpha$ in the first branch, and thus decompose the type of x using the partition $\{\alpha, \neg\alpha\}$.

We now describe an annotation refinement algorithm that transforms these initial partial annotations into valid annotations for the algorithmic type system. It is presented as a system of deduction rules. Judgments have the form $\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}$, where $\mathbb{R}$ is a result, of the form:

$$\textbf{Result}\ \mathbb{R} ::= \mathtt{Ok}(a, t) \mid \mathtt{Fail} \mid \mathtt{Subst}(\Phi, \bar{a}, \bar{a})$$

Ok(a, t) means that the reconstruction has succeeded, the annotation it produced is a and the associated type derived by the algorithmic type system is t.

Fail means that the reconstruction failed (either the expression is untypeable, or it is typeable but the reconstruction algorithm failed to find a suitable annotation tree).

Subst$(\Phi, \bar{a}_1, \bar{a}_2)$ means that some substitutions should be applied to the current environment Σ before resuming the reconstruction. More precisely, the reconstruction algorithm should be called again several times: (i) on the current environment Σ using partial annotation $\bar{a}_2$ (*general case*), and (ii) for every substitution $\phi \in \Phi$, on the environment $\Sigma\phi$ using partial annotation $\bar{a}_1\phi$.

Formally, our algorithm is composed of two judgments: (i) a structural judgment $\Sigma \vdash_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}$ with an associated set of structural rules, which represents a reconstruction step from the partial annotation $\bar{a}$, and (ii) a threading judgment $\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}$ with an associated set of threading rules, whose only job is to backtrack appropriately whenever a set of substitutions is returned by some tallying problem. Both sets of rules are mutually recursive: whenever a structural rule needs to type a sub-expression, it does so by calling a threading rule, giving the algorithm the opportunity to backtrack to that point ; conversely, whenever a threading rule receives an expression to type and multiple substitutions, it will use structural rules to type the expression several times and combine the results.

The complete type system features many rules (all given in Appendix D). Most of them are bureaucratic. We focus our presentation on some of the key rules which illustrate the essence of the algorithm. Some rules may overlap: they are introduced in order of decreasing priority.

Threading rules. These rules are responsible for recursively calling the reconstruction algorithm again when a $\mathtt{Subst}(\Phi, \bar{a}_1, \bar{a}_2)$ has been issued and that all substitutions in Φ have a domain disjoint with Σ (meaning that we do not need to backtrack further). In this case, the following rule applies:

$$[\textsc{Subst-R}]\ \frac{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}_1, \bar{a}_2) \qquad \Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\bar{a}_1\phi \mid \phi \in \Phi\} \cup \{\bar{a}_2\})] \Rightarrow \mathbb{R}}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}}\ \forall \phi \in \Phi.\ \mathrm{dom}(\phi) \# \mathrm{vars}(\Sigma)$$

The different annotations on which we want to call the reconstruction algorithm are regrouped within an intersection annotation $\bar{\bigwedge}(\{\bar{a}_1\phi \mid \phi \in \Phi\} \cup \{\bar{a}_2\})$. The notation $\bar{a}_1\phi$ corresponds to the annotation $\bar{a}_1$ where every type t has been replaced by $t\phi$, and every substitution ϕ' has been replaced by the substitution $\{\alpha \rightsquigarrow t\phi \mid (\alpha \rightsquigarrow t) \in \phi'\}$.

$$[\textsc{Propagate-R}]\ \frac{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}}$$

If the result is a $\mathtt{Ok}(a, t)$, a $\mathtt{Fail}$, or a $\mathtt{Subst}(\Phi, \bar{a}_1, \bar{a}_2)$ that contains a substitution $\phi \in \Phi$ not disjoint with the current environment Σ, then it is returned as is, propagating this result upward.

Structural rules. These rules perform a reconstruction step, trying to refine a partial annotation into a complete annotation for the algorithmic type system.

$$[\textsc{Ok}_1\text{-R}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e \mid a] : t}{\Sigma \vdash_{\mathcal{R}} [e \mid a] \Rightarrow \mathtt{Ok}(a, t)} \qquad [\textsc{Fail-R}]\ \frac{}{\Sigma \vdash_{\mathcal{R}} [e \mid \mathtt{untyp}] \Rightarrow \mathtt{Fail}}$$

If the annotation is already a complete annotation, then [$\textsc{Ok}_1$-R] returns $\mathtt{Ok}(a, t)$ with t being the type returned by the algorithmic type system for this annotation and expression. If the annotation is the partial annotation $\mathtt{untyp}$, then [Fail-R] returns $\mathtt{Fail}$.

$$[\textsc{Var-R}]\ \frac{\Sigma(x) = \forall\vec{\alpha}.t \qquad \Sigma \vdash_{\mathcal{R}} [x \mid \mathtt{var}(\phi|_{\vec{\alpha}})] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [x \mid \bar{\mathtt{var}}(\phi)] \Rightarrow \mathbb{R}}$$

The annotation generated for a variable is $\mathtt{var}(\phi|_{\vec{\alpha}})$: only the type variables that are polymorphic in $\Sigma(x)$ should be renamed (we recall that the substitution ϕ is initially a renaming mapping each type variable to fresh a one). The purpose of this renaming is to avoid unwanted correlations: for instance, the pair (f, f) with $f : \forall\alpha.\alpha \to \alpha$ will be typed $(\beta \to \beta) \times (\gamma \to \gamma)$ instead of

$(\alpha \to \alpha) \times (\alpha \to \alpha)$. Note that the substitution $\phi|_{\vec{\alpha}}$ in the resulting annotation may get composed with other substitutions later in the reconstruction process (cf. threading rule [Subst-R]).

$$[\to\text{I}_1\text{-R}]\ \frac{\Sigma, x : s \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Fail}}{\Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \bar{\lambda}(s, \bar{a})] \Rightarrow \texttt{Fail}} \qquad [\to\text{I}_2\text{-R}]\ \frac{\Sigma, x : s \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Subst}(\Phi, \bar{a}', \bar{a}'')}{\Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \bar{\lambda}(s, \bar{a})] \Rightarrow \texttt{Subst}(\Phi, \bar{\lambda}(s, \bar{a}'), \bar{\lambda}(s, \bar{a}''))}$$

$$[\to\text{I}_3\text{-R}]\ \frac{\Sigma, x : s \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, t) \qquad \Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \lambda(s, a)] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \bar{\lambda}(s, \bar{a})] \Rightarrow \mathbb{R}}$$

Those three rules are mostly plumbing: if reconstructing the body of the λ-abstraction yields a `Fail`, it is simply propagated. If it yields a $\texttt{Subst}(\Phi, \bar{a}_1, \bar{a}_2)$, it is also propagated while reconstructing the partial annotation. Lastly, if it yields a $\texttt{Ok}(a, t)$, the reconstruction algorithm produces the annotation $\lambda(s, a)$, which is then used to type the λ-abstraction with the algorithmic type system (the second premise in rule [$\to$I$_3$-R] will use the rule [Ok$_1$-R] presented above).

$$[\to\text{E}_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \texttt{Ok}(a_1, t_1) \qquad \Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \texttt{Ok}(a_2, t_2) \qquad \{t_1 \mathrel{\dot{\leq}} t_2 \to t\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [e_1 e_2 \mid \bar{@}(\bar{a}_1, \bar{a}_2, t)] \Rightarrow \texttt{Subst}(\Phi, @(a_1, a_2, t), \texttt{untyp})}$$

When reconstructing an application, if the two operands have already been reconstructed successfully, then [$\to$E$_4$-R] solves the constraint $t_1 \mathrel{\dot{\leq}} t_2 \to t$ (t_1 being the type of the function, t_2 of the argument, and t of the result) using tallying. It then returns $\texttt{Subst}(\Phi, @(a_1, a_2, t), \texttt{untyp})$, meaning that, for any $\phi \in \Phi$, the annotation $@(a_1\phi, a_2\phi, t\phi)$ can be used for typing the application under the context $\Sigma\phi$. The general case (corresponding to the environment Σ unmodified) is associated with the annotation `untyp` (i.e. the application cannot be typed). Note that, if the application is already typeable under the current environment Σ, then the tallying algorithm will return the identity substitution as a solution, in which case the general case associated with the annotation `untyp` is redundant because the environment Σ is already entirely covered by Φ.

$$[\times\text{E}_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \{s \mathrel{\dot{\leq}} t \times \mathbb{1}\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [\pi_1 e \mid \bar{\pi}(\bar{a}, t)] \Rightarrow \texttt{Subst}(\Phi, \pi(a, t), \texttt{untyp})}$$

$$[\times\text{E}_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \{s \mathrel{\dot{\leq}} \mathbb{1} \times t\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [\pi_2 e \mid \bar{\pi}(\bar{a}, t)] \Rightarrow \texttt{Subst}(\Phi, \pi(a, t), \texttt{untyp})}$$

Projections are similar to applications, though the constraint generated is simpler.

$$[\in_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \{s \mathrel{\dot{\leq}} \neg\tau\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{\texttt{maybe}}(\bar{a}_1), \bar{b}_2)] \Rightarrow \texttt{Subst}(\Phi, \in(a, \texttt{skip}, \bar{b}_2), \in(a, \bar{\texttt{type}}(\bar{a}_1), \bar{b}_2))}$$

$$[\in_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \{s \mathrel{\dot{\leq}} \tau\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{\texttt{maybe}}(\bar{a}_2))] \Rightarrow \texttt{Subst}(\Phi, \in(a, \bar{b}_1, \texttt{skip}), \in(a, \bar{b}_1, \bar{\texttt{type}}(\bar{a}_2)))}$$

When reconstructing a branch of a type-case for the first time, we use tallying to determine whether this branch could be skipped in some contexts. This is necessary to infer intersection types for overloaded functions. For instance, consider the expression $\lambda x.(x{\in}\texttt{int})\,?\,42 : \texttt{false}$. When reconstructing it, x first gets a fresh type α. Then, before reconstructing the first branch, the [$\in_3$-R] rule solves the constraint $\alpha \leq \neg\texttt{int}$, which yields a substitution $\phi = \{\alpha \rightsquigarrow \neg\texttt{int} \wedge \alpha\}$. It then returns $\texttt{Subst}(\{\phi\}, \in(a, \texttt{skip}, \bar{b}_2), \in(a, \bar{\texttt{type}}(\bar{a})_1, \bar{b}_2))$, which will have as effect to backtrack before

the introduction of x in the environment and insert an intersection annotation composed of two branches: (i) a first branch where x has the type $\texttt{int} \wedge \alpha$ and where the first branch of the type-case is skipped (annotation $\in(a, \texttt{skip}, \bar{b}_2)$), and (ii) a second branch where x has the type α and the first branch of the type-case is typed (annotation $\in(a, \bar{\texttt{type}}(\bar{a})_1, \bar{b}_2)$).

$$[\textsc{Let}_3\text{-R}]\ \frac{s \wedge s' \simeq \mathbb{0} \qquad \begin{array}{c}\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \texttt{Ok}(a_1, s) \\ \Sigma \vdash_{\mathcal{R}} [\texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid \bar{\texttt{let}}\ (a_1, \bar{U})] \Rightarrow \mathbb{R}\end{array}}{\Sigma \vdash_{\mathcal{R}} [\texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid \bar{\texttt{let}}\ (\bar{a}_1, \{(s', \bar{a}_2)\} \cup \bar{U})] \Rightarrow \mathbb{R}}$$

If one of the parts of the type decomposition of a let-definition is disjoint with the type of the definition e_1, then this part can be removed from the decomposition.

$$[\wedge_1\text{-R}]\ \frac{}{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\})] \Rightarrow \texttt{Fail}} \qquad [\wedge_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Fail} \qquad \Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\bar{I})] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\bar{a}\} \cup \bar{I})] \Rightarrow \mathbb{R}}$$

When refining an intersection annotation, the different annotations composing the intersection are refined sequentially. If one of them yields $\texttt{Fail}$, it is removed from the intersection (rule $[\wedge_2\text{-R}]$). If the intersection becomes empty, then the reconstruction fails (rule $[\wedge_1\text{-R}]$).

As we already mentioned, the full set of rules can be found in Appendix D. To reconstruct the type of an expression e under a context Σ, we can just derive a judgment $\Sigma \vdash^*_{\mathcal{R}} [e \mid \text{init}(e)] \Rightarrow \mathbb{R}$. If $\mathbb{R} = \texttt{Fail}$, it means our reconstruction algorithm failed to find a suitable annotation tree to type e. If $\mathbb{R} = \texttt{Ok}(a, t)$, then we know that $\Sigma \vdash_{\mathcal{A}} [e \mid a] : t$. The result $\mathbb{R}$ cannot be $\texttt{Subst}(\Phi, \bar{a}_1, \bar{a}_2)$ if the current environment does not have free type variables ($\text{vars}(\Sigma) = \emptyset$).

Incompleteness. Each time a constraint is generated, the reconstruction algorithm explores all solutions returned by the tallying algorithm, which results in an intersection type that captures the multiple behaviors the function can exhibit. Still, this reconstruction algorithm is not complete for two reasons. First, when typing an application, it may sometimes be necessary to generate a constraint that expands the type of the argument. For instance, for the application $f\ x$ where $f : (\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}$ and $x : \alpha \to \alpha$, the reconstruction algorithm generates a constraint $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool}) \to \texttt{bool} \mathrel{\dot{\leq}} \alpha \to \alpha \to \beta$, which has no solution. Typing this application would require generating a constraint where the type of x is expanded: $(\texttt{int} \to \texttt{int}) \wedge (\texttt{bool} \to \texttt{bool}) \to \texttt{bool} \mathrel{\dot{\leq}} (\alpha \to \alpha) \wedge (\alpha' \to \alpha') \to \beta$. However, there is no general way to know when (or how many times) such expansions are needed. Second, the reconstruction algorithm depends on the oracle $\text{decomposition}(x)$ that determines the type decomposition for each variable x. Though we mainly use union-elimination for type narrowing, it can also be used to capture correlation, for instance between two components of a pair: depending on the type decomposition chosen for x, the expression $\texttt{let}\, x = f\ 42\, \texttt{in}\, (x, x)$ with $f : \mathbb{1} \to \mathbb{1}$ can be typed $(\texttt{int} \times \texttt{int}) \vee (\neg\texttt{int} \times \neg\texttt{int})$, or $(\texttt{bool} \times \texttt{bool}) \vee (\neg\texttt{bool} \times \neg\texttt{bool})$, and so on. No finite type decomposition derives a type strictly smaller than all the others: our type system does not feature principal types in general, therefore the reconstruction algorithm is necessarily incomplete.

Some practical considerations about the implementation of the reconstruction algorithm are available in Appendix D.3.

4 Mutable data structures

The language formalized in Section 2 features non-determinism, but does not have mutable data structures. In this section, we extend our type algebra and type system in order to support mutable data structures like references, arrays, and dictionaries.

4.1 Opaque data types

We first focus on extending our type algebra. One possible way to do that is to add a built-in type constructor for each primitive mutable data structure, for instance references, for which [Castagna and Frisch 2005] propose a set-theoretic interpretation. However, their interpretation of references does not admit a set-theoretic model when recursion occurs inside a reference, which forces them to restrict the shape of valid types. While this is not an issue in their context where types are monomorphic (they do not feature type variables), this may be problematic in our setting as the tallying algorithm may build recursive types that do not meet these restrictions. Moreover, adding tailored type constructors for each primitive data structure requires, for each of them, to extend the type interpretation, the subtyping algorithm, and the tallying algorithm. Instead, we define in this section a general notion of opaque data types that allows us to represent new parametric data structures, disjoint from the other values of our language, and without worrying about their underlying structure while still accounting for the variance of their parameters.

It is also worth noting that our focus here is solely on the typing of opaque data types; we do not explore the semantic aspects relating to the conversion of a concrete data structure into an abstract one, which would require the boxing and unboxing of values at the interface level. Such an abstraction mechanism would allow for greater modularity and enable separate compilation, but this is not the focus of this paper.

4.1.1 Opaque data types and subtyping. We extend the type syntax with constructors representing opaque data types:

$$\textbf{Types} \quad t \quad ::= \quad \ldots \mid \#\mathtt{o}(t) \mid \#\mathtt{o}$$

where $\mathtt{o}$ ranges over a set of names.

The constructor $\#\mathtt{o}$ corresponds to the top element of the type lattice for the opaque data type $\mathtt{o}$, while $\#\mathtt{o}(t)$ constructs a specific instance. For each opaque data type $\mathtt{o}$, our subtyping and tallying algorithms [Castagna et al. 2015] are extended with the following case:

$$\bigwedge_{p\in P} \#\mathtt{o}(t_p) \leq \bigvee_{n\in N} \#\mathtt{o}(s_n) \quad \Leftrightarrow \quad \exists p \in P.\ \exists n \in N.\ t_p \simeq s_n$$

Roughly, a conjunction of instances of an opaque data type $\mathtt{o}$ is a subtype of a disjunction of instances of $\mathtt{o}$ if and only if both sides share an instance with the same parameter (modulo semantic equivalence).

This subtyping relation is rather weak: in particular, two instances $\#\mathtt{o}(t_1)$ and $\#\mathtt{o}(t_2)$ are comparable if and only if $t_1 \simeq t_2$. However, it is possible to refine this subtyping relation if we know that the concrete set-theoretic interpretation of $\#\mathtt{o}(.)$ satisfies some properties. We distinguish 7 cases and provide a subtyping relation for each:

General case: we know nothing about the set-theoretic interpretation of $\#\mathtt{o}(.)$. This case can be used to represent an opaque data type with an invariant parameter, and corresponds to the subtyping relation above.

Monotonic interpretation: when $t_1 \leq t_2 \implies \#\mathtt{o}(t_1) \leq \#\mathtt{o}(t_2)$. This case can be used to represent an opaque data type with a covariant parameter.

Monotonic interpretation and $\wedge$-preservation: when the interpretation of $\#\mathtt{o}(.)$ is monotonic and satisfies $\#\mathtt{o}(t_1 \wedge t_2) \simeq \#\mathtt{o}(t_1) \wedge \#\mathtt{o}(t_2)$.

Monotonic interpretation and $\wedge$-$\mathbb{0}$-preservation: when the interpretation of $\#\mathtt{o}(.)$ is monotonic and satisfies $\#\mathtt{o}(t_1 \wedge t_2) \simeq \#\mathtt{o}(t_1) \wedge \#\mathtt{o}(t_2)$ and $\#\mathtt{o}(\mathbb{0}) \simeq \mathbb{0}$.

Monotonic interpretation and $\vee$-preservation: when the interpretation of $\#\mathtt{o}(.)$ is monotonic and satisfies $\#\mathtt{o}(t_1 \vee t_2) \simeq \#\mathtt{o}(t_1) \vee \#\mathtt{o}(t_2)$.

Monotonic interpretation and $\vee$-$\mathbb{1}$-preservation: when the interpretation of $\#\texttt{o}(.)$ is monotonic and satisfies $\#\texttt{o}(t_1 \vee t_2) \simeq \#\texttt{o}(t_1) \vee \#\texttt{o}(t_2)$ and $\#\texttt{o}(\mathbb{1}) \simeq \#\texttt{o}$.

Monotonic interpretation and $\wedge$-$\vee$-preservation: when the interpretation of $\#\texttt{o}(.)$ is monotonic and preserves $\wedge$ and $\vee$. This case can be used to represent boxed values (i.e. when the interpretation of $\#\texttt{o}(t)$ just adds a tag to the values in the interpretation of t).

