Polymorphic Type Inference for Dynamic Languages
Reconstructing Types for Systems Combining Parametric, Ad-Hoc, and Subtyping Polymorphism

GIUSEPPE CASTAGNA, CNRS, Université Paris Cité, France
MICKAËL LAURENT, Université Paris Cité, France
KIM NGUYEN, Université Paris-Saclay, France

We present a type system that combines, in a controlled way, first-order polymorphism with intersection types, union types, and subtyping, and prove its safety. We then define a type reconstruction algorithm that is sound and terminating. This yields a system in which unannotated functions are given polymorphic types (thanks to Hindley-Milner) that can express the overloaded behavior of the functions they type (thanks to the intersection introduction rule) and that are deduced by applying advanced techniques of type narrowing (thanks to the union elimination rule). This makes the system a prime candidate to type dynamic languages.

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

Additional Key Words and Phrases: polymorphism, subtyping, union types, intersection types, type-case, dynamic languages, type systems, type reconstruction.

1 INTRODUCTION

Typing dynamic languages is a challenging endeavour even for very simple pieces of code. For instance, JavaScript’s logical or operator “||” behaves like the following function (also in JavaScript):

1 function lOr (x, y) {
2   if (x) { return x; } else { return y; }
3 }

A naive type for this function is (Bool, Bool) → Bool, which states that 1Or is a function that takes two Boolean arguments and returns a Boolean result. This however is an overly restrictive type, that does not account for the fact that in JavaScript logical operators such as 1Or can be applied to any pairs of arguments, not just to Boolean ones. JavaScript distinguishes two kinds of values: eight “falsy” values (i.e., false, "", 0, −0, 0n, undefined, null, and NaN) and the “truthy” values (all the others). The expression if executes the else code if and only if the tested value is falsy. If we want to change the previous type to account for this fact, then we should give 1Or the type (Any, Any) → Any (where Any is the type of all values), which is a rather useless type since it essentially states that 1Or is a binary function. To give 1Or a more informative type, we need union and intersection types (which are already integrated in typed versions of JavaScript such as TypeScript [Microsoft] and Flow [Facebook]): we define the type Falsy as the following union type false ∨ "" ∨ 0 ∨ −0 ∨ 0n ∨ undefined ∨ null ∨ NaN, where each value denotes here the singleton type containing that value, and the type Truthy to be its complement, ¬Falsy, that is, the type of all values that are not of type Falsy. Then we can deduce for 1Or the following more precise type

((Truthy, Any) → Truthy) ∧ ((Falsy, Truthy) → Truthy) ∧ ((Falsy, Falsy) → Falsy) (1)

Authors’ addresses: Giuseppe Castagna, Mickaël Laurent, Institut de Recherche en Informatique Fondamentale (IRIF), Université Paris Cité, CNRS, 8 place Aurélie Nemours, 75013 Paris, France; Kim Nguyen, Laboratoire Méthodes Formelles (LMF), Université Paris-Saclay, CNRS