The subtyping relations associated with each case can be found in Appendix E. Note that, even though we do not provide a case for anti-monotonic interpretations, it can be handled with the monotonic case by negating the parameter. For instance, an instance of an opaque data type with a contravariant parameter α can be encoded as $\#\texttt{o}(\neg\alpha)$ where $\texttt{o}$ is a monotonic opaque data type.

4.1.2 Interpretation and soundness. Instead of having a fixed set-theoretic interpretation for opaque data types, we parametrize our interpretation of types by some arbitrary functions – one for each opaque data type $\texttt{o}$ – mapping the interpretation of a parameter t to the interpretation of the opaque data type $\#\texttt{o}(t)$.

Then, for each of the cases above, we ensure the associated subtyping relation is sound for every possible interpretation satisfying the properties. More precisely, for any such type interpretation $[\![.]\!]$, our subtyping relation must satisfy $\#\texttt{o}(t_1) \leq \#\texttt{o}(t_2) \implies [\![\#\texttt{o}(t_1)]\!] \subseteq [\![\#\texttt{o}(t_1)]\!]$. Note that we do not have the other direction in general: this is the price to pay for using a general notion of opaque data types instead of tailored types for each primitive data structure of our language. A formal statement of soundness and proofs for each case can be found in Appendix E.

4.2 Value restriction

Opaque data types allows us to define interfaces operating on mutable data structures, for instance:

```
abstract type ref('a) (* Define an opaque type w/ an invariant param. *)
val ref : 'a -> ref('a)          (* Constructor *)
val (<-) : ref('a) -> 'a -> ()   (* Setter *)
val (!) : ref('a) -> 'a          (* Getter *)
```

However, in order for these type signatures to be sound, our type system must implement the value restriction used in Hindley-Milner type systems with mutable references [Wright 1995], such as SML, OCaml and F#. Indeed, the type system defined in Section 3 with no further restriction can derive a type for the following expression, even though its reduction gets stuck:

$$\texttt{let}\, x = \texttt{ref 0}\, \texttt{in}\, (x \texttt{ <- false}, (!x) + 1)$$

Indeed, the definition `ref 0` can be typed $\texttt{ref}(0 \vee \alpha)$, which is then generalized into the type $\forall\alpha.\texttt{ref}(0 \vee \alpha)$. Then, the typing derivation of the left-hand side of the pair can choose to instantiate α with `false`, while the right-hand side can choose to instantiate α with $\mathbb{0}$.

The value restriction avoids this issue by preventing let-bindings to generalize arbitrary expressions: only values can be generalized. As values cannot be reduced any further, evaluating them cannot cause any side effects (such as creating or writing to a reference). Therefore, their type can be safely generalized. The value restriction can easily be implemented by amending the typing

rules for let-bindings in the declarative and algorithmic type systems:

$$[\textsc{Let}]\ \frac{\Sigma \vdash e_1 : s \qquad (\forall i \in I) \quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t}{\Sigma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \ : t}\ \begin{array}{l} s \leq \bigvee_{i\in I} s_i, \quad \neg\mathtt{value}(e_1) \Rightarrow \vec{\alpha} = \emptyset \\ \vec{\alpha}\#\mathrm{vars}(\Sigma), \quad \forall i \in I.\ \vec{\alpha}\#\mathrm{vars}(s_i) \end{array}$$

$$[\textsc{Let-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e_1 \mid a] : s \qquad \vec{\alpha} = \begin{cases} \mathrm{vars}(s) \setminus (\mathrm{vars}(\Sigma) \cup \bigcup_{i\in I} \mathrm{vars}(s_i)) & \text{if } \mathtt{value}(e_1) \\ \emptyset & \text{otherwise} \end{cases} \qquad (\forall i \in I) \quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash_{\mathcal{A}} [e_2 \mid a_i] : t_i}{\Sigma \vdash_{\mathcal{A}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \mathtt{let}\ (a, \{(s_i, a_i)\}_{i\in I})] : \bigvee_{i\in I} t_i}\ s \leq \bigvee_{i\in I} s_i$$

The additional side-conditions (in red) state that, if e_1 is not a value, then the set of generalized type variables $\vec{\alpha}$ must be empty.

5 MLsem: a prototype implementation

The type system, reconstruction algorithm and optimizations described in this paper have been implemented in a prototype, MLsem (*sem* stands for *semantic subtyping*), of about 5000 lines of OCaml (including parsing). The purpose of this prototype implementation is to demonstrate that our type system is general enough to type both functional and imperative features.

5.1 Program transformations and encodings

MLsem implements the type system of Section 3 as well as several extensions, in particular user type annotations, pattern matching, and recursive functions (which can be encoded using a fixpoint combinator). More importantly, MLsem extends our language with imperative features such as mutable variables, sequences, loops, and more advanced imperative control flow (in particular `break` and `return`). These imperative extensions are encoded in the functional core calculus of Section 2, as shown in Figure 4. The encodings and program transformations shown in this section are only used for typeability, they are not meant to preserve the semantics of the program.

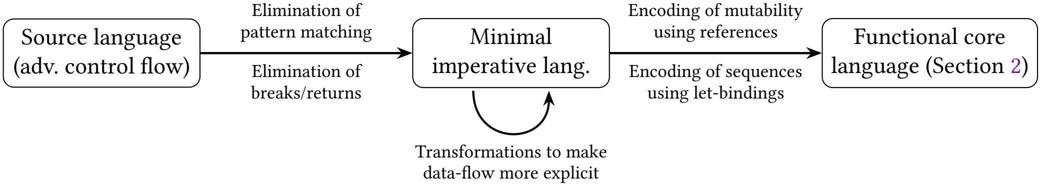

Fig. 4. Program transformations in MLsem

Source language. The source language has all the features described above: mutable variables, assignments, sequences, loops, *return* and *break* statements, and pattern matching. Pattern matching is eliminated through a program transformation as described in [Laurent 2024], by encoding it with type-cases and let-bindings. Another local program transformation eliminates advanced control flow instructions, in particular some *return* and *break* statements. Unhandled exceptions can be replaced with an expression of type $\mathbb{0}$, making subsequent code in the current block unreachable.

Minimal imperative language. This intermediate language still has mutable variables, assignments, sequences and loops, which allows running some local program transformations to make the data-flow more explicit. The objective is to minimize the use of mutable variables in order to enable the type system to perform type narrowing, which cannot occur on mutable variables as a reference

type cannot be decomposed in a useful way (`ref('a|'b)` cannot be decomposed into `ref('a)`, `ref('b)`).

Functional core language. This corresponds to the language defined in Section 2, with only immutable let-bindings, the usual constructors and destructors, and type-cases. Sequences are encoded with let-bindings, mutable variables are encoded as references, and the associated read/write operations are encoded as applications. At this point, loops can be replaced by simple conditionals (which are encoded as type-cases): since our typing of mutable variables is flow-insensitive (their type is invariant), typing the body of a loop once or multiple times is the same.

```
(* 1. Source language *)
let neg_and_pos x =
  let mut x = x in
  if x is Nil do return x end ;

  if x < 0 do x := -x end ;
  x := (-x,x) ;
  return x
```

```
(* 2. Minimal imperative language *)
let neg_and_pos x =
  let mut x = x in
  if x is Nil then x
  else begin
    if x < 0 do x := -x end ;
    x := (-x,x) ;
    x
  end
```

```
(* 3. Optimized imp. lang. *)
let neg_and_pos x =
  let mut x' = x in
  if x is Nil then x
  else begin
    if x < 0 do x' := -x end ;
    let x'' = (-x',x') in
    x''
  end
```

```
(* 4. Functional core language *)
let neg_and_pos x =
  let x' = ref x in
  if x is Nil then x
  else begin
    let _ = if x < 0 then x' <- -x else () in
    let x'' = (0 - !x', !x') in
    x''
  end
```

Fig. 5. Example of code transformation. Type inferred: `(int -> (int,int)) & (Nil -> Nil)`

Figure 5 shows an example of transformation from the source language to the functional core language of Section 2. The syntax used is the one of MLsem. The first transformation eliminates `return x` statements. While the final `return` statement can directly be removed as it is the last operation, the one in the `if` must be eliminated by moving the code after the `if` into a `else`. This makes explicit the fact that the code after the `if` is only executed when `x is Nil` is false.

The second transformation tries to make the data-flow more explicit by introducing shadowing in place of assignments when possible. Mutable variable declarations and assignments are preceded by a similar immutable variable declaration (for instance, `x := e ; ...` becomes `let x' = e in x := x' ; ...`) that stays valid as long as the mutable variable is not reassigned. Reads that happen in this area of validity will use the immutable version of the variable. In our example, this makes both the test `if x is Nil` and the reads of `x` in the `else` branch to use the same immutable variable, allowing the type system to perform type narrowing.

Finally, the third transformation encodes the remaining mutable operations into our core language: mutable cell creations become applications of `ref`, reads become applications of `!`, and writes become applications of `<-` (cf. Section 4.2).

Why not monadic encodings? A common way to turn a program with effects into a pure functional core is by using monadic transformations, such as the ones defined by [Ullrich and de Moura 2022] to

eliminate local imperativity. In our case, however, we decided to encode mutability using references, as in the OCaml language. We think this approach is more flexible and compositional for two reasons. First, while local imperativity can easily be eliminated using monads, doing so for global mutable variables would require rewriting the whole program in a monadic style. Second, the opaque data types we define to type references can also be used to encode other mutable data structures and operations, such as arrays and dictionaries. However, encoding mutable variables using references forces them to be typed invariantly throughout their entire scope of definition, making it impossible to perform type narrowing on them: this is why we perform additional program transformations to restore some of the control flow. Alternatively, it would be possible to use monadic encodings to eliminate local imperativity, but still use references to encode global mutability. We do not know what would be the impact on performance, and to which extent our type system would be able to perform type narrowing on the state monad.

5.2 Evaluation

```
let loop_invalid x =
  let mut x = x in while true do
    x := x + 1 ; x := false
  end ; x
(* Untypeable (+ only defined on int) *)
let loop_valid x =
  let mut x = x in while true do
    if x is ~int do return x end ;
    x := x + 1 ; x := false
  end ; x
(* ('a\int -> 'a\int) & ('a -> false|'a) *)
```

```
val rand_any : () -> any

let loop_type_narrowing y =
  let mut x in
  let mut y = y in
  while is_int
    (x := rand_any () ; x) do
    y := y + x
  end ;
  return (x,y)
(* int -> (any, int) *)
```

```
abstract type dict('k, 'v)
val dict : () -> dict('a, 'b)

abstract type array('a)
val array : () -> array('a)

val ([]<-) : ((dict('a, 'b), 'a) -> 'b -> ())
           & ((array('b), int) -> 'b -> ())

val ([]) : ((dict('a, 'b), 'a) -> 'b)
         & ((array('b), int) -> 'b)
```

```
let nested x y =
  let d = dict () in
  d[x]<- (array ()) ;
  (d[x])[0]<- y ; (d[x])[0]
(* any -> 'a -> 'a *)
let swap i j x =
    let tmp = x[i] in
    x[i]<- x[j] ; x[j]<- tmp
(* ('a -> 'a -> dict('a,'b))
& (int -> 'a&int -> array('b))
-> () *)
```

Fig. 6. Imperative and overloaded code examples (types inferred are written as comments)

We tested MLsem on a corpus of 150 small functions (amounting to about 550 lines of code), focusing on the evaluation of type narrowing, on the typing and application of overloaded functions, on operations involving opaque data types, and on type inference. Some examples from this corpus can be found in Figure 6 and are discussed below. These examples can be tested on our prototypeavailable online , as well as many others (with recursive functions, pattern matching, etc.).

Type narrowing. Our test corpus contains many functions involving type narrowing, and in particular the 14 examples from [Tobin-Hochstadt and Felleisen 2010] (available in Appendix F). For each, two variants have been tested, one where the function parameters are explicitly annotated

with their type, and one where they are not. Both variants successfully type-check, and for the unannotated ones, our prototype infers some intersections of arrows that capture the overloaded behaviors of the functions.

Type narrowing also plays a role in the typing of `loop_valid` and `loop_type_narrowing` (Figure 6). In both cases, it is used to deduce that `x` is an integer in the body of the loop, which is required to type the application of + (of type `int -> int -> int`).

Likewise, type narrowing is used to type the example `filter_imp` from the introduction: when typing the body of the conditional `if f e do push res e end`, our type system is able to deduce that `e` has type `'a \'b`, allowing to derive for `filter_imp` the following type:

```
('a -> bool)&('b -> false) -> array('a|'b) -> array('a\'b|'c)
```

Type inference and overloaded behaviors. When the type of a function parameter is not explicitly annotated, the type reconstruction algorithm infers it. For instance, the function `nested` (Figure 6) – which stores its second argument in an array, itself stored in a dictionary, and then retrieves and returns this argument – gets the type `any -> 'a -> 'a`. Note the use of the overloaded operators `([])` and `([]<-)` that can apply on both arrays and dictionaries. The function `swap` gets the following type, capturing both cases where `x` is a dictionary and an array:

```
('a -> 'a -> dict('a,'b) -> ()) & (int -> 'a&int -> array('b) -> ())
```

Having type inference is especially useful when dealing with anonymous functions, as their signatures are generally not provided. Moreover, anonymous functions are usually small and are only applied locally (meaning that their type will not be visible to the programmer), thus inferring complex overloaded types for them is less of an issue. Type inference can also be used to help the programmer annotate top-level functions by providing type suggestions in an interactive way, even though it may sometimes fail to find a suitable type. Though our reconstruction algorithm explores all possible behaviors, sometimes resulting in a combinatorial explosion for large overloaded functions, type system implementations are free to rely on heuristics to constrain the search space: this paper provides a general and flexible approach, but typing dynamic languages remains a difficult problem that may require compromises.

Time performance. One of the difficulties when dealing with set-theoretic types is to maintain good and homogeneous performance. MLsem has made significant progress in this matter. Type-checking the annotated version of the 14 examples from [Tobin-Hochstadt and Felleisen 2010] (cf. Appendix F) takes 30ms[2]. Without annotations, the type of these functions is reconstructed in 75ms. Each small annotated function from our corpus (1-10 lines each) type-checks in 1-25ms, including recursive functions. Unannotated (or partially annotated) functions can be slower, in particular recursive functions: for instance, the type of a unannotated `map` function (which is recursive and takes two parameters, one of which is a function) is reconstructed in 28ms. Some more complex unannotated recursive functions may not finish type-checking even after a minute.

In addition to our corpus of small functions, we evaluated MLsem on a bigger function, adapted from the `bal`(ance) function used in the module `Map` of the OCaml standard library [OCaml 2023]. The code for this function is available in Appendix F. This function has 6 different pattern-matching constructs and 5 type-cases, making it a good candidate for evaluating the performance of our approach to type narrowing. Our prototype can type-check it in 220ms. For comparison, using the prototype of [Castagna et al. 2024b], this function type-checks in about 2200ms. More generally, across all examples from the [Castagna et al. 2024b] test corpus, our approach performs similarly or better. These performance gains stem mainly from (i) the reconstruction algorithm, which avoids

[2]These tests have been performed on a laptop with an Intel i5-12500H CPU.

re-typing definitions during backtracking, and (*ii*) implementation differences: [Castagna et al. 2024b] rely on the CDuce library for type operations, while we use a custom library.

We also evaluated our approach on the `filtermap` function presented in [Schimpf et al. 2023] (cf. Appendix F). This recursive function is type-checked against an intersection of arrows, capturing its overloaded behavior, and requires type narrowing to be type-checked. [Schimpf et al. 2023] state that their type system is able to type-check this function, but that it takes over one minute. With our approach, we are able to type-check it in 15ms with a similar hardware. While the performance improvements observed through these examples are encouraging, our implementation is still in the prototype stage and has yet to be tested on a large codebase.

6 Related Work

Comparison with approaches using set-theoretic types. This paper reuse several ideas from [Castagna et al. 2024b], in particular the use of a union elimination rule to implement type narrowing, and the definition of an algorithmic type system that relies on an annotation tree. Though, they are several major differences between the two approaches. First, their type system only generalizes type variables at top-level, hence the need to define a notion of programs in addition to the notion of expressions. Programs have generalizing let-bindings, and expressions do not have let-bindings (though they formalize an extension adding non-generalizing let-bindings to expressions). Their formalism also does not rely on type schemes, but on two disjoint set of type variables: one for *polymorphic* type variables, and one for *monomorphic* type variables. Also, for any expression their type system can derive types containing polymorphic type variables (which is incompatible with the implementation of a value restriction), whereas in our system type schemes from the environment are immediately instantiated by the [Var] rule.

Another difference is the presence of a general union elimination rule, which can be used on any expression (not only on let-bindings). A consequence is that their type system is able to narrow the type of any sub-expression and not only variables: for instance, their type system is able to type the expression $(f\ 42 \in \texttt{int})\ ?\ (f\ 42) + 1 : 0$ with $f : \mathbb{1} \to \mathbb{1}$. However, this is only sound on a pure λ-calculus. This more general union elimination rule also makes the implementation of the type system more complex. Indeed, their algorithmic type system works on expressions of a particular form that they call *maximal sharing canonical* (MSC) form, and that factorizes and isolates every distinct sub-expression in a separate definition (a *binding*). This also adds bureaucracy to the reconstruction algorithm, as these *bindings* must be typed only if they are actually used later in the current branch (thus they are different from our let-bindings that have a strict semantics). Our algorithm for reconstructing annotation trees is simpler and generates fewer constraints: while the annotation reconstruction algorithm of [Castagna et al. 2024b] invalidates and recomputes type substitutions of each definition when the algorithm backtracks, we don't have to do so in our case.

Another approach to type inference is used in [Petrucciani 2019; Schimpf et al. 2023]. Instead of a backtracking algorithm that solves local constraints as it explores the expression, they generate for the whole expression a large set of constraints that is then normalized, simplified, and solved using tallying. This approach has been shown to be very efficient for Hindley-Milner systems, where constraints are solved using unification and have at most one principal solution. However, solving large semantic subtyping constraints can be expensive, and the number of solutions can grow exponentially with the number of type variables. This may explain why [Schimpf et al. 2023] sometimes exhibit long type-checking times even for annotated functions (for instance the `filtermap` example mentioned in Section 5.2). Instead, our approach only requires solving small local constraints, and solutions can be immediately simplified to eliminate redundant substitutions and type variables.

Lastly, our approach to type narrowing differs from [Castagna et al. 2024a; Schimpf et al. 2023]. While our type system works on a language featuring type-cases that can test arbitrary expressions, and presents a general solution for type narrowing that relies on (a restricted version of) the union elimination rule, the approaches mentioned above implement type narrowing in a more specific context: a language with pattern-matching and where patterns can be guarded by expressions of a specific form. They use specific deduction rules to strengthen the type of the variables occurring in a pattern just before typing the associated branch. Although less general, their approach may be more efficient in their setting as it does not require typing the body of let-bindings multiple times.

Comparison with Hindley-Milner and its extensions. Our type discipline combines several ideas from the Hindley-Milner family of type systems. Hindley-Milner type systems have first-order prenex polymorphism, principal types, and type inference is decidable: [Damas and Milner 1982] describes a simple method, referred as algorithm $\mathcal{W}$. Our approach for type inference takes inspiration from algorithm $\mathcal{W}$, but instead of using unification to solve syntactic constraints, we use tallying to solve semantic subtyping constraints. This results in a branching type inference algorithm, because unlike unification, tallying may yield multiple solutions (which is expected, as one of our purpose is to type overloaded functions). Another key idea we reuse from Hindley-Milner systems is the value restriction [Wright 1995], necessary to preserve soundness in the presence of parametric mutable data structures.

Inference for ML systems with subtyping, unions, and intersections has been studied in MLsub [Dolan and Mycroft 2017] and extended with richer types and a limited form of negation in MLstruct [Parreaux and Chau 2022]. They define a lattice of types and an algebraic subtyping relation with some restrictions that ensure principality, but forbid (or approximate) intersections of arrow types, making it impossible to use intersections to achieve ad-hoc polymorphism. We justify our choice of set-theoretic types, with no principality and a complex inference, by our aim to type dynamic languages, such as JavaScript or Python, where overloading plays an important role. We favor the expressivity necessary to type many idioms of these languages, and rely on user-defined annotations when necessary to compensate for the incompleteness of the type inference.

In a more recent work, [Parreaux et al. 2024] define $F_{\{\leq\}}$, a λ-calculus with first-class polymorphism. It features intersections and unions, but the use of those is restricted: intersections can only appear in negative positions, and unions can only appear in positive positions. They define a type inference for this system, SuperF, inspired by polar type systems such as [Jim 2000]. Their type system is more expressive than MLsub and MLstruct, but their type inference cannot infer principal types in general. This approach takes a different direction from ours: while we choose to restrict polymorphism to be prenex, they use unrestricted first-class polymorphism but restrict the use of intersections and unions in negative and positive positions respectively. In our context of typing dynamic languages, intersections in positive positions are used to capture the behavior of overloaded functions (ad-hoc polymorphism), and unions in negative positions can be used to represent a structural form of algebraic data types.

Comparison with type-checkers for dynamic languages. Languages such as JavaScript or Python already have extensions and/or tools providing static type-checking. For instance, TypeScript is a statically-typed extension of JavaScript (adding syntax for types and providing a type-checker), and Mypy is a tool for type-checking Python (reusing the type syntax already present in the specification of Python). However, these approaches lack a formal foundation, and usually prioritize flexibility for the programmer over type safety. Still, they can detect many type inconsistencies (though not all of them), and provide type information that is useful for the documentation and the toolchain, which explains their success and the increasing interest programmers have for static type-checking of dynamic languages.

In terms of features, both approaches support parametric polymorphism (sometimes called *generics*). Python lacks intersection types, both for functional types and objects. For objects, it can be partly mitigated using *protocols* (interfaces, but typed structurally instead of nominally), though it may cost verbosity (one may have to define both an abstract class and the corresponding protocol). TypeScript does support intersection types, but not for function types. Overloaded functions at top-level can be given several type signatures, but these overloaded signatures are not first-class types, and thus it cannot be used to type, for example, our `filter` function from the introduction. Neither Python nor TypeScript have negation types, nor type narrowing (except for tests directly on variables such as `if (typeof x === "number")`).

When not annotated, Mypy tries to infer the type of the parameters of functions. However, their type inference is not based on a well-etablished constraint solving algorithm such as tallying, and may be unpredictable due to the use of heuristics. TypeScript does not provide type inference for the parameter of functions, but performs contextual typing: for instance, when writing `window.onmousedown = function (ev) { ... }`, it can deduce the type of `ev` because it knows what `window.onmousedown` expects.

Another project that adds static typing to an initially untyped dynamic language is Luau. Luau uses a type system featuring semantic subtyping to statically type Lua code. In addition to the safety guarantees it brings, the static type information is used by their interpreter to perform optimizations. Their implementation of semantic subtyping, which they call *pragmatic semantic subtyping*, is inspired by set-theoretic types, though it differs from the set-theoretic interpretation on some aspects, notably for functions [Jeffrey 2022]. As with set-theoretic types, they support singleton types, unions, and intersections. However, negation is only supported on test types (those that can appear in type-cases), and not on structural types (like arrow types). Overall, they have opted for a pragmatic approach that restricts some features of set-theoretic types for the sake of simplicity and performance. However, the absence of arbitrary negation types is a limitation for implementing our general approach for type narrowing and type inference using tallying.

7 Conclusion

This work presents a type system that aims at typing dynamic languages mixing both functional and imperative features (e.g. Python, R, or JavaScript). It reuses ideas from previous work, in particular the use of union-elimination to implement type narrowing [Castagna et al. 2024b], and the use of *tallying* to solve subtyping constraints for type inference [Castagna et al. 2015]. Unlike [Castagna et al. 2024b], our approach is compatible with non-determinism and mutability, and does not require turning expressions in an special MSC-form. We also extended set-theoretic types with a notion opaque data types that may be used to represent various data structures such as *references*, *arrays* or *dictionaries*. The type system and reconstruction algorithm presented in this paper are general and are not meant to be used as is to type a real-world language, but rather to serve as a basis that can be adapted to a more specific setting and extended with language-specific features. To demonstrate the feasibility, we implemented a functional and imperative language with type-cases, MLsem, and showed how it can be typed by our system using simple encodings and program transformations. We hope that our work could form the basis of new static type systems for dynamic languages, providing more expressive types and typing rules that are both safer and more predictable.

7.1 Future work

Gradual typing. Though our type discipline makes it possible to type functions manipulating heterogeneous data with precision, it cannot type all the features dynamic languages usually have. For instance, functions involving reflective operations like `eval` cannot be typed. Type-checkers for dynamic languages usually mitigate this issue through the use of gradual typing, allowing some

functions to remain untyped while still being able to type other parts of the codebase. Gradual typing offers interoperability with untyped codebases: for instance, TypeScript programs can freely use untyped JavaScript libraries. Gradual typing comes in many flavors: some implementations may prioritize type precision, others may prioritize the preservation of the semantics (no added cast at runtime), or the compatibility with unmigrated code. In the context of set-theoretic types, gradual typing has first been studied in [Castagna et al. 2019]. They propose a sound approach to gradual typing where casts are added to the initial program. In their efforts to type Elixir using set-theoretic types, [Castagna et al. 2024a] present a type system with some tailored sound gradual typing. They do not add new casts to the program, but manage to mitigate the propagation of gradual types by reusing casts that are already present. They do that by distinguishing two kinds of functions: those that check the type of their arguments at runtime (*strong arrows*) and those that do not (*weak arrows*). While we have no doubt about the feasibility of adding gradual typing to our framework, the way to do it may depend on the needs of the target language.

Typing of objects and interfaces. Many dynamic languages are object-oriented (e.g. Python and JavaScript). Usually, the semantics of objects in dynamic languages follow the *duck-typing* rule: if an object implements all the features of a duck, then it can be considered as a duck, regardless of how it has been created. In terms of static typing, it naturally translates to *structural subtyping*, where an object o_1 is considered to be a subtype of an object o_2 if and only if all the attributes and methods o_2 exposes are also in o_1 with similar or smaller types signatures. This is opposed to *nominal subtyping*, where an object gets its type from the constructor used to create it, and where subtyping may capture the *inheritance* relations between the object definitions. Structural and nominal subtyping each have advantages, and some languages feature both (e.g., *ABC* and *Protocols* in Python). Nominal subtyping keeps types simple and gives more control to the programmer about the subtyping relation. Structural subtyping is more flexible, but types (and subtyping) can quickly become more complex as objects get composed together. Finding an encoding for objects and interfaces that remains simple while capturing the idioms of dynamic languages is a challenge that will need to be addressed in future work.

Type inference. This paper formalizes a reconstruction algorithm that is able to infer the domain of λ-abstractions. In the case of functions with an overloaded behavior, the type inferred may be composed of several intersections of arrows that capture the different behaviors. However, as seen in Section 5.2, the types inferred may sometimes be overly complicated. Some research about the simplification of polymorphic types could be beneficial. Moreover, as overloaded functions are composed together, they may exhibit more and more overloaded behaviors. We do not want types to capture all these behaviors, as it would result in increasingly complex types. Instead, intersection of arrows should be used with parsimony. To achieve such a compromise between precision and simplicity, one may rely on the use of LLMs as an oracle to simplify types or to suggest, for a given function, which domains are worth exploring separately and which ones can be assimilated. In addition, type inference should be thought as an interactive tool integrated into the IDE.

Language-specific features and optimizations. This paper presents a general framework, in which type narrowing is implemented through union elimination, and mutable data structures are modeled with parametrized opaque data types. Though, specific aspects and features of the target language we want to type should be taken into account in order to tailor a more efficient and optimized type system. This paper defines a minimal language that supports basic functional and imperative features, but the encoding of a real programming language with more advanced features (such as a built-in mechanism for dynamic dispatch, or R-like laziness) into this minimal language may not be trivial.

Data availability statement

All the auxiliary definitions and proofs that we omitted from the main text are available in the appendices of the extended version. An online version of our prototype MLsem is available at https://e-sh4rk.github.io/MLsem/.

References

Giuseppe Castagna. 2024. *Programming with Union, Intersection, and Negation Types.* Springer International Publishing, Cham, 309–378. doi:10.1007/978-3-031-34518-0_12

Giuseppe Castagna, Guillaume Duboc, and José Valim. 2024a. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (2024).

Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. doi:10.1145/1069774.1069793

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: a New Perspective. *Proc. ACM Program. Lang.* 3, Article 16, POPL '19: 46th ACM Symposium on Principles of Programming Languages (jan 2019).

Giuseppe Castagna, Mickël Laurent, and Kim Nguyen. 2024b. Polymorphic Type Inference for Dynamic Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 40 (Jan. 2024), 32 pages. doi:10.1145/3632882

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15)*. 289–302. doi:10.1145/2676726.2676991

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '14)*. 5–17. doi:10.1145/2676726.2676991

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. 2016. Set-Theoretic Types for Polymorphic Variants. *SIGPLAN Not.* 51, 9 (sep 2016), 378–391. doi:10.1145/3022670.2951928

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic foundation of parametric polymorphism and subtyping. *SIGPLAN Not.* 46, 9 (Sept. 2011), 94–106. doi:10.1145/2034574.2034788

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Albuquerque, New Mexico). Association for Computing Machinery, New York, NY, USA, 207–212. doi:10.1145/582153.582176

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). Association for Computing Machinery, New York, NY, USA, 60–72. doi:10.1145/3009837.3009882

Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML.* Ph. D. Dissertation. Université Paris Diderot.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM* 55, 4 (Sept. 2008), 19:1–19:64. http://doi.acm.org/10.1145/1391289.1391293

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2015. A logical approach to deciding semantic subtyping. *ACM Transactions on Programming Languages and Systems* 38, 1 (2015), 3. doi:10.1145/2812805

Alan Jeffrey. 2022. Semantic Subtyping in Luau. Blog post. https://blog.roblox.com/2022/11/semantic-subtyping-luau Accessed on May 6th 2023.

Trevor Jim. 2000. A Polar Type System. In *ICALP Workshops 2000, Proceedings of the Satelite Workshops of the 27th International Colloquium on Automata, Languages and Programming, Geneva, Switzerland, July 9-15, 2000*, José D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells (Eds.). Carleton Scientific, Waterloo, Ontario, Canada, 323–338.

Mickaël Laurent. 2024. *Polymorphic type inference for dynamic languages : reconstructing types for systems combining parametric, ad-hoc, and subtyping polymorphism.* Theses. Université Paris Cité. https://theses.hal.science/tel-04972652

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

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

OCaml. 2023. Standard Library: Map module. Github repository. https://github.com/ocaml/ocaml/blob/trunk/stdlib/map.ml

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

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

Tommaso Petrucciani. 2019. *Polymorphic Set-Theoretic Types for Functional Languages.* Ph. D. Dissertation. Joint Ph.D. Thesis, Università di Genova and Université Paris Diderot. https://tel.archives-ouvertes.fr/tel-02119930 Available at https://tel.archives-ouvertes.fr/tel-02119930.

Albert Schimpf, Stefan Wehr, and Annette Bieniusa. 2023. Set-theoretic Types for Erlang. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. doi:10.1145/3587216.3587220

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 117–128. doi:10.1145/1863543.1863561

Sebastian Ullrich and Leonardo de Moura. 2022. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.* 6, ICFP, Article 109 (Aug. 2022), 28 pages. doi:10.1145/3547640

Andrew K. Wright. 1995. Simple imperative polymorphism. *Lisp Symb. Comput.* 8, 4 (Dec. 1995), 343–355. doi:10.1007/BF01018828

A Semantic Subtyping

This appendix summarizes the interpretation of set-theoretic types formalized in [Gesbert et al. 2015] and how it can be used to define semantic subtyping.

A.1 Type interpretation

In order to define subtyping over these types, the idea is to interpret each ground type (i.e., a type that does not contain type variables) as a set of values of our language. Then, subtyping can be defined as set containment over the interpretation of types. Intuitively, each ground type is associated to the set of values having this type: for instance, the base type `true` is interpreted as the singleton containing the constant `true`, while the type `bool` = `true` ∨ `false` is interpreted as the set {`true`, `false`}.

However, this idea becomes subtler when dealing with arrow types. Although an arrow type intuitively corresponds to a function (i.e., a λ-abstraction), interpreting an arrow type as a set of λ-abstractions is problematic as it yields a circular reasoning: determining if a λ-abstraction is in the interpretation of a type requires to define a type system, which in turns needs the subtyping relation that we are trying to build. In order to break this circularity, the interpretation of types is not defined over values of our language but over a domain $\mathcal{D}$ defined below. Note that this does not necessarily invalidate the “types as set of values” intuition, as it is discussed in Castagna and Frisch [2005, Section 2.7].

In addition, we need to define an interpretation for all types, and not only ground ones (the interpretation domain $\mathcal{D}$ should account for type variables). A simple model was proposed by [Gesbert et al. 2015]. We succinctly present it in this section. The reader may refer to [Castagna 2024, Section 3.3] for more details.

Definition A.1 (Interpretation domain [Gesbert et al. 2015]). The *interpretation domain* $\mathcal{D}$ is the set of finite terms d produced inductively by the following grammar

$$d ::= c^L \mid (d, d)^L \mid \{(d, \partial), \dots, (d, \partial)\}^L$$
$$\partial ::= d \mid \Omega$$

where c ranges over the set C of constants, L ranges over finite sets of type variables, and where Ω is such that $\Omega \notin \mathcal{D}$.

The elements of $\mathcal{D}$ correspond, intuitively, to (denotations of) the results of the evaluation of expressions, labeled by finite sets of type variables. In particular, in a higher-order language, the results of computations can be functions which, in this model, are represented by sets of finite relations of the form $\{(d_1, \partial_1), \dots, (d_n, \partial_n)\}^L$, where Ω (which is not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This is implemented by using in the second projection the meta-variable ∂ which ranges over $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ (we reserve d to range over $\mathcal{D}$, thus excluding Ω). This constant Ω is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used d instead of ∂, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently of its argument as long as this argument is typeable (see Section 4.2 of [Frisch et al. 2008] for details). The restriction to *finite* relations corresponds to the intuition that the denotational semantics of a function is given by the set of its finite approximations, where finiteness is a restriction necessary (for cardinality reasons) to give the semantics to higher-order functions. Finally, the sets of type variables that label the elements of the domain are used to interpret type variables: we interpret a type variable α by the set of all elements that are labeled by α, that is the set $\{d \mid \alpha \in \text{tags}(d)\}$ (where we define $\text{tags}(c^L) = \text{tags}((d, d')^L) = \text{tags}(\{(d_1, \partial_1), \dots, (d_n, \partial_n)\}^L) = L$).

We define the interpretation $[\![t]\!]$ of a type t so that it satisfies the following equalities, where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets and $\mathbb{B}$ denotes the function that assigns to each base type the set of constants of that type, so that for every constant c we have $c \in \mathbb{B}(\boldsymbol{b}_c)$ (we use $\boldsymbol{b}_c$ to denote the base type of the constant c):

$$[\![\mathbb{0}]\!] = \emptyset \qquad [\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\} \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![b]\!] = \mathbb{B}(b) \qquad [\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \{R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, \partial) \in R.\, d \in [\![t_1]\!] \implies \partial \in [\![t_2]\!]\}$$

Note that, even though we included $\mathbb{1}$ and the intersection $\wedge$ in the syntax of our types (Definition 2.1), those two can be defined from the other constructors: $\mathbb{1} = \neg\mathbb{0}$ and $t_1 \wedge t_2 = \neg(\neg t_1 \vee \neg t_2)$ (De Morgan's law). It is easy to see that, with these definitions, we have $[\![\mathbb{1}]\!] = \mathcal{D}$ and $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$. Thus, it is not necessary to define an interpretation for them.

We cannot take the equations above directly as an inductive definition of $[\![\cdot]\!]$ because types are not defined inductively but coinductively. Notice however that the contractivity condition of Definition 2.1 ensures that the binary relation $\rhd \subseteq \mathcal{T} \times \mathcal{T}$ defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian. This gives an induction principle[3] on $\mathcal{T}$ that we use combined with structural induction on $\mathcal{D}$ to give the following definition, which validates the equalities above.

Definition A.2 (Set-theoretic interpretation of types). We define a binary predicate $(d : t)$ ("the element d belongs to the type t"), where $d \in \mathcal{D}$ and $t \in \mathcal{T}$, by induction on the pair (d, t) ordered lexicographically. The predicate is defined as follows:

$$\begin{aligned}
(c : b) &= c \in \mathbb{B}(b) \\
(d : \alpha) &= \alpha \in \mathsf{tags}(d) \\
((d_1, d_2) : t_1 \times t_2) &= (d_1 : t_1) \text{ and } (d_2 : t_2) \\
(\{(d_1, \partial_1), ..., (d_n, \partial_n)\} : t_1 \to t_2) &= \forall i \in [1..n].\ \text{if } (d_i : t_1) \text{ then } (\partial_i : t_2) \\
(d : t_1 \vee t_2) &= (d : t_1) \text{ or } (d : t_2) \\
(d : \neg t) &= \text{not } (d : t) \\
(\partial : t) &= \text{false} \qquad \text{otherwise}
\end{aligned}$$

We define the *set-theoretic interpretation* $[\![.]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ as $[\![t]\!] = \{d \in \mathcal{D} \mid (d : t)\}$.

A.2 Semantic subtyping

Now that we have a set-theoretic interpretation of types, we can define the subtyping preorder and its associated equivalence relation as follows.

Definition A.3 (Subtyping relation). We define the *subtyping* relation $\leq$ and the *subtyping equivalence* relation $\simeq$ as $t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ and $t_1 \simeq t_2 \overset{\text{def}}{\iff} (t_1 \leq t_2) \text{ and } (t_2 \leq t_1)$.

This subtyping relation is decidable and is sometimes referred to as *semantic subtyping*, as it is not defined on the syntax of the type but on its interpretation.

[3] In a nutshell, we can do proofs and give definitions by induction on the structure of unions and negations—and, thus, intersections—but arrows, products, and base types are the base cases for the induction.

With this set-theoretic definition of subtyping, usual properties of sets are inherited by subtyping, for instance:

$$t_1 \vee t_2 \simeq t_2 \vee t_1 \qquad t_1 \wedge t_2 \simeq t_2 \wedge t_1 \qquad \text{(commutativity)}$$
$$t \vee t \simeq t \qquad t \wedge t \simeq t \qquad \text{(idempotence)}$$
$$\neg(\neg t) \simeq t \qquad \text{(double complement)}$$
$$t \vee (s_1 \wedge s_2) \simeq (t \vee s_1) \wedge (t \vee s_2) \qquad t \wedge (s_1 \vee s_2) \simeq (t \wedge s_1) \vee (t \wedge s_2) \qquad \text{(distributivity)}$$

For any two type substitutions ϕ_1 and ϕ_2, we write $\phi_1 \simeq \phi_2$ the pointwise subtyping equivalence of ϕ_1 and ϕ_2. An important property of the interpretation above is that subtyping is preserved by type substitutions:

$$\forall t_1, t_2, \phi.\ t_1 \leq t_2 \Rightarrow t_1\phi \leq t_2\phi$$

However, a naive definition of $\text{vars}(t)$ is not preserved by subtyping equivalence: for instance, we have $\mathbb{1} \simeq \alpha \vee \neg\alpha$, while a purely syntactic definition of $\text{vars}(t)$ would yield $\text{vars}(\mathbb{1}) = \emptyset$ and $\text{vars}(\alpha \vee \neg\alpha) = \{\alpha\}$. In order to avoid this, we define $\text{vars}(t)$ as being the set of *meaningful type variables* in t. This notion has been introduced by Castagna et al. [2016], where it was noted as $\texttt{mvar}(t)$, and is defined below.

Definition A.4 (Type variables). The set of type variables of a type t, noted $\text{vars}(t)$, is the following set of type variables:

$$\text{vars}(t) \stackrel{\text{def}}{=} \{\alpha \in \mathcal{V} \mid t\{\alpha \rightsquigarrow \mathbb{0}\} \not\simeq t\}$$

With this definition, the set of variables of a type is preserved by subtyping equivalence: $\forall t_1, t_2.\ t_1 \simeq t_2 \Rightarrow \text{vars}(t_1) = \text{vars}(t_2)$.

B Proof of Type Safety

$$[\textsc{Var}]\ \frac{\Sigma(x) = \forall\vec{\alpha}.t}{\Sigma \vdash x : t\phi}\ \mathrm{dom}(\phi) \subseteq \vec{\alpha} \qquad [\textsc{Const}]\ \frac{}{\Sigma \vdash c : \boldsymbol{b_c}} \qquad [\textsc{Choose}]\ \frac{\Sigma \vdash e_1 : t \qquad \Sigma \vdash e_2 : t}{\Sigma \vdash e_1 \oplus e_2 : t}$$

$$[\textsc{Let}]\ \frac{\Sigma \vdash e_1 : s \qquad (\forall i \in I) \quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t}{\Sigma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2\ : t}\ \begin{array}{l} s \leq \bigvee_{i \in I} s_i \\ \vec{\alpha}\#\mathrm{vars}(\Sigma),\quad \forall i \in I.\ \vec{\alpha}\#\mathrm{vars}(s_i) \end{array}$$

$$[\rightarrow\mathrm{I}]\ \frac{\Sigma, x : s \vdash e : t}{\Sigma \vdash \lambda x.e : s \rightarrow t} \qquad [\rightarrow\mathrm{E}]\ \frac{\Sigma \vdash e_1 : t_1 \rightarrow t_2 \qquad \Sigma \vdash e_2 : t_1}{\Sigma \vdash e_1 e_2 : t_2}$$

$$[\times\mathrm{I}]\ \frac{\Sigma \vdash e_1 : t_1 \qquad \Sigma \vdash e_2 : t_2}{\Sigma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\mathrm{E}_1]\ \frac{\Sigma \vdash e : t_1 \times t_2}{\Sigma \vdash \pi_1 e : t_1} \qquad [\times\mathrm{E}_2]\ \frac{\Sigma \vdash e : t_1 \times t_2}{\Sigma \vdash \pi_2 e : t_2}$$

$$[\in]\ \frac{\Sigma \vdash e : s \qquad s \wedge \tau \not\simeq \mathbb{0} \Rightarrow \Sigma \vdash e_1 : t \qquad s \setminus \tau \not\simeq \mathbb{0} \Rightarrow \Sigma \vdash e_2 : t}{\Sigma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t}$$

$$[\wedge]\ \frac{\Sigma \vdash e : t_1 \qquad \Sigma \vdash e : t_2}{\Sigma \vdash e : t_1 \wedge t_2} \qquad [\leq]\ \frac{\Sigma \vdash e : t}{\Sigma \vdash e : t'}\ t \leq t'$$

Fig. 7. Declarative type system

In this section, we prove the following type safety theorem (Theorem B.19).

Theorem (Type safety). *For every expression e and test type τ, if $\varnothing \vdash e : \tau$, then for any e' such that $e \rightsquigarrow^* e'$, we have either $e' \rightsquigarrow^* v$ for some $v \in \tau$ or $e' \rightsquigarrow^\infty$.*

B.1 Overview

Type safety is usually decomposed into two lemmas: type preservation (i.e., typeability is preserved by reduction) and progress (i.e., any well-typed expression is either a value or can be reduced). Though, the usual type preservation does not hold for our type system. Consider for instance the following expression:

$$\mathtt{let}\, y = \lambda x.x\, \mathtt{in}\, (y\ 42, y)$$

By performing a specific type decomposition, the type $t = 42 \times ((42 \rightarrow 42) \wedge \neg(42 \rightarrow 43))$ can be derived for this expression:

$$[\textsc{Let}]\ \frac{\dfrac{A}{\varnothing \vdash \lambda x.x : 42 \rightarrow 42} \quad \dfrac{B}{y : (42 \rightarrow 42) \wedge (42 \rightarrow 43) \vdash (y\ 42, y) : t} \quad \dfrac{C}{y : (42 \rightarrow 42) \setminus (42 \rightarrow 43) \vdash (y\ 42, y) : t}}{\varnothing \vdash \mathtt{let}\, y = \lambda x.x\, \mathtt{in}\, (y\ 42, y)\ : t}$$

The subderivations A and C are straightforward, and the subderivation B is as follows (note that $(42 \rightarrow 42) \wedge (42 \rightarrow 43) \simeq 42 \rightarrow \mathbb{0}$, and that for any type s, $\mathbb{0} \times s \simeq \mathbb{0}$):

$$[\leq]\ \dfrac{[\times\mathrm{I}]\ \dfrac{[\rightarrow\mathrm{E}]\ \dfrac{[\textsc{Var}]\ \dfrac{}{y : 42 \rightarrow \mathbb{0} \vdash y : 42 \rightarrow \mathbb{0}} \quad [\textsc{Const}]\ \dfrac{}{y : 42 \rightarrow \mathbb{0} \vdash 42 : 42}}{y : 42 \rightarrow \mathbb{0} \vdash y\ 42 : \mathbb{0}} \quad [\textsc{Var}]\ \dfrac{}{y : 42 \rightarrow \mathbb{0} \vdash y : 42 \rightarrow \mathbb{0}}}{y : 42 \rightarrow \mathbb{0} \vdash (y\ 42, y) : \mathbb{0}}}{y : 42 \rightarrow \mathbb{0} \vdash (y\ 42, y) : t}$$

What happens here is that, after decomposing the type of y into the two types $(42 \to 42) \wedge (42 \to 43)$ and $(42 \to 42) \wedge \neg(42 \to 43)$ in the [Let] rule, we are able to get rid of the case $(42 \to 42) \wedge (42 \to 43)$ as typing the application $y\ 42$ yields $\mathbb{0}$. We are thus left with the case $y : (42 \to 42) \wedge \neg(42 \to 43)$, where the type of y features a non-trivial negative arrow.

After reduction, our example becomes:

$$\texttt{let}\ y = \lambda x.x\ \texttt{in}\ (y\ 42, y) \rightsquigarrow ((\lambda x.x)\ 42, \lambda x.x) \rightsquigarrow (42, \lambda x.x)$$

The type $42 \times ((42 \to 42) \wedge \neg(42 \to 43))$ is not derivable anymore for $(42, \lambda x.x)$, because we have no way to derive a negative arrow type for $\lambda x.x$.

A way of interpreting this example is by seeing the term $y\ 42$ as a witness of the absurdity of deriving the type $42 \to 43$ for y, thus allowing to derive its negation $\neg(42 \to 43)$ using the union-elimination mechanism embedded in the [Let] rule. After reduction, the let-binding disappears, and thus the type $\neg(42 \to 43)$ is not derivable anymore for the λ-abstraction $\lambda x.x$.

In order to retrieve type preservation and prove the type safety theorem, we define an auxiliary type system, less practical but more powerful, that is able to derive negative arrows for λ-abstractions.

B.2 Preliminary definitions and lemmas

We start by proving some lemmas on $\vdash$ judgments.

Definition B.1 (Type scheme and environment substitution).

$$(\forall\vec{\alpha}.t)\phi \stackrel{\text{def}}{=} \forall\vec{\alpha}.t(\phi \setminus \vec{\alpha})$$

$$\Sigma\phi \stackrel{\text{def}}{=} \{(x : \sigma\phi) \mid (x : \sigma) \in \Sigma\}$$

Lemma B.2 (Type substitution). *Let Σ be an environment, e an expression, and t a type such that $\Sigma \vdash e : t$ is derivable. Let ϕ be a substitution. Then, $\Sigma\phi \vdash e : t\phi$ is derivable.*

Proof. We consider a derivation D of $\Sigma \vdash e : t$ and show that we can build a derivation D' of $\Sigma\phi \vdash e : t\phi$. We proceed by structural induction on the proof tree D.

For the case of a [Var] rule on x such that $\Sigma(x) = \forall\vec{\alpha}.t'$, and that performs a substitution ϕ' on t', we perform a substitution $(\phi|_{\vec{\alpha}}) \circ \phi'$ instead.

For the case of a [Let] rule that generalizes the type variables $\vec{\alpha}$, we apply the induction hypothesis on the first premise $\Sigma \vdash e_1 : s$ in order to derive $\Sigma(\phi \setminus \vec{\alpha}) \vdash e_1 : s(\phi \setminus \vec{\alpha})$. Note that $\Sigma(\phi \setminus \vec{\alpha}) = \Sigma\phi$ as we have $\vec{\alpha}\#\text{vars}(\Sigma)$. Also note that $s \leq \bigvee_{i \in I} s_i$ implies $s(\phi \setminus \vec{\alpha}) \leq \bigvee_{i \in I} s_i$ as subtyping is preserved by substitution, and that $\forall i \in I.\ \vec{\alpha}\#\text{vars}(s_i)$. We can thus conclude by induction on the other premises.

The other cases are straightforward. □

Definition B.3 (Application of a set of substitutions). We define the application of a set of substitutions Φ on a type t as follows:

$$t\Phi \stackrel{\text{def}}{=} \textstyle\bigwedge_{\phi \in \Phi} t\phi$$

Definition B.4 (Subtyping on type schemes and environments).

$$\forall\vec{\alpha}.t \leq \forall\vec{\alpha}'.t' \iff \forall\phi' \text{ s.t. } \text{dom}(\phi') \subseteq \vec{\alpha}'.\ \exists\Phi.\ (\forall\phi \in \Phi.\ \text{dom}(\phi) \subseteq \vec{\alpha}) \text{ and } t\Phi \leq t'\phi'$$

$$\Sigma \leq \Sigma' \iff \forall(x : \sigma') \in \Sigma'.\ \exists(x : \sigma) \in \Sigma.\ \sigma \leq \sigma'$$

Lemma B.5 (Monotonicity). *Let Σ be an environment, e an expression, and t a type such that $\Sigma \vdash e : t$ is derivable. Let Σ' be an environment such that $\Sigma' \leq \Sigma$. Then, $\Sigma' \vdash e : t$ is derivable.*

Proof. We consider a derivation D of $\Sigma \vdash e : t$ and show that we can build a derivation D' of $\Sigma' \vdash e : t$. We proceed by structural induction on the proof tree D.

For the case of a [Var] rule on x such that $\Sigma(x) = \forall\vec{\alpha}.t$ that performs a substitution ϕ on t, we can deduce from $\Sigma' \leq \Sigma$ that $\Sigma'(x) = \forall\vec{\alpha}'.t'$ for some $\vec{\alpha}'$ and t', and that there exists a set of substitutions Φ' such that $\forall\phi' \in \Phi'.\ \mathrm{dom}(\phi') \subseteq \vec{\alpha}'$ and $t'\Phi' \leq t\phi$. We can thus conclude with some [$\leq$], [$\wedge$] and [Var] rules.

Another interesting case is the case of a generalizing [Let]. If the set $\vec{\alpha}$ of generalized type variables conflicts with a type variable in $\mathrm{vars}(\Sigma')$, we use Lemma B.2 on the first premise in order to rename type variables in $\vec{\alpha}$ into fresh type variables.

The other cases are straightforward. □

B.3 Elimination of generalizations

First, we get rid of the generalizations that may happen in [Let] nodes. Intuitively, this can be done by using ad-hoc polymorphism (i.e. intersections) instead of parametric polymorphism.

Lemma B.6 (Elimination of quantifications). *Let Σ be an environment, x a variable, $\forall\vec{\alpha}.s$ a type-scheme, e an expression, and t a type. Let D be a derivation of $\Sigma, x : \forall\vec{\alpha}.s \vdash e : t$. Then, there exists a set of substitutions Φ such that $\forall\phi \in \Phi.\ dom(\phi) \subseteq \vec{\alpha}$ and $\Sigma, x : s\Phi \vdash e : t$ is derivable.*

Proof. We proceed by structural induction on the derivation D.

We consider the root of D:

[Const] We can directly conclude with an empty set of substitutions ($\Phi = \emptyset$).

[Var] We can directly conclude with the set of substitutions $\Phi = \{\phi\}$, with ϕ the substitution performed by this rule.

[$\wedge$] We apply the induction hypothesis to the two premises $\Sigma, x : \forall\vec{\alpha}.s \vdash e : t_1$ and $\Sigma, x : \forall\vec{\alpha}.s \vdash e : t_2$ in order to derive $\Sigma, x : s\Phi_1 \vdash e : t_1$ and $\Sigma, x : s\Phi_2 \vdash e : t_2$. We then consider the set of substitutions $\Phi = \Phi_1 \cup \Phi_2$, and derive $\Sigma, x : s\Phi \vdash e : t_1$ and $\Sigma, x : s\Phi \vdash e : t_2$ using Lemma B.5. We conclude with a [$\wedge$] node.

Other rules The others rules are similar to the [$\wedge$] case.

□

Definition B.7 (Generalization-free derivation). A derivation D of a judgment $\Sigma \vdash e : t$ is generalization-free if and only if every [Let] node in D uses $\vec{\alpha} = \emptyset$ (it does not generalize any type variable).

Lemma B.8 (Elimination of generalizations). *Let Σ be an environment, e an expression, and t a type. Let D be a derivation of $\Sigma \vdash e : t$. Then, there exists a generalization-free derivation of $\Sigma \vdash e : t$.*

Proof. We proceed by structural induction on e, and for a given e, by structural induction on D.

The interesting case is the case of a [Let]. If the [Let] has a generalization set $\vec{\alpha} \neq \emptyset$, then we apply Lemma B.6 on its premises $\Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t$ for $i \in I$ in order to derive, for each $i \in I$, a premise $\Sigma, x : (s \wedge s_i)\Phi_i \vdash e_2 : t$. We then consider $\Phi = \bigcup_{i\in I} \Phi_i$ and derive for each $i \in I$ the judgment $\Sigma, x : (s \wedge s_i)\Phi \vdash e_2 : t$ using Lemma B.5.

Then, we derive $\Sigma \vdash e_1 : s\Phi$ from the premise $\Sigma \vdash e_1 : s$ using Lemma B.2 and some [$\wedge$] nodes.

Finally, we apply the induction hypothesis on the derivations $\Sigma \vdash e_1 : s\Phi$ and $\Sigma, x : (s \wedge s_i)\Phi \vdash e_2 : t$ (for each $i \in I$) and use those derivations as premises of a non-generalizing [Let] node that derives $\Sigma \vdash \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 : t$.

The other cases are straightforward. □

B.4 Derivation of negative arrows

We will prove the type safety theorem by decomposing it into two lemmas: the type preservation (i.e., typeability is preserved by reduction) and the progress (i.e., any well-typed expression is either a value or can be reduced). However, the type preservation does not hold for our type system due to the impossibility to derive negative arrows for λ-abstraction.

$$[\textsc{Var-N}]\ \frac{\Sigma(x) = \forall\vec{\alpha}.t}{\Sigma \vdash_N x : \bigwedge_{i\in I} t\phi_i}\ \forall i \in I.\ \mathrm{dom}(\phi)_i \subseteq \vec{\alpha} \qquad [\textsc{Const-N}]\ \frac{}{\Sigma \vdash_N c : \boldsymbol{b_c}}$$

$$[\textsc{Let-N}]\ \frac{\Sigma \vdash_N e_1 : s \qquad (\forall i \in I) \quad \Sigma, x : s \wedge s_i \vdash_N e_2 : t}{\Sigma \vdash_N \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \ : t}\ s \leq \bigvee_{i\in I} s_i$$

$$[\to\text{I-N}]\ \frac{(\forall i \in I) \quad \Sigma, x : s_i \vdash_N e : t_i}{\Sigma \vdash_N \lambda x.e : \bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{j\in J} \neg(s'_j \to t'_j)}\ \begin{array}{l} I \neq \emptyset, \quad \forall i,j \in I.\ i \neq j \Rightarrow s_i \wedge s_j \simeq \mathbb{0} \\ (\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{j\in J} \neg(s'_j \to t'_j)) \not\simeq \mathbb{0} \end{array}$$

$$[\to\text{E-N}]\ \frac{\Sigma \vdash_N e_1 : t_1 \to t_2 \qquad \Sigma \vdash_N e_2 : t_1}{\Sigma \vdash_N e_1 e_2 : t_2} \qquad [\textsc{Choose-N}]\ \frac{\Sigma \vdash_N e_1 : t \qquad \Sigma \vdash_N e_2 : t}{\Sigma \vdash_N e_1 \oplus e_2 : t}$$

$$[\times\text{I-N}]\ \frac{\Sigma \vdash_N e_1 : t_1 \qquad \Sigma \vdash_N e_2 : t_2}{\Sigma \vdash_N (e_1, e_2) : t_1 \times t_2} \qquad [\times\text{E}_1\text{-N}]\ \frac{\Sigma \vdash_N e : t_1 \times t_2}{\Sigma \vdash_N \pi_1 e : t_1} \qquad [\times\text{E}_2\text{-N}]\ \frac{\Sigma \vdash_N e : t_1 \times t_2}{\Sigma \vdash_N \pi_2 e : t_2}$$

$$[\in\text{-N}]\ \frac{\Sigma \vdash_N e : s \qquad s \wedge \tau \not\simeq \mathbb{0} \Rightarrow \Sigma \vdash_N e_1 : t \qquad s \setminus \tau \not\simeq \mathbb{0} \Rightarrow \Sigma \vdash_N e_2 : t}{\Sigma \vdash_N (e{\in}\tau)\ ?\, e_1 : e_2 : t} \qquad [\leq\text{-N}]\ \frac{\Sigma \vdash_N e : t}{\Sigma \vdash_N e : t'}\ t \leq t'$$

Fig. 8. Declarative type system with negative arrows

A new type system with the possibility to derive negative arrow types for λ-abstractions is formalized in Figure 8. There are several things to note about this new type system. First, there is no intersection rule: instead, the intersection is embedded into the [Var-N] and [→I-N] rules. As shown in Lemma B.12 below, this can be done without loss of generality. The reason why the intersection rule has been eliminated is because it would make the type system unsafe in the presence of this new [→I-N] rule (this is explained later).

Second, the [Let-N] rule does not perform generalization. Again, the reason is that it would make the type system unsafe in the presence of the new [→I-N] rule.

Lastly, the [→I-N] rule allows deriving any conjunction of negative arrow types for a λ-abstraction, as long as these negative arrow types are compatible with the positive arrow types inferred. This is ensured by the side condition $(\bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{j\in J} \neg(s'_j \to t'_j)) \not\simeq \mathbb{0}$. This side condition prevents a [→I-N] node from deriving the type $\mathbb{0}$ for a λ-abstraction, which would be unsound. Note that this is also the reason why the intersection rule has been removed: it would make it possible to infer the type $\mathbb{0}$ for a λ-abstraction, as shown by the derivation below.

$$[\wedge]\ \frac{[\to\text{I-N}]\ \dfrac{[\textsc{Var-N}]\ \dfrac{}{x : \mathbb{0} \vdash_N x : \mathbb{0}}}{\varnothing \vdash_N \lambda x.x : (\mathbb{0} \to \mathbb{0}) \wedge \neg(\mathtt{false} \to \mathtt{false})} \qquad [\to\text{I-N}]\ \dfrac{[\textsc{Var-N}]\ \dfrac{}{x : \mathtt{false} \vdash_N x : \mathtt{false}}}{\varnothing \vdash_N \lambda x.x : \mathtt{false} \to \mathtt{false}}}{\varnothing \vdash_N \lambda x.x : \mathbb{0}}$$

Although it is more subtle, the side condition $\forall i, j \in I.\ i \neq j \Rightarrow s_i \wedge s_j \simeq \mathbb{0}$ has the same purpose: it prevents a potentially unsound intersection from happening. Consider the expression

$(\lambda y.(\lambda x.x))$ 42. As we have just seen, $\lambda x.x$ can be typed $\texttt{false} \to \texttt{false}$ using a [→I-N] node, and it can also be typed $\neg(\texttt{false} \to \texttt{false})$ using a different [→I-N] node. When typing the outer λ-abstraction $\lambda y.(\lambda x.x)$ using a [→I-N] node without this side condition, we may choose to type it twice for the same domain $\mathbb{1}$. This way, we may derive the type $(\mathbb{1} \to (\texttt{false} \to \texttt{false})) \wedge (\mathbb{1} \to (\neg(\texttt{false} \to \texttt{false})))$. The issue with this type is that it is equivalent to $\mathbb{1} \to ((\texttt{false} \to \texttt{false}) \wedge \neg(\texttt{false} \to \texttt{false})) \simeq \mathbb{1} \to \mathbb{0}$. Consequently, the type $\mathbb{0}$ can be derived for $(\lambda y.(\lambda x.x))$ 42, which is unsound. This issue can be avoided by forcing the domains explored for a λ-abstraction to be disjoint. By doing so, the different codomains are never intersected, regardless of the value to which this λ-abstraction is applied. This side condition is used in the proof of subject reduction (Theorem B.16, case [→E-N]).

Lemma B.9 (Monotonicity 2). *Let Σ be an environment, e an expression, and t a type such that $\Sigma \vdash_N e : t$ is derivable. Let Σ' be an environment such that $\Sigma' \leq \Sigma$. Then, $\Sigma' \vdash_N e : t$ is derivable.*

Proof. Similar to the proof of Lemma B.5. The case of a [Let-N] node does not require using a type substitution lemma as it does not generalize type variables. □

Definition B.10 (Normalized derivation). A derivation D of $\Sigma \vdash e : t$ is normalized if and only if D is generalization-free, and every $[\wedge]$ node appearing in D applies on an expression that is either a λ-abstraction or a variable.

Lemma B.11 (Normalization). *Let Σ be an environment, e an expression, and t a type such that $\Sigma \vdash e : t$. Then, there exists a normalized derivation of $\Sigma \vdash e : t$.*

Proof. First, we apply Lemma B.7 on D in order to get rid of generalizing [Let] nodes.

Now, we show that we can build a normalized derivation D' of $\Sigma \vdash e : t$. We proceed by structural induction on e, and for a given e, by structural induction on D.

The idea is to push $[\wedge]$ nodes towards the leaves, and stop when the $[\wedge]$ node has either only leaves premises or only [→I] premises.

The interesting case is the case of a $[\wedge]$ node with [Let] premises. In this case we apply the following transformation to D:

$$[\wedge]\ \dfrac{[\textsc{Let}]\ \dfrac{\Sigma \vdash e_1 : s \quad (\forall i \in I) \quad \Sigma, x : \forall \vec{\alpha}.s \wedge s_i \vdash e_2 : t}{\Sigma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2\ : t} \qquad [\textsc{Let}]\ \dfrac{\Sigma \vdash e_1 : s' \quad (\forall j \in J) \quad \Sigma, x : \forall \vec{\alpha}'.s' \wedge s'_j \vdash e_2 : t'}{\Sigma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2\ : t'}}{\Sigma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2\ : t \wedge t'}$$

$$\downarrow$$

$$[\textsc{Let}]\ \dfrac{[\wedge]\ \dfrac{\Sigma \vdash e_1 : s\phi \quad \Sigma \vdash e_1 : s'\phi}{\Sigma \vdash e_1 : s''} \qquad (\forall (i,j) \in I \times J) \qquad [\wedge]\ \dfrac{\begin{array}{c}\Sigma, x : \forall \vec{\alpha}''.s'' \wedge s_i \wedge s'_j \vdash e_2 : t \\ \Sigma, x : \forall \vec{\alpha}''.s'' \wedge s_i \wedge s'_j \vdash e_2 : t'\end{array}}{\Sigma, x : \forall \vec{\alpha}''.s'' \wedge s_i \wedge s'_j \vdash e_2 : t \wedge t'}}{\Sigma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2\ : t \wedge t'}$$

where ϕ is a renaming from type variables in $\vec{\alpha} \cup \vec{\alpha}'$ to fresh type variables, $s'' = (s \wedge s')\phi$, and $\vec{\alpha}'' = \{\phi(\alpha) \mid \alpha \in \vec{\alpha} \cup \vec{\alpha}'\}$.

All the premises in the resulting derivation can be constructed from premises of the initial derivation, using Lemma B.2 and Lemma B.5. We can then conclude this case by using the induction hypothesis on the premises of the [Let] node.

The other cases are straightforward. □

Lemma B.12 (Inclusion of $\vdash$ in $\vdash_N$). *Let Σ be an environment, e an expression, and t a type such that $\Sigma \vdash e : t$. Then, $\Sigma \vdash_N e : t$ is derivable.*

Proof. We consider a derivation D of $\Sigma \vdash e : t$ and show that we can build a derivation D' of $\Sigma \vdash_N e : t$.

We first apply Lemma B.11 on D in order to get a normalized derivation D'' of $\Sigma \vdash e : t$. We then proceed with a straighforward structural induction on D''. □

B.5 Type preservation

Lemma B.13 (Substitution). *Let Σ be an environment, x a variable, e, e' two expressions, and t, s two types. Let B be a derivation of $\Sigma, x : s \vdash_N e : t$, and A a derivation of $\Sigma \vdash_N e' : s$. Then, there exists a derivation of $\Sigma \vdash_N e\{e'/x\} : t$.*

Proof. We build a derivation $\Sigma \vdash_N e\{e'/x\} : t$ by structural induction on B.

In the case of a [Var-N] node applied on x, we just return the derivation A. Note that this [Var-N] rule cannot perform any substitution other than the identity because the type scheme $(\Sigma, x : s)(x)$ does not have quantified type variables.

All the other cases are straightforward using Lemma B.9. □

Lemma B.14 (Atomicity of value derivations). *Let Σ be an environment, and v a value. Let D be a derivation of $\Sigma \vdash_N v : s$. Let $\{t_i\}_{i\in I}$ a set of types such that $s \leq \bigvee_{i\in I} t_i$. Then, there exists for some $i \in I$ a derivation of $\Sigma \vdash_N v : s \wedge t_i$.*

Proof. We proceed by structural induction on D.

- If the root of D is a [$\rightarrow$I-N] node, then v is a λ-abstraction $\lambda x.e$, and s is a conjunction of positive and negative arrows. Let us show that we can find $i \in I$ such that $s \wedge t_i \geq s \wedge \bigwedge_{j\in J} \neg(s'_j \rightarrow t'_j)$ and $s \wedge \bigwedge_{j\in J} \neg(s'_j \rightarrow t'_j) \not\simeq \mathbb{0}$ for some $\{(s'_j, t'_j)\}_{j\in J}$.
  By contradiction, let us assume that it is not the case. Let us consider $i \in I$. We consider a set of types $\{s_k\}_{k\in K}$ such that $s \wedge t_i = \bigvee_{k\in K} s \wedge s_k$ ($s \wedge t_i$ can be written in disjunctive normal form), where every s_k is a conjunction of positive and negative arrows. We know that, for every $k \in K$, $s \wedge s_k \not\geq s \wedge s''$ for every conjunction of negative arrows s'' such that $s \wedge s'' \not\simeq \mathbb{0}$. This means that, for every $k \in K$, there exists a positive arrow $s''_k \rightarrow t''_k$ such that $s \wedge s_k \leq s \wedge (s''_k \rightarrow t''_k)$ and $(s''_k \rightarrow t''_k) \not\leq s$. Thus, we have $s \wedge t_i \leq s \wedge (\bigvee_{k\in K} s''_k \rightarrow t''_k)$ with $\bigvee_{k\in K} s''_k \rightarrow t''_k \not\leq s$.
  As this is true for every $i \in I$, we have $s \wedge \bigvee_{i\in I} t_i \leq s \wedge \bigvee_{i\in I}(\bigvee_{k\in K} s''_k \rightarrow t''_k) \not\leq s$, which contradicts the fact that $\{t_i\}_{i\in I}$ covers s. Thus, there exists some $i \in I$ and some types $\{(s''_k, t''_k)\}_{k\in K}$ such that $s \wedge t_i \geq s \wedge \bigwedge_{k\in K} \neg(s''_k \rightarrow t''_k) \not\simeq \mathbb{0}$.
  We can thus derive $\Sigma \vdash_N \lambda x.e : s \wedge \bigwedge_{k\in K} \neg(s''_k \rightarrow t''_k)$ using a [$\rightarrow$I-N] node, and conclude by inserting a [$\leq$-N] node at the root to derive $\Sigma \vdash_N \lambda x.e : s \wedge t_i$.
- The other cases are straightforward.

□

Proposition B.15. *Let Σ be an environment, v a value, and τ a test type. Let D be a derivation of $\Sigma \vdash_N v : \tau$. Then, we have the relation $v \in \tau$ (see Figure 9 for the definition of $\in$).*

Proof. Straightforward structural induction on D. Note that the case of λ-abstractions is trivial as arrows in τ can only be $\mathbb{0} \rightarrow \mathbb{1}$. □

Theorem B.16 (Type preservation). *Let e, e' be two expressions such that $e \rightsquigarrow e'$, and t be a type. Let D be a derivation of $\varnothing \vdash_N e : t$. Then, $\varnothing \vdash_N e' : t$ is derivable.*

Proof. We proceed by structural induction on D.

We consider the root of D:

[Const-N] Impossible case (e is not reducible).

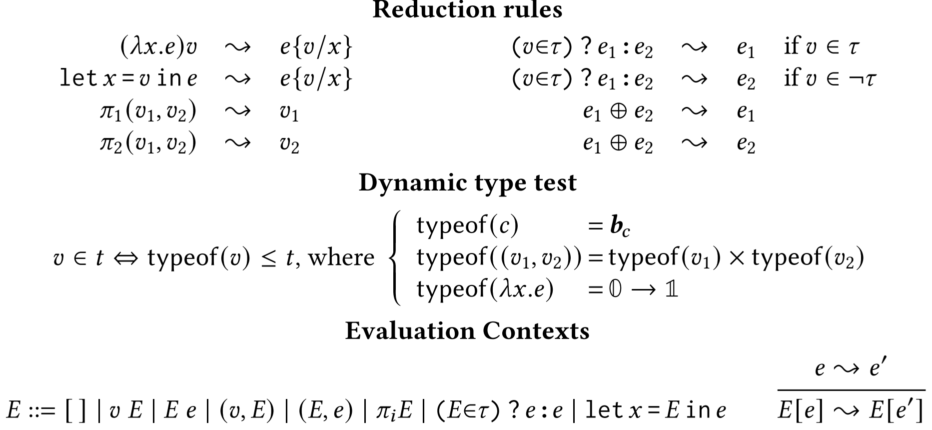

Fig. 9. Semantics of the source language

[→I-N] Impossible case (e is not reducible).
[Var-N] Impossible case (the rule cannot be applied under an empty environment).
[≤-N] By applying the induction hypothesis on the premise $\varnothing \vdash_N e : t'$ (with $t' \leq t$), we get a derivation of $\varnothing \vdash_N e' : t'$, thus we can derive $\varnothing \vdash_N e' : t$ by using a [≤-N] node.
[Choose-N] We have $e \equiv e_1 \oplus e_2$, and either $e' \equiv e_1$ or $e' \equiv e_2$. In any case, we can conclude with the corresponding premise.
[×I-N] If $e \equiv (v_1, e_2)$, then the reduction happens in e_2. In this case, we can conclude by using the induction hypothesis on the second premise. Otherwise, the reduction happens in e_1 with $e \equiv (e_1, e_2)$. In this case, we can conclude by using the induction hypothesis on the first premise.
[→E-N] We have $e \equiv e_1 e_2$. If either e_1 or e_2 is not a value, then we can conclude as in the previous case. Otherwise, we have $e \equiv (\lambda x.e_\lambda)v$ and $e' \equiv e_\lambda\{v/x\}$.
We have the following premises:
(1) $\varnothing \vdash_N \lambda x.e_\lambda : s \to t$
(2) $\varnothing \vdash_N v : s$
We can extract from (1) the collection of derivations typing the body e_λ: it gives us some derivations $\{A_i\}_{i\in I}$ of $x : s_i \vdash_N e_\lambda : t_i$ for $i \in I$, and such that $\forall i, j \in I.\ i \neq j \Rightarrow s_i \wedge s_j \simeq \mathbb{0}$ (all $\{s_i\}_{i\in I}$ are mutually disjoint) and $\bigwedge_{i\in I}(s_i \to t_i) \leq s \to t$.
By applying Lemma B.14 on the premise (2) and the set of types $\{s_i\}_{i\in I}$, we are able to derive a derivation B of $\varnothing \vdash_N v : s \wedge s_i$ for some $i \in I$. Using the fact that $\bigwedge_{i\in I}(s_i \to t_i) \leq s \to t$ and that all $\{s_i\}_{i\in I}$ are mutually disjoint, we can deduce $t_i \leq t$. By inserting a [≤-N] node at the root of B, we obtain a derivation B' of $\varnothing \vdash_N v : s_i$.
Using the substitution lemma (Lemma B.13) on A_i and B', we obtain a derivation of $\varnothing \vdash_N e_\lambda\{v/x\} : t_i$ (with, we recall, $t_i \leq t$). We obtain $\varnothing \vdash_N e_\lambda\{v/x\} : t$ by inserting a [≤-N] node at the root of this derivation.
[×E$_1$-N] We have $e \equiv \pi_1 e_\pi$. If e_π is not a value, then we can conclude by using the induction hypothesis on the premise. Otherwise, we have $e \equiv \pi_1(v_1, v_2)$.
We can extract from D a derivation A_1 of $\emptyset \vdash_N v_1 : s_1$ and a derivation A_2 of $\emptyset \vdash_N v_2 : s_2$ such that $s_1 \times s_2 \leq t_1 \times t_2$. We thus have $s_1 \leq t_1$, as s_2 cannot be $\mathbb{0}$ (this would contradict Proposition B.15). Therefore, we can conclude this case by using a [≤-N] node with the premise A_1 in order to derive $\emptyset \vdash_N v_1 : t_1$.
[×E$_2$-N] Similar to the previous case.

[Let-N] We have $e \equiv \texttt{let}\, x = e_1 \,\texttt{in}\, e_2$. If e_1 is not a value, then we can conclude by using the induction hypothesis on the first premise. Otherwise, we have $e \equiv \texttt{let}\, x = v_1 \,\texttt{in}\, e_2$ and $e' \equiv e_2\{v_1/x\}$.
We have the following premises:
(1) $\varnothing \vdash_N v_1 : s$
(2) $\forall i \in I.\ x : s \wedge s_i \vdash_N e_2 : t$
By applying Lemma B.14 on the premise (1) and the set of types $\{s_i\}_{i\in I}$, we are able to derive a derivation A of $\varnothing \vdash_N v_1 : s \wedge s_i$ for some $i \in I$.
Using the substitution lemma (Lemma B.13) on the premise $x : s \wedge s_i \vdash_N e_2 : t$ and A, we obtain a derivation of $\varnothing \vdash_N e_2\{v_1/x\} : t$.

[∈-N] We have $e \equiv (e_\in \in \tau)\ ?\, e_1 : e_2$. If e_1 is not a value, then we can conclude by using the induction hypothesis on the first premise. Otherwise, we have $e \equiv (v \in \tau)\ ?\, e_1 : e_2$ and either $e' \equiv e_1$ or $e' \equiv e_2$.
If $e' \equiv e_1$, it means that $v \in \tau$. We can deduce that the first premise $\varnothing \vdash_N e : s$ is such that $s \wedge \tau \not\simeq \mathbb{0}$: otherwise, we would have $\varnothing \vdash_N e : \neg\tau$ and we would deduce $v \in \neg\tau$ by Proposition B.15. We can thus conclude using the with the premise $\varnothing \vdash_N e_1 : t$.
We conclude similarly for the case $e' \equiv e_2$.

□

B.6 Progress

Theorem B.17. *Let e be an expression and t a type. Let D be a derivation of $\varnothing \vdash_N e : t$. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof. We proceed by structural induction on D.
We consider the root of D:

[Const-N] Trivial (e is a value).
[→I-N] Trivial (e is a value).
[Var-N] Impossible case (the rule cannot be applied under an empty environment).
[≤-N] By using the induction hypothesis on the premise.
[Choose-N] Trivial ($e \equiv e_1 \oplus e_2$ and thus can be reduced).
[×I-N] We have $e \equiv (e_1, e_2)$.

- If e_1 is not a value, we know by applying the induction hypothesis on the first premise that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $([\,], e_2)$.
- If e_1 is a value, then we can apply the induction hypothesis on the second premise. It gives that either e_2 is a value or it can be reduced. We can easily conclude in both cases: if e_2 is a value, then e is also a value, otherwise, e can be reduced under the evaluation context $(e_1, [\,])$.

[→E-N] We have $e \equiv e_1 e_2$, with $\varnothing \vdash_N e_1 : s \to t$ and $\varnothing \vdash_N e_2 : s$.

- If e_1 is not a value, we know by applying the induction hypothesis on the first premise that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $[\,]e_2$.
- If e_1 is a value and e_2 is not a value, we know by applying the induction hypothesis on the second premise that e_2 can be reduced. Thus, e can also be reduced under the evaluation context $e_1[\,]$.
- If e_1 and e_2 are both values, we can apply Proposition B.15 on e_1: as $\varnothing \vdash_N e_1 : \mathbb{0} \to \mathbb{1}$, it implies that $e_1 \in \mathbb{0} \to \mathbb{1}$ and thus $e_1 \equiv \lambda x.\ e_\lambda$. Thus, e can be reduced using the β-reduction rule.

[×E$_1$-N] We have $e \equiv \pi_1 e_\pi$, with $\varnothing \vdash_N e_\pi : t \times s$. By applying the induction hypothesis on the premise, we know that e_π is either a value or it can be reduced. If e_1 can be reduced, then e can

also be reduced under the evaluation context $\pi_1[\,]$. Otherwise, as $\varnothing \vdash_N e_\pi : \mathbb{1} \times \mathbb{1}$, we can apply Proposition B.15 on it, yielding $e_\pi \in \mathbb{1} \times \mathbb{1}$. Thus, $e_p i \equiv (v_1, v_2)$ for some values v_1 and v_2, and consequently e can be reduced with the reduction rule for left projection.

[$\times$E$_2$-N] Similar to the previous case.

[Let-N] We have $e \equiv \texttt{let}\, x = e_1 \,\texttt{in}\, e_2$, with $\varnothing \vdash_N e_1 : s$ and $\forall i \in I.\ x : s \wedge s_i \vdash_N e_2 : t$.

- If e_1 is not a value, we know by applying the induction hypothesis on the first premise that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $\texttt{let}\, x = [\,] \,\texttt{in}\, e_2$.
- If e_1 is a value, e can be reduced with the reduction rule for let-bindings.

[$\in$-N] We have $e \equiv (e_\in \in \tau)\ ?\ e_1 : e_2$.

- If $e_\in$ is not a value, we know by applying the induction hypothesis on the first premise that $e_\in$ can be reduced. Thus, e can also be reduced under the evaluation context $([\,] \in \tau)\ ?\ e_1 : e_2$.
- If $e_\in$ is a value, then we have either $e_\in \in \tau$ or $e_\in \in \neg\tau$. Thus, e can be reduced with one of the reduction rules for type-cases.

□

B.7 Type safety

We can now prove type safety by combining the type preservation and progress theorems.

Lemma B.18 (Type safety for $\vdash_N$). *For every expression e and type t, if $\varnothing \vdash_N e : t$, then for any e' such that $e \rightsquigarrow^* e'$, we have either $e' \rightsquigarrow^* v$ for some v such that $\varnothing \vdash_N v : t$, or $e' \rightsquigarrow^\infty$.*

Proof. Direct consequence of Theorem B.16 and Theorem B.17. □

Theorem B.19 (Type safety). *For every expression e and test type τ, if $\varnothing \vdash e : \tau$, then for any e' such that $e \rightsquigarrow^* e'$, we have either $e' \rightsquigarrow^* v$ for some $v \in \tau$, or $e' \rightsquigarrow^\infty$.*

Proof. From $\varnothing \vdash e : \tau$, we can derive $\varnothing \vdash_N e : \tau$ using Lemma B.12. By applying Lemma B.18 on this new derivation, we get either:

- $e' \rightsquigarrow^\infty$ (in which case we can directly conclude), or
- $e' \rightsquigarrow^* v$ for some v such that $\varnothing \vdash_N v : \tau$. Then, by Proposition B.15, we get $v \in \tau$, which allows us to conclude.

□

C Proof of equivalence between the declarative and algorithmic systems

$$[\textsc{Const-A}]\ \frac{}{\Sigma \vdash_{\mathcal{A}} [c \mid \varnothing] : \boldsymbol{b}_c} \qquad [\textsc{Var-A}]\ \frac{\Sigma(x) = \forall\vec{\alpha}.t}{\Sigma \vdash_{\mathcal{A}} [x \mid \mathtt{var}(\phi)] : t\phi}\ \mathrm{dom}(\phi) \subseteq \vec{\alpha}$$

$$[\textsc{Let-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e_1 \mid a] : s \qquad \vec{\alpha} = \mathrm{vars}(s) \setminus (\mathrm{vars}(\Sigma) \cup \bigcup_{i\in I} \mathrm{vars}(s_i)) \qquad (\forall i \in I)\quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash_{\mathcal{A}} [e_2 \mid a_i] : t_i}{\Sigma \vdash_{\mathcal{A}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \mathtt{let}\ (a, \{(s_i, a_i)\}_{i\in I})] : \bigvee_{i\in I} t_i}\ s \leq \bigvee_{i\in I} s_i$$

$$[\to\text{I-A}]\ \frac{\Sigma, x : s \vdash_{\mathcal{A}} [e \mid a] : t}{\Sigma \vdash_{\mathcal{A}} [\lambda x.e \mid \lambda(s, a)] : s \to t} \qquad [\to\text{E-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e_1 \mid a_1] : t_1 \quad \Sigma \vdash_{\mathcal{A}} [e_2 \mid a_2] : t_2}{\Sigma \vdash_{\mathcal{A}} [e_1 e_2 \mid @(a_1, a_2, t)] : t}\ t_1 \leq t_2 \to t$$

$$[\times\text{I-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e_1 \mid a_1] : t_1 \qquad \Sigma \vdash_{\mathcal{A}} [e_2 \mid a_2] : t_2}{\Sigma \vdash_{\mathcal{A}} [(e_1, e_2) \mid \times(a_1, a_2)] : t_1 \times t_2} \qquad [\times\text{E}_1\text{-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e \mid a] : s}{\Sigma \vdash_{\mathcal{A}} [\pi_1 e \mid \pi(a, t)] : t}\ t \leq (s \times \mathbb{1})$$

$$[\in\text{-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e \mid a] : s \qquad b_1 = \mathtt{skip} \Rightarrow s \leq \neg\tau \qquad b_2 = \mathtt{skip} \Rightarrow s \leq \tau \qquad \Sigma \vdash_{\mathcal{A}} [e_1 \mid b_1] : t_1 \qquad \Sigma \vdash_{\mathcal{A}} [e_2 \mid b_2] : t_2}{\Sigma \vdash_{\mathcal{A}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \in(a, b_1, b_2)] : t_1 \vee t_2}$$

$$[\textsc{Choose-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e_1 \mid a_1] : t_1 \quad \Sigma \vdash_{\mathcal{A}} [e_2 \mid a_2] : t_2}{\Sigma \vdash_{\mathcal{A}} [e_1 \oplus e_2 \mid \oplus(a_1, a_2)] : t_1 \vee t_2} \qquad [\wedge\text{-A}]\ \frac{(\forall i \in I) \quad \Sigma \vdash_{\mathcal{A}} [e \mid a_i] : t_i}{\Sigma \vdash_{\mathcal{A}} [e \mid \bigwedge(\{a_i\}_{i\in I})] : \bigwedge_{i\in I} t_i}\ I \neq \emptyset$$

$$[\textsc{Skip-A}]\ \frac{}{\Sigma \vdash_{\mathcal{A}} [e \mid \mathtt{skip}] : \mathbb{0}} \qquad [\textsc{Type-A}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e \mid a] : t}{\Sigma \vdash_{\mathcal{A}} [e \mid \mathtt{type}(a)] : t}$$

Fig. 10. Algorithmic type system

Lemma C.1 (Inclusion of $\vdash_{\mathcal{A}}$ in $\vdash$). *Let e be an expression, Σ a type environment, a an annotation tree, and t a type. We have* $\Sigma \vdash_{\mathcal{A}} [e \mid a] : t \implies \Sigma \vdash e : t$.

Proof. We consider a derivation D of $\Sigma \vdash_{\mathcal{A}} [e \mid a] : t$ and build a derivation $\Sigma \vdash e : t$ by structural induction on D.

[Const-A] Trivial.

[Var-A] Trivial.

[Let-A] We use the induction hypothesis on the first premise to derive $\Sigma \vdash e_1 : s$, and on the other premises to derive, for each $i \in I$, $\Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t_i$. Using [≤] nodes, we can derive $\Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : \bigvee_{i\in I} t_i$ for each $i \in I$. We then use these premises in a [Let] rule to derive $\Sigma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \bigvee_{i\in I} t_i$ (we can easily check that $\vec{\alpha}$ satisfies the side-conditions).

[→I-A] Trivial.

[→E-A] We know that $t_1 \leq t_2 \to t$. By using the induction hypothesis on the first premise followed by a [≤] node, we can thus derive $\Sigma \vdash e_1 : t_2 \to t$. By using the induction hypothesis on the second premise, we can derive $\Sigma \vdash e_2 : t_2$. We can conclude with [→E] node.

[×I-A] Trivial.

[×E$_1$-A] We know that $s \leq t \times \mathbb{1}$. We can thus conclude by using the induction hypothesis on the premise followed by a [≤] node.

[$\times$E$_2$-A] Similar to the previous case.
[$\in$-A] We apply the induction hypothesis on the first premise, and:

- If $s \wedge \tau \not\simeq \mathbb{0}$, we apply the induction hypothesis on the $\Sigma \vdash_{\mathcal{A}} [e_1 \mid a_1] : t_1$ premise of the corresponding [Type-A] node, and derive $\Sigma \vdash e_1 : t_1 \vee t_2$ using a [$\leq$] node, and
- If $s \wedge \neg\tau \not\simeq \mathbb{0}$, we apply the induction hypothesis on the $\Sigma \vdash_{\mathcal{A}} [e_2 \mid a_2] : t_2$ premise of the corresponding [Type-A] node, and derive $\Sigma \vdash e_2 : t_1 \vee t_2$ using a [$\leq$] node.

We can then conclude with a [$\in$] node.
[Choose-A] Trivial.
[$\wedge$-A] Trivial.

□

Lemma C.2 (Inclusion of $\vdash$ in $\vdash_{\mathcal{A}}$). *Let e be an expression, Σ a type environment, and t a type. If $\Sigma \vdash e : t$, then there exists an annotation tree a and a type t' such that $t' \leq t$ and $\Sigma \vdash_{\mathcal{A}} [e \mid a] : t'$.*

Proof. We consider a derivation D of $\Sigma \vdash et$ and build an annotation tree a and type t' such that $\Sigma \vdash_{\mathcal{A}} [e \mid a] : t'$. We proceed by structural induction on e, and for a given e, by structural induction on D.

[$\leq$] Trivial.
[Const] Trivial.
[Var] Trivial.
[Let] We use the induction hypothesis on the first premise to get a and $s' \leq s$ such that $\Sigma \vdash_{\mathcal{A}} [e_1 \mid a] : s'$.
For each $i \in I$, we consider the premise $\Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t$. We know, from the side-conditions of [Let], that $\vec{\alpha} \cap \text{vars}(s \wedge s_i) \subseteq \text{vars}(s) \setminus (\text{vars}(\Sigma) \cup \bigcup_{i\in I} \text{vars}(s_i))$, and thus we can deduce that $\forall\vec{\alpha}.s \wedge s_i \leq \forall\vec{\alpha}'.s' \wedge s_i$ with $\vec{\alpha}' = \text{vars}(s') \setminus (\text{vars}(\Sigma) \cup \bigcup_{i\in I} \text{vars}(s_i))$. Using Lemma B.5 on $\Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash e_2 : t$, we get $\Sigma, x : \forall\vec{\alpha}'.s' \wedge s_i \vdash e_2 : t$, and by using the induction hypothesis we get an annotation a_i and type $t_i \leq t$ such that $\Sigma, x : \forall\vec{\alpha}'.s' \wedge s_i \vdash_{\mathcal{A}} [e_2 \mid a_i] : t_i$.
Finally, we build the annotation $\texttt{let}\,(a, \{a_i\}_{i\in I})$ to derive the type $\bigvee_{i\in I} t_i \leq t$ for $\texttt{let}\, x = e_1 \,\texttt{in}\, e_2$.
[$\to$I] Trivial.
[$\to$E] We have the premises $\Sigma \vdash e_1 : s \to t$ and $\Sigma \vdash e_2 : s$. By using the induction hypothesis on these premises, we get some derivations $\Sigma \vdash_{\mathcal{A}} [e_1 \mid a_1] : t_1$ and $\Sigma \vdash_{\mathcal{A}} [e_2 \mid a_2] : t_2$ with $t_1 \leq s \to t$ and $t_2 \leq s$. We thus have $t_1 \leq s \to t \leq t_2 \to t$. We can thus conclude by building the annotation $@(a_1, a_2, t)$.
[$\times$I] Trivial.
[$\times$E$_1$] We have the premise $\Sigma \vdash e' : t \times \mathbb{1}$. By using the induction hypothesis on this premise, we get a derivation $\Sigma \vdash_{\mathcal{A}} [e' \mid a] : s$ with $s \leq t \times \mathbb{1}$. We can thus conclude by building the annotation $\pi(a, t)$.
[$\times$E$_2$] Similar to the previous case.
[$\in$] We apply the induction hypothesis on the first premise to get a derivation $\Sigma \vdash_{\mathcal{A}} [e' \mid a] : s'$ with $s' \leq s$, and:

- If $s \wedge \tau \simeq \mathbb{0}$, then we have $s' \leq \neg\tau$, and we pose $b_1 = \texttt{skip}$. Otherwise, we apply the induction hypothesis on the $\Sigma \vdash e_1 : t$ premise to derive $\Sigma \vdash_{\mathcal{A}} [e_1 \mid a_1] : t_1$ with $t_1 \leq t$, and we pose $b_1 = \texttt{type}(a_1)$.
- If $s \wedge \neg\tau \simeq \mathbb{0}$, then we have $s' \leq \tau$, and we pose $b_2 = \texttt{skip}$. Otherwise, we apply the induction hypothesis on the $\Sigma \vdash e_2 : t$ premise to derive $\Sigma \vdash_{\mathcal{A}} [e_2 \mid a_2] : t_2$ with $t_2 \leq t$, and we pose $b_2 = \texttt{type}(a_2)$.

We can then conclude by building an annotation $\in(a, b_1, b_2)$.
[Choose] Trivial.

[$\wedge$] Trivial.

□

THEOREM C.3 (EQUIVALENCE WITH THE DECLARATIVE TYPE SYSTEM). *Let e be an expression, Σ a type environment, and t a type.*

$$\Sigma \vdash e : t \quad \Leftrightarrow \quad \exists a, t'.\ \Sigma \vdash_{\mathcal{A}} [e \mid a] : t' \text{ and } t' \leq t$$

PROOF. Direct consequence of Lemma C.1 and Lemma C.2. □

D Full type reconstruction system

$$
\begin{array}{lrl}
\textbf{Partial annotations} & \bar{a} ::= & a \mid \mathtt{untyp} \mid \bar{\mathtt{var}}(\phi) \mid \bar{@}(\bar{a},\bar{a},t) \mid \bar{\pi}(\bar{a},t) \mid \bar{\times}(\bar{a},\bar{a}) \\
 & & \mid \bar{\in}(\bar{a},\bar{b},\bar{b}) \mid \bar{\lambda}(t,\bar{a}) \mid \bar{\mathtt{let}}\ (\bar{a},\bar{U}) \mid \bar{\bigwedge}(\bar{I}) \\
\textbf{Branch partial annotations} & \bar{b} ::= & \bar{\mathtt{maybe}}(\bar{a}) \mid \bar{\mathtt{type}}(\bar{a}) \mid b \\
\textbf{Inter partial annotations} & \bar{I} ::= & \{\bar{a},\ldots,\bar{a}\} \\
\textbf{Union partial annotations} & \bar{U} ::= & \{(t,\bar{a}),\ldots,(t,\bar{a})\}
\end{array}
$$

The initial partial annotation for an expression e is generated as follows:

$$
\begin{array}{rclrcl}
\mathtt{init}(c) & = & \varnothing & \mathtt{init}(\lambda x.e) & = & \bar{\lambda}(\alpha, \mathtt{init}(e)) \ \text{ with } \alpha \text{ fresh} \\
\mathtt{init}(x) & = & \bar{\mathtt{var}}(\phi) \text{ with } \phi \text{ fresh} & \mathtt{init}(e_1 e_2) & = & \bar{@}(\mathtt{init}(e_1), \mathtt{init}(e_2), \alpha) \ \text{ with } \alpha \text{ fresh} \\
\mathtt{init}((e_1,e_2)) & = & \bar{\times}(\mathtt{init}(e_1), \mathtt{init}(e_2)) & \mathtt{init}(\pi_i e) & = & \bar{\pi}(\mathtt{init}(e), \alpha) \ \text{ with } \alpha \text{ fresh}
\end{array}
$$

$$
\begin{array}{rcl}
\mathtt{init}((e_0 \in \tau)\ ?\ e_1 : e_2) & = & \bar{\in}(\mathtt{init}(e_0), \bar{\mathtt{maybe}}(\mathtt{init}(e_1)), \bar{\mathtt{maybe}}(\mathtt{init}(e_2))) \\
\mathtt{init}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) & = & \bar{\mathtt{let}}\ (\mathtt{init}(e_1), \{(s_i, \mathtt{init}(e_2))\}_{i \in I}) \text{ where } \{s_i\}_{i\in I} = \text{decomposition}(x)
\end{array}
$$

The annotation refinement algorithm generates results of the following syntax:

$$\textbf{Result}\ \mathbb{R} ::= \mathtt{Ok}(a,t) \mid \mathtt{Fail} \mid \mathtt{Subst}(\Phi, \bar{a}, \bar{a})$$

Its full set of deduction rules is given below, ordered by decreasing priority (the first rule that can apply must be applied).

D.1 Threading rules

$$
[\textsc{Subst-R}]\ \frac{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}_1, \bar{a}_2) \qquad \Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\bar{a}_1\phi \mid \phi \in \Phi\} \cup \{\bar{a}_2\})] \Rightarrow \mathbb{R}}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}}\ \forall \phi \in \Phi.\ \text{dom}(\phi) \# \text{vars}(\Sigma)
$$

$$
[\textsc{Propagate-R}]\ \frac{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathbb{R}} \qquad [\textsc{Branch}_1\textsc{-R}]\ \frac{\Sigma \vdash_{\mathcal{R}} [e \mid \mathtt{skip}] \Rightarrow \mathbb{R}}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \mathtt{skip}] \Rightarrow \mathbb{R}}
$$

$$
[\textsc{Branch}_2\textsc{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Ok}(a,t)}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{\mathtt{type}}(\bar{a})] \Rightarrow \mathtt{Ok}(\mathtt{type}(a), t)} \qquad [\textsc{Branch}_3\textsc{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Fail}}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{\mathtt{type}}(\bar{a})] \Rightarrow \mathtt{Fail}}
$$

$$
[\textsc{Branch}_4\textsc{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}', \bar{a}'')}{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{\mathtt{type}}(\bar{a})] \Rightarrow \mathtt{Subst}(\Phi, \bar{\mathtt{type}}(\bar{a}'), \bar{\mathtt{type}}(\bar{a}''))}
$$

D.2 Structural rules

$$
[\textsc{Ok}_1\textsc{-R}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e \mid a] : t}{\Sigma \vdash_{\mathcal{R}} [e \mid a] \Rightarrow \mathtt{Ok}(a,t)} \qquad [\textsc{Ok}_2\textsc{-R}]\ \frac{\Sigma \vdash_{\mathcal{A}} [e \mid b] : t}{\Sigma \vdash_{\mathcal{R}} [e \mid b] \Rightarrow \mathtt{Ok}(b,t)}
$$

$$
[\textsc{Fail-R}]\ \frac{}{\Sigma \vdash_{\mathcal{R}} [e \mid \mathtt{untyp}] \Rightarrow \mathtt{Fail}}
$$

$$
[\textsc{Var-R}]\ \frac{\Sigma(x) = \forall \vec{\alpha}.t \qquad \Sigma \vdash_{\mathcal{R}} [x \mid \mathtt{var}(\phi|_{\vec{\alpha}})] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [x \mid \bar{\mathtt{var}}(\phi)] \Rightarrow \mathbb{R}}
$$

Pairs.

$$[\times\mathrm{I}_1\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_i \mid \bar{a}_i] \Rightarrow \mathtt{Fail}}{\Sigma \vdash_{\mathcal{R}} [(e_1, e_2) \mid \bar{\times}(\bar{a}_1, \bar{a}_2)] \Rightarrow \mathtt{Fail}}\ i \in \{1,2\}$$

$$[\times\mathrm{I}_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}_1', \bar{a}_1'')}{\Sigma \vdash_{\mathcal{R}} [(e_1, e_2) \mid \bar{\times}(\bar{a}_1, \bar{a}_2)] \Rightarrow \mathtt{Subst}(\Phi, \bar{\times}(\bar{a}_1', \bar{a}_2), \bar{\times}(\bar{a}_1'', \bar{a}_2))}$$

$$[\times\mathrm{I}_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}_2', \bar{a}_2'')}{\Sigma \vdash_{\mathcal{R}} [(e_1, e_2) \mid \bar{\times}(\bar{a}_1, \bar{a}_2)] \Rightarrow \mathtt{Subst}(\Phi, \bar{\times}(\bar{a}_1, \bar{a}_2'), \bar{\times}(\bar{a}_1, \bar{a}_2''))}$$

$$[\times\mathrm{I}_4\text{-R}]\ \frac{\begin{array}{c}\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Ok}(a_1, t_1) \\ \Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \mathtt{Ok}(a_2, t_2) \qquad \Sigma \vdash_{\mathcal{R}} [(e_1, e_2) \mid \times(a_1, a_2)] \Rightarrow \mathbb{R}\end{array}}{\Sigma \vdash_{\mathcal{R}} [(e_1, e_2) \mid \bar{\times}(\bar{a}_1, \bar{a}_2)] \Rightarrow \mathbb{R}}$$

Non-deterministic choices. Similar to the rules for pairs.

Projections.

$$[\times\mathrm{E}_1\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Fail}}{\Sigma \vdash_{\mathcal{R}} [\pi_i e \mid \bar{\pi}(\bar{a}, t)] \Rightarrow \mathtt{Fail}}\ i \in \{1,2\}$$

$$[\times\mathrm{E}_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}', \bar{a}'')}{\Sigma \vdash_{\mathcal{R}} [\pi_i e \mid \bar{\pi}(\bar{a}, t)] \Rightarrow \mathtt{Subst}(\Phi, \bar{\pi}(\bar{a}', t), \bar{\pi}(\bar{a}'', t))}\ i \in \{1,2\}$$

$$[\times\mathrm{E}_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Ok}(a, s) \qquad \{s \dot{\leq} t \times \mathbb{1}\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [\pi_1 e \mid \bar{\pi}(\bar{a}, t)] \Rightarrow \mathtt{Subst}(\Phi, \pi(a, t), \mathtt{untyp})}$$

$$[\times\mathrm{E}_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Ok}(a, s) \qquad \{s \dot{\leq} \mathbb{1} \times t\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [\pi_2 e \mid \bar{\pi}(\bar{a}, t)] \Rightarrow \mathtt{Subst}(\Phi, \pi(a, t), \mathtt{untyp})}$$

Abstractions.

$$[\to\mathrm{I}_1\text{-R}]\ \frac{\Sigma, x : s \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Fail}}{\Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \bar{\lambda}(s, \bar{a})] \Rightarrow \mathtt{Fail}}$$

$$[\to\mathrm{I}_2\text{-R}]\ \frac{\Sigma, x : s \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}', \bar{a}'')}{\Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \bar{\lambda}(s, \bar{a})] \Rightarrow \mathtt{Subst}(\Phi, \bar{\lambda}(s, \bar{a}'), \bar{\lambda}(s, \bar{a}''))}$$

$$[\to\mathrm{I}_3\text{-R}]\ \frac{\Sigma, x : s \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Ok}(a, t) \qquad \Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \lambda(s, a)] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [\lambda x.e \mid \bar{\lambda}(s, \bar{a})] \Rightarrow \mathbb{R}}$$

Applications.

$$[\to\text{E}_1\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_i \mid \bar{a}_i] \Rightarrow \texttt{Fail}}{\Sigma \vdash_{\mathcal{R}} [e_1 e_2 \mid \bar{@}(\bar{a}_1, \bar{a}_2, t)] \Rightarrow \texttt{Fail}}\ i \in \{1,2\}$$

$$[\to\text{E}_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \texttt{Subst}(\Phi, \bar{a}'_1, \bar{a}''_1)}{\Sigma \vdash_{\mathcal{R}} [e_1 e_2 \mid \bar{@}(\bar{a}_1, \bar{a}_2, t)] \Rightarrow \texttt{Subst}(\Phi, \bar{@}(\bar{a}'_1, \bar{a}_2, t), \bar{@}(\bar{a}''_1, \bar{a}_2, t))}$$

$$[\to\text{E}_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \texttt{Subst}(\Phi, \bar{a}'_2, \bar{a}''_2)}{\Sigma \vdash_{\mathcal{R}} [e_1 e_2 \mid \bar{@}(\bar{a}_1, \bar{a}_2, t)] \Rightarrow \texttt{Subst}(\Phi, \bar{@}(\bar{a}_1, \bar{a}'_2, t), \bar{@}(\bar{a}_1, \bar{a}''_2, t))}$$

$$[\to\text{E}_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \texttt{Ok}(a_1, t_1) \qquad \Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \texttt{Ok}(a_2, t_2) \qquad \{t_1 \mathrel{\dot{\leq}} t_2 \to t\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [e_1 e_2 \mid \bar{@}(\bar{a}_1, \bar{a}_2, t)] \Rightarrow \texttt{Subst}(\Phi, @(a_1, a_2, t), \texttt{untyp})}$$

Type-cases.

$$[\in_1\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Fail}}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}_2)] \Rightarrow \texttt{Fail}}$$

$$[\in_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Subst}(\Phi, \bar{a}', \bar{a}'')}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}_2)] \Rightarrow \texttt{Subst}(\Phi, \bar{\in}(\bar{a}', \bar{b}_1, \bar{b}_2), \bar{\in}(\bar{a}'', \bar{b}_1, \bar{b}_2))}$$

$$[\in_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \{s \mathrel{\dot{\leq}} \neg\tau\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{\texttt{maybe}}(\bar{a}_1), \bar{b}_2)] \Rightarrow \texttt{Subst}(\Phi, \in(a, \texttt{skip}, \bar{b}_2), \in(a, \bar{\texttt{type}}(\bar{a}_1), \bar{b}_2))}$$

$$[\in_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \{s \mathrel{\dot{\leq}} \tau\} \Vdash \Phi}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{\texttt{maybe}}(\bar{a}_2))] \Rightarrow \texttt{Subst}(\Phi, \in(a, \bar{b}_1, \texttt{skip}), \in(a, \bar{b}_1, \bar{\texttt{type}}(\bar{a}_2)))}$$

$$[\in_5\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \Sigma \vdash^*_{\mathcal{R}} [e_i \mid \bar{b}_i] \Rightarrow \texttt{Fail}}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}_2)] \Rightarrow \texttt{Fail}}\ i \in \{1,2\}$$

$$[\in_6\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{b}_1] \Rightarrow \texttt{Subst}(\Phi, \bar{b}'_1, \bar{b}''_1)}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}_2)] \Rightarrow \texttt{Subst}(\Phi, \bar{\in}(\bar{a}, \bar{b}'_1, \bar{b}_2), \bar{\in}(\bar{a}, \bar{b}''_1, \bar{b}_2))}$$

$$[\in_7\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{b}_2] \Rightarrow \texttt{Subst}(\Phi, \bar{b}'_2, \bar{b}''_2)}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}_2)] \Rightarrow \texttt{Subst}(\Phi, \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}'_2), \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}''_2))}$$

$$[\in_8\text{-R}]\ \frac{\begin{array}{c}\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \texttt{Ok}(a, s) \qquad \Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{b}_1] \Rightarrow \texttt{Ok}(b_1, t_1) \\ \Sigma \vdash^*_{\mathcal{R}} [e_2 \mid \bar{b}_2] \Rightarrow \texttt{Ok}(b_2, t_2) \qquad \Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \in(a, b_1, b_2)] \Rightarrow \mathbb{R}\end{array}}{\Sigma \vdash_{\mathcal{R}} [(e{\in}\tau)\,?\,e_1 : e_2 \mid \bar{\in}(\bar{a}, \bar{b}_1, \bar{b}_2)] \Rightarrow \mathbb{R}}$$

The recursive calls on branches are handled by the threading rules [Branch$_1$-R], [Branch$_2$-R], [Branch$_3$-R] and [Branch$_4$-R].

Let-bindings.

$$[\textsc{Let}_1\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Fail}}{\Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (\bar{a}_1, \bar{U})] \Rightarrow \mathtt{Fail}}$$

$$[\textsc{Let}_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}'_1, \bar{a}''_1)}{\Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (\bar{a}_1, \bar{U})] \Rightarrow \mathtt{Subst}(\Phi, \bar{\mathtt{let}}\ (\bar{a}'_1, \bar{U}), \bar{\mathtt{let}}\ (\bar{a}''_1, \bar{U}))}$$

$$[\textsc{Let}_3\text{-R}]\ \frac{\begin{array}{c}\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Ok}(a_1, s) \\ s \wedge s' \simeq \mathbb{0} \qquad \Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (a_1, \bar{U})] \Rightarrow \mathbb{R}\end{array}}{\Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (\bar{a}_1, \{(s', \bar{a}_2)\} \cup \bar{U})] \Rightarrow \mathbb{R}}$$

$$[\textsc{Let}_4\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Ok}(a_1, s) \qquad \Sigma, x : \forall\vec{\alpha}.s \wedge s' \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \mathtt{Fail}}{\Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (\bar{a}_1, \{(s', \bar{a}_2)\} \cup \bar{U})] \Rightarrow \mathtt{Fail}}\ (\star)$$

$$[\textsc{Let}_5\text{-R}]\ \frac{\begin{array}{c}\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Ok}(a_1, s) \qquad \Sigma, x : \forall\vec{\alpha}.s \wedge s' \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_2] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}'_2, \bar{a}''_2) \\ \mathbb{R} = \mathtt{Subst}(\Phi, \bar{\mathtt{let}}\ (a_1, \{(s', \bar{a}'_2)\} \cup \bar{U}), \bar{\mathtt{let}}\ (a_1, \{(s', \bar{a}''_2)\} \cup \bar{U}))\end{array}}{\Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (\bar{a}_1, \{(s', \bar{a}_2)\} \cup \bar{U})] \Rightarrow \mathbb{R}}\ (\star)$$

$$[\textsc{Let}_6\text{-R}]\ \frac{\begin{array}{c}\Sigma \vdash^*_{\mathcal{R}} [e_1 \mid \bar{a}_1] \Rightarrow \mathtt{Ok}(a_1, s) \qquad (\forall i \in I) \quad \Sigma, x : \forall\vec{\alpha}.s \wedge s_i \vdash^*_{\mathcal{R}} [e_2 \mid \bar{a}_i] \Rightarrow \mathtt{Ok}(a_i, t) \\ \Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \mathtt{let}\ (a_1, \{(s_i, a_i)\}_{i\in I})] \Rightarrow \mathbb{R}\end{array}}{\Sigma \vdash_{\mathcal{R}} [\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \mid \bar{\mathtt{let}}\ (\bar{a}_1, \{(s_i, \bar{a}_i)\}_{i\in I})] \Rightarrow \mathbb{R}}\ (\star)$$

where, in $(\star)$ rules, we have $\vec{\alpha} = \begin{cases} \mathrm{vars}(s) \setminus \mathrm{vars}(\Sigma) & \text{if no value restriction, or if } \mathtt{value}(e_1) \\ \emptyset & \text{otherwise} \end{cases}$

Intersections.

$$[\wedge_1\text{-R}]\ \frac{}{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\})] \Rightarrow \mathtt{Fail}} \qquad [\wedge_2\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Fail} \qquad \Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\bar{I})] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\bar{a}\} \cup \bar{I})] \Rightarrow \mathbb{R}}$$

$$[\wedge_3\text{-R}]\ \frac{\Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}] \Rightarrow \mathtt{Subst}(\Phi, \bar{a}', \bar{a}'')}{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\bar{a}\} \cup \bar{I})] \Rightarrow \mathtt{Subst}(\Phi, \bar{\bigwedge}(\{\bar{a}'\} \cup \bar{I}), \bar{\bigwedge}(\{\bar{a}''\} \cup \bar{I}))}$$

$$[\wedge_4\text{-R}]\ \frac{(\forall i \in I) \quad \Sigma \vdash^*_{\mathcal{R}} [e \mid \bar{a}_i] \Rightarrow \mathtt{Ok}(a_i, t_i) \qquad \Sigma \vdash_{\mathcal{R}} [e \mid \bigwedge(\{a_i\}_{i\in I})] \Rightarrow \mathbb{R}}{\Sigma \vdash_{\mathcal{R}} [e \mid \bar{\bigwedge}(\{\bar{a}_i\}_{i\in I})] \Rightarrow \mathbb{R}}$$

D.3 Practical aspects and optimizations

The annotation reconstruction algorithm can infer parametric types (for generic functions) and intersection types (for overloaded functions). However, a naive implementation may generate redundant intersections and introduce useless type variables, resulting in performance issues, especially when typing λ-abstractions with functional parameters. In this appendix, we describe several key optimizations that have been implemented to mitigate these issues.

D.3.1 Minimization of type variables. The annotation reconstruction algorithm relies on tallying in order to solve subtyping constraints. The solution of these tallying instances may introduce new type variables that are not strictly necessary and will pollute our context. This is an issue when typing large programs as the number of tallying instances generated may be high, leading to an increasing complexity in our types. In order to tackle this issue, we implemented two simple optimizations whose goal is to avoid introducing useless type variables.

Reusing the same type variables when solving constraints. When solving tallying constraints, the solutions should be expressed by reusing the type variables of the input. For instance, for a tallying instance $\{\alpha \dot{\leq} \mathtt{int}\}$, the set of solutions should be captured by the substitution $\phi = \{\alpha \rightsquigarrow \mathtt{int} \wedge \alpha\}$ rather than a substitution $\phi' = \{\alpha \rightsquigarrow \mathtt{int} \wedge \alpha'\}$ where α' is a fresh type variable. This way, we avoid introducing new type variables after each call to the tallying algorithm, and the different branches of the annotation tree will reuse the same type variables when possible (for instance, the domain of a λ-abstraction may get intersected with different types in different branches, but its initial top-level variable α will stay the same).

Getting rid of univariant type variables. Type variables that only appear in a covariant position (respectively, those that only appear in a contravariant position) can be removed by substituting them by $\mathbb{0}$ (respectively, by $\mathbb{1}$). For instance, consider an application $f\ x$ with $f : \forall\alpha.\alpha \to \alpha$ and $x : \mathtt{int}$. When reconstructing the annotation tree for this application, the following constraint is generated: $\alpha' \to \alpha' \dot{\leq} \mathtt{int} \to \beta$ (with β a type variable capturing the result of the application).

The solutions to this tallying instance are characterized by the principal substitution $\{\beta \rightsquigarrow \mathtt{int} \vee \beta,\ \alpha' \rightsquigarrow \mathtt{int} \vee (\beta \wedge \alpha')\}$. This yields for the application the type $\mathtt{int} \vee \beta$. However, as β only appears in this result in a covariant position, and as it is not bound to the current type environment, it can be substituted by $\mathbb{0}$, yielding for this application the simpler type $\mathtt{int}$.

D.3.2 Branch pruning. Each time a constraint is generated and solved using tallying (for projections, applications, and type-cases), the reconstruction algorithm backtracks and inserts an intersection node to consider all the different solutions. This behavior is necessary in order to infer types that capture overloaded behaviors, and because there is not necessarily a “better” solution to a tallying instance. However, some branches generated by the reconstruction algorithm may be redundant. For instance, consider the following expression, which simulates JavaScript’s `typeof` function:

$$\lambda x.(x{\in}\mathtt{int})\,?\,\text{"}\mathtt{number}\text{"}{:}(x{\in}\mathtt{bool})\,?\,\text{"}\mathtt{boolean}\text{"}{:}\ \ldots$$

Before reconstructing annotations for the branch of a type-case, the reconstruction algorithm solves a constraint to determine if under some contexts this branch can be skipped. It then backtracks – in our example, just before the λ-abstraction – and inserts an intersection annotation to explore these contexts, as well as the initial context under which the branch must be typed. The branches generated for our example are illustrated in Figure 11.

We can see in this figure that some of the branches explored are redundant (in the sense that they cover a domain that has already been covered by some other branches). For instance, the domain $\alpha \wedge \mathtt{bool}$ is explored twice. In order to avoid this combinatorial explosion, we can implement branch pruning.

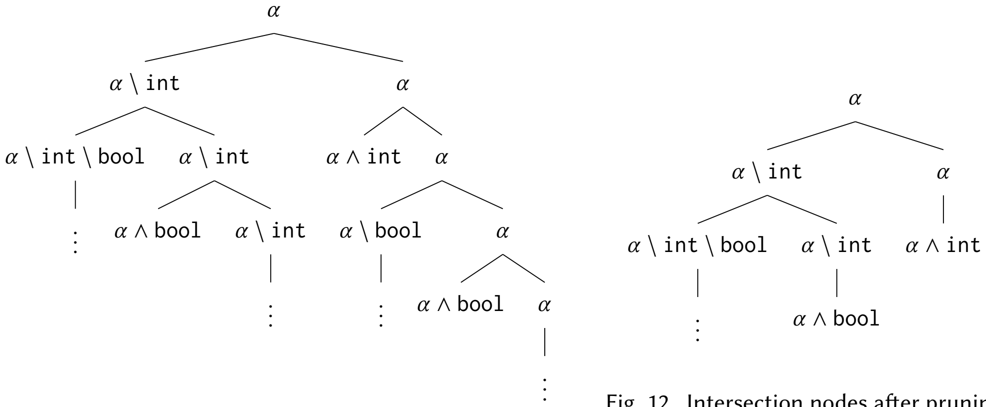

Fig. 11. Intersection nodes generated for the `typeof` function (nodes are labeled by the domain covered by the branch)

Fig. 12. Intersection nodes after pruning

The first step is to fix the order of exploration of the branches: more specific branches (i.e. those that are more likely to yield a precise type) will be explored first. In Figure 11, this corresponds to a depth-first search from left to right. Then, the second step consists in checking, before exploring a branch, if the domain it covers has already been covered by (the union of the domains covered by) the previously-explored branches. If it is the case, then the branch can be pruned. In our example, this eliminates all redundant branches, as illustrated in Figure 12.

Note that in our case, our expression is only composed of one λ-abstraction, so the nodes can be labeled with just one type corresponding to the domain covered by the branch for this λ-abstraction. In the general case, when we have nested λ-abstractions, nodes should be labeled with environments instead.

All these optimizations have been implemented in our prototype MLsem, presented in Section 5.

E Definition and proofs of opaque data types

For simplicity, we consider in this appendix a unique opaque data type $\#(.)$. We first extend the interpretation domain $\mathcal{D}$ and interpretation $[\![.]\!]$ to support $\#(.)$ types. The interpretation $[\![.]\!]_F$ we define is parametrized by a function $F : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ that defines the interpretation of $\#(.)$ types.

The *interpretation domain* $\mathcal{D}$ is the set of finite terms d produced inductively by the following grammar:

$$d ::= c^L \mid (d,d)^L \mid \{(d,\partial),\ldots,(d,\partial)\}^L \mid \#(d)^L$$
$$\partial ::= d \mid \Omega$$

Let $F : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$. Let $[\![.]\!]_F : \mathcal{T} \to \mathcal{D}$ be an interpretation satisfying the following equations:

$$[\![\mathbb{0}]\!]_F = \emptyset \qquad [\![\alpha]\!]_F = \{d \mid \alpha \in \mathsf{tags}(d)\} \qquad [\![t_1 \vee t_2]\!]_F = [\![t_1]\!]_F \cup [\![t_2]\!]_F$$
$$[\![b]\!]_F = \mathbb{B}(b) \qquad [\![\neg t]\!]_F = \mathcal{D} \setminus [\![t]\!]_F \qquad [\![t_1 \wedge t_2]\!]_F = [\![t_1]\!]_F \cap [\![t_2]\!]_F$$
$$[\![t_1 {\to} t_2]\!]_F = \{R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d,\partial) \in R.\ d \in [\![t_1]\!]_F \implies \partial \in [\![t_2]\!]_F\}$$
$$[\![t_1 \times t_2]\!]_F = [\![t_1]\!]_F \times [\![t_2]\!]_F \qquad [\![\#(t)]\!]_F = \{\#(d) \mid d \in F([\![t]\!]_F)\} \qquad [\![\#]\!]_F = \{\#(d) \mid d \in \mathcal{D}\}$$

We define the *subtyping* relation $\leq_F$ as $t_1 \leq_F t_2 \overset{\text{def}}{\iff} [\![t_1]\!]_F \subseteq [\![t_2]\!]_F$, and the *equivalence* relation $\simeq_F$ as $t_1 \simeq_F t_2 \overset{\text{def}}{\iff} [\![t_1]\!]_F = [\![t_2]\!]_F$.

E.1 General case (invariant parameter)

In this section, we make no particular assumption over F. We extend the subtyping algorithm with this case for $\#(.)$ types:

$$\bigwedge_{p \in P} \#(t_p) \leq \bigvee_{n \in N} \#(s_n) \quad \Leftrightarrow \quad \exists p \in P.\ \exists n \in N.\ t_p \simeq s_n$$

Theorem E.1 (Soundness of the characterization).

$$\bigwedge_{p \in P} \#(t_p) \leq_F \bigvee_{n \in N} \#(s_n) \quad \Longleftarrow \quad \exists p \in P.\ \exists n \in N.\ t_p \simeq_F s_n$$

Proof. Let $p \in P$ and $n \in N$ such that $t_p \simeq_F s_n$.

$$\begin{aligned}
& t_p \simeq_F s_n \\
\Rightarrow\ & [\![t_p]\!]_F = [\![s_n]\!]_F \\
\Rightarrow\ & F([\![t_p]\!]_F) = F([\![s_n]\!]_F) \\
\Rightarrow\ & \{\#(d) \mid d \in F([\![t_p]\!]_F)\} = \{\#(d) \mid d \in F([\![s_n]\!]_F)\} \\
\Rightarrow\ & [\![\#(t_p)]\!]_F = [\![\#(s_n)]\!]_F \\
\Rightarrow\ & \bigcap_{p \in P} [\![\#(t_p)]\!]_F \subseteq \bigcup_{n \in N} [\![\#(s_n)]\!]_F \\
\Rightarrow\ & [\![\bigwedge_{p \in P} \#(t_p)]\!]_F \subseteq [\![\bigvee_{n \in N} \#(s_n)]\!]_F \\
\Rightarrow\ & \bigwedge_{p \in P} \#(t_p) \leq_F \bigvee_{n \in N} \#(s_n)
\end{aligned}$$

□

The soundness of the whole subtyping algorithm can be derived from [Castagna and Xu 2011, Theorem 3.16] and Theorem E.1.

E.2 Monotonic case (covariant parameter)

In this section, we assume that F is monotonic, that is: $\forall D, D' \subseteq \mathcal{D}.\ D \subseteq D' \implies F(D) \subseteq F(D')$. We extend the subtyping algorithm with this case for #(.) types:

$$\bigwedge_{p \in P} \#(t_p) \leq \bigvee_{n \in N} \#(s_n) \quad \Leftrightarrow \quad \exists p \in P.\ \exists n \in N.\ t_p \leq s_n$$

THEOREM E.2 (SOUNDNESS OF THE CHARACTERIZATION).

$$\bigwedge_{p \in P} \#(t_p) \leq_F \bigvee_{n \in N} \#(s_n) \quad \Longleftarrow \quad \exists p \in P.\ \exists n \in N.\ t_p \leq_F s_n$$

PROOF. Let $p \in P$ and $n \in N$ such that $t_p \leq_F s_n$.

$$\begin{array}{lcr} & t_p \leq_F s_n & \\ \Rightarrow & [\![t_p]\!]_F \subseteq [\![s_n]\!]_F & \\ \Rightarrow & F([\![t_p]\!]_F) \subseteq F([\![s_n]\!]_F) & \text{(by monotonicity)} \\ \Rightarrow & \{\#(d) \mid d \in F([\![t_p]\!]_F)\} \subseteq \{\#(d) \mid d \in F([\![s_n]\!]_F)\} & \\ \Rightarrow & [\![\#(t_p)]\!]_F \subseteq [\![\#(s_n)]\!]_F & \\ \Rightarrow & \bigcap_{p \in P} [\![\#(t_p)]\!]_F \subseteq \bigcup_{n \in N} [\![\#(s_n)]\!]_F & \\ \Rightarrow & [\![\bigwedge_{p \in P} \#(t_p)]\!]_F \subseteq [\![\bigvee_{n \in N} \#(s_n)]\!]_F & \\ \Rightarrow & \bigwedge_{p \in P} \#(t_p) \leq_F \bigvee_{n \in N} \#(s_n) & \end{array}$$

□

E.3 Monotonic ∩-preserving case

In this section, we assume that F is monotonic, and that it preserves ∩: $\forall D, D' \subseteq \mathcal{D}.\ D \subseteq D' \implies F(D) \subseteq F(D')$, and $\forall D, D' \subseteq \mathcal{D}.\ F(D \cap D') = F(D) \cap F(D')$. We extend the subtyping algorithm with this case for #(.) types:

$$\bigwedge_{p \in P} \#(t_p) \leq \bigvee_{n \in N} \#(s_n) \quad \Leftrightarrow \quad \exists n \in N.\ \bigwedge_{p \in P} t_p \leq s_n$$

THEOREM E.3 (SOUNDNESS OF THE CHARACTERIZATION).

$$\bigwedge_{p \in P} \#(t_p) \leq_F \bigvee_{n \in N} \#(s_n) \quad \Longleftarrow \quad \exists n \in N.\ \bigwedge_{p \in P} t_p \leq_F s_n$$

Proof. Let $n \in N$ such that $\bigwedge_{p \in P} t_p \leq s_n$.

$$
\begin{array}{lll}
 & \bigwedge_{p\in P} t_p \leq_F s_n & \\
\Rightarrow & [\![\bigwedge_{p\in P} t_p]\!]_F \subseteq [\![s_n]\!]_F & \\
\Rightarrow & \bigcap_{p\in P} [\![t_p]\!]_F \subseteq [\![s_n]\!]_F & \\
\Rightarrow & F(\bigcap_{p\in P} [\![t_p]\!]_F) \subseteq F([\![s_n]\!]_F) & \text{(by monotonicity)} \\
\Rightarrow & \bigcap_{p\in P} F([\![t_p]\!]_F) \subseteq F([\![s_n]\!]_F) & \text{(by } \cap\text{-preservation)} \\
\Rightarrow & \{\#(d) \mid d \in \bigcap_{p\in P} F([\![t_p]\!]_F)\} \subseteq \{\#(d) \mid d \in F([\![s_n]\!]_F)\} & \\
\Rightarrow & [\![\#(t_p)]\!]_F \subseteq [\![\#(s_n)]\!]_F & \\
\Rightarrow & \bigcap_{p\in P} [\![\#(t_p)]\!]_F \subseteq \bigcup_{n\in N} [\![\#(s_n)]\!]_F & \\
\Rightarrow & [\![\bigwedge_{p\in P} \#(t_p)]\!]_F \subseteq [\![\bigvee_{n\in N} \#(s_n)]\!]_F & \\
\Rightarrow & \bigwedge_{p\in P} \#(t_p) \leq_F \bigvee_{n\in N} \#(s_n) &
\end{array}
$$

□

E.4 Monotonic $\cap$-$\mathbb{0}$-preserving case

In this section, we assume that F is monotonic, and that it preserves $\cap$: $\forall D, D' \subseteq \mathcal{D}.\ D \subseteq D' \implies F(D) \subseteq F(D')$, $\forall D, D' \subseteq \mathcal{D}.\ F(D \cap D') = F(D) \cap F(D')$, and $F(\emptyset) = \emptyset$. We extend the subtyping algorithm with this case for $\#(.)$ types:

$$\bigwedge_{p\in P} \#(t_p) \leq \bigvee_{n\in N} \#(s_n) \quad\Leftrightarrow\quad \bigwedge_{p\in P} t_p \leq \mathbb{0} \quad \text{or} \quad \exists n \in N.\ \bigwedge_{p\in P} t_p \leq s_n$$

Theorem E.4 (Soundness of the characterization).

$$\bigwedge_{p\in P} \#(t_p) \leq_F \bigvee_{n\in N} \#(s_n) \quad\Longleftarrow\quad \bigwedge_{p\in P} t_p \leq_F \mathbb{0} \quad \textit{or} \quad \exists n \in N.\ \bigwedge_{p\in P} t_p \leq_F s_n$$

Proof. If $\bigwedge_{p\in P} t_p \leq_F \mathbb{0}$, we have $\bigwedge_{p\in P} \#(t_p) \leq_F \#(\bigwedge_{p\in P} t_p) \leq_F \#(\mathbb{0}) \leq_F \mathbb{0}$. Otherwise, the proof is similar to the proof of the previous case. □

E.5 Monotonic $\cup$-preserving case

In this section, we assume that F is monotonic, and that it preserves $\cup$: $\forall D, D' \subseteq \mathcal{D}.\ D \subseteq D' \implies F(D) \subseteq F(D')$, and $\forall D, D' \subseteq \mathcal{D}.\ F(D \cup D') = F(D) \cup F(D')$. We extend the subtyping algorithm with this case for $\#(.)$ types:

$$\bigwedge_{p\in P} \#(t_p) \leq \bigvee_{n\in N} \#(s_n) \quad\Leftrightarrow\quad \exists p \in P.\ t_p \leq \bigvee_{n\in N} s_n$$

Theorem E.5 (Soundness of the characterization).

$$\bigwedge_{p\in P} \#(t_p) \leq_F \bigvee_{n\in N} \#(s_n) \quad \Longleftarrow \quad \exists p \in P.\ t_p \leq_F \bigvee_{n\in N} s_n$$

Proof. Let $p \in P$ such that $t_p \leq_F \bigvee_{n\in N} s_n$.

$$\begin{array}{lrcl}
 & t_p & \leq_F & \bigvee_{n\in N} s_n \\
\Rightarrow & [\![t_p]\!]_F & \subseteq & [\![\bigvee_{n\in N} s_n]\!]_F \\
\Rightarrow & [\![t_p]\!]_F & \subseteq & \bigcup_{n\in N} [\![s_n]\!]_F \\
\Rightarrow & F([\![t_p]\!]_F) & \subseteq & F(\bigcup_{n\in N} [\![s_n]\!]_F) \qquad \text{(by monotonicity)} \\
\Rightarrow & F([\![t_p]\!]_F) & \subseteq & \bigcup_{n\in N} F([\![s_n]\!]_F) \qquad \text{(by $\cup$-preservation)} \\
\Rightarrow & \{\#(d) \mid d \in F([\![t_p]\!]_F)\} & \subseteq & \{\#(d) \mid d \in \bigcup_{n\in N} F([\![s_n]\!]_F)\} \\
\Rightarrow & [\![\#(t_p)]\!]_F & \subseteq & [\![\#(s_n)]\!]_F \\
\Rightarrow & \bigcap_{p\in P} [\![\#(t_p)]\!]_F & \subseteq & \bigcup_{n\in N} [\![\#(s_n)]\!]_F \\
\Rightarrow & [\![\bigwedge_{p\in P} \#(t_p)]\!]_F & \subseteq & [\![\bigvee_{n\in N} \#(s_n)]\!]_F \\
\Rightarrow & \bigwedge_{p\in P} \#(t_p) & \leq_F & \bigvee_{n\in N} \#(s_n)
\end{array}$$

□

E.6 Monotonic ∪-𝟙-preserving case

In this section, we assume that F is monotonic, and that it preserves $\cup$ and $\mathbb{1}$: $\forall D, D' \subseteq \mathcal{D}.\ D \subseteq D' \implies F(D) \subseteq F(D')$, $\forall D, D' \subseteq \mathcal{D}.\ F(D \cup D') = F(D) \cup F(D')$, and $F(\mathcal{D}) = \mathcal{D}$. We extend the subtyping algorithm with this case for #(.) types:

$$\bigwedge_{p\in P} \#(t_p) \leq \bigvee_{n\in N} \#(s_n) \quad \Leftrightarrow \quad \mathbb{1} \leq \bigvee_{n\in N} s_n \ \text{ or } \ \exists p \in P.\ t_p \leq \bigvee_{n\in N} s_n$$

Theorem E.6 (Soundness of the characterization).

$$\bigwedge_{p\in P} \#(t_p) \leq_F \bigvee_{n\in N} \#(s_n) \quad \Longleftarrow \quad \mathbb{1} \leq_F \bigvee_{n\in N} s_n \ \textit{ or } \ \exists p \in P.\ t_p \leq_F \bigvee_{n\in N} s_n$$

Proof. If $\mathbb{1} \leq_F \bigvee_{n\in N} s_n$, we have $\# \leq_F \#(\mathbb{1}) \leq_F \#(\bigvee_{n\in N} s_n) \leq_F \bigvee_{n\in N} \#(s_n)$. Otherwise, the proof is similar to the proof of the previous case. □

E.7 Monotonic ∩-∪-preserving case (*tagged value*)

In this section, we assume that F is monotonic, and that it preserves $\cap$ and $\cup$: $\forall D, D' \subseteq \mathcal{D}.\ D \subseteq D' \implies F(D) \subseteq F(D')$, and $\forall D, D' \subseteq \mathcal{D}.\ F(D \cap D') = F(D) \cap F(D')$, and $\forall D, D' \subseteq \mathcal{D}.\ F(D \cup D') = F(D) \cup F(D')$. We extend the subtyping algorithm with this case for #(.) types:

$$\bigwedge_{p \in P} \#(t_p) \leq \bigvee_{n \in N} \#(s_n) \quad \Leftrightarrow \quad \bigwedge_{p \in P} t_p \leq \bigvee_{n \in N} s_n$$

THEOREM E.7 (SOUNDNESS OF THE CHARACTERIZATION).

$$\bigwedge_{p \in P} \#(t_p) \leq_F \bigvee_{n \in N} \#(s_n) \quad \Longleftarrow \quad \bigwedge_{p \in P} t_p \leq_F \bigvee_{n \in N} s_n$$

PROOF. Similar to the previous cases. □

F Additional code examples

Original `filtermap` function from [Schimpf et al. 2023]:

```
spec filtermap(fun((T) -> boolean()), [T]) -> [T]
  ; (fun((T) -> {true, U} | false), [T]) -> [U]
  ; (fun((T) -> {true, U} | boolean()), [T]) -> [T | U].
filtermap(_F, []) -> [];
filtermap(F, [X|XS]) ->
  case F(X) of
    false -> filtermap(F, XS );
    true -> [X | filtermap(F, XS)];
    {true, Y} -> [Y | filtermap(F, XS)]
end.
```

Translation of `filtermap` for MLsem:

```
val filtermap :
    (('t -> ((true, 'u) | false), ['t*]) -> ['u*])
  & (('t -> ((true, 'u) | bool), ['t*]) -> [('t | 'u)*])
let filtermap (f, l) =
    match l with
    | [] -> []
    | x::xs ->
        match f x with
        | false -> filtermap (f, xs)
        | true -> x::(filtermap (f, xs))
        | (true, y) -> y::(filtermap (f, xs))
        end
    end
```

Auxiliary definitions for the bal function from [OCaml 2023]:

```
val invalid_arg : string -> empty
val (<) : int -> int -> bool
val (<=) : int -> int -> bool
val (>) : int -> int -> bool
val (>=) : int -> int -> bool

type t('a) =
    Nil | (t('a), Key, 'a, t('a), int)

let height (x: t('a)) =
    match x with
    | :Nil -> 0
    | (_,_,_,_,h) -> h
    end

let create l x d r =
    let hl = height l in
    let hr = height r in
    (l, x, d, r, (if hl >= hr then hl + 1 else hr + 1))
```

Auxiliary definitions for the 14 examples from [Tobin-Hochstadt and Felleisen 2010]:

```
let and_ = fun (x, y) ->
        if x is true then if y is true then x else false else false
let not_ = fun x -> if x is true then false else true
let or_ =  fun (x,y) ->
    not_ (and_ (not_ x, not_ y))

let is_string = fun x ->
        if x is string then true else false
let is_int = fun x ->
        if x is int then true else false

val strlen : string -> int
val add : int -> int -> int
val add1 : int -> int
val f : (int | string) -> int
val g : (int, int) -> int
```

Translation of bal for MLsem:

```
(* Uncomment to type-check with type narrowing disabled *)
(* #type_narrowing = false *)

val bal : t('a) -> Key -> 'a -> t('a) -> t('a)
let bal l x d r =
(* let bal (l:t('a)) (x: Key) (d:'a) (r:t('a)) : t('a) = *)
    let hl = match l with :Nil -> 0 | (_,_,_,_,h) -> h end in
    let hr = match r with :Nil -> 0 | (_,_,_,_,h) -> h end in
    if hl > (hr + 2) then
    match l with
    | :Nil -> invalid_arg "Map.bal"
    | (ll, lv, ld, lr, _) ->
        if (height ll) >= (height lr) then
            create ll lv ld (create lr x d r)
        else match lr with
        | :Nil -> invalid_arg "Map.bal"
        | (lrl, lrv, lrd, lrr, _)->
            create (create ll lv ld lrl) lrv lrd (create lrr x d r)
        end
    end
    else if hr > (hl + 2) then
    match r with
    | :Nil -> invalid_arg "Map.bal"
    | (rl, rv, rd, rr, _) ->
        if (height rr) >= (height rl) then
            create (create l x d rl) rv rd rr
        else match rl with
        | :Nil -> invalid_arg "Map.bal"
        | (rll, rlv, rld, rlr, _) ->
            create (create l x d rll) rlv rld (create rlr rv rd rr)
        end
    end
    else (l, x, d, r, (if hl >= hr then hl + 1 else hr + 1))
```

Translation of the 14 examples from [Tobin-Hochstadt and Felleisen 2010] for MLsem:

```
let example1 = fun (x:any) ->
    if x is int then add1 x else 0
let example2 = fun (x:string|int) ->
    if x is int then add1 x else strlen x
let example3 = fun (x: any) ->
    if x is (any \ false) then (x,x) else false
let example4 = fun (x : any) ->
    if or_ (is_int x, is_string x) is true then x else 'A'
let example5 = fun (x : any) -> fun (y : any) ->
    if and_ (is_int x, is_string y) is true then
    add x (strlen y) else 0
val example6 : (int -> string -> int) & (string -> any -> int)
let example6 = fun x -> fun y ->
    if and_ (is_int x, is_string y) is true then
    add  x (strlen y) else strlen x
let example7 = fun (x : any) -> fun (y : any) ->
    if (if is_int x is true then is_string y else false) is true then
    add x (strlen y) else 0
let example8 = fun (x : any) ->
    if or_ (is_int x, is_string x) is true then true else false
let example9 = fun (x : any) ->
    if (if is_int x is true then is_int x else is_string x) is true
    then f x else 0
let example10 = fun (p : (any,any)) ->
    if is_int (fst p) is true then add1 (fst p) else 7
let example11 = fun (p : (any, any)) ->
    if and_ (is_int (fst p), is_int (snd p)) is true then g p else No
let example12 = fun (p : (any, any)) ->
    if is_int (fst p) is true then true else false
let example13 =
    fun (x : any) ->
        fun (y : any) ->
            if and_ (is_int x, is_string y) is true then 1
            else if is_int x is true then 2
            else 3
let example14 = fun (input : int|string) ->
    fun (extra : (any, any)) ->
        if and_(is_int input , is_int(fst extra)) is true then
            add input (fst extra)
        else if is_int(fst extra) is true then
            add (strlen input) (fst extra)
        else 0
```

Unannotated versions are similar but with no type annotation on the parameters of functions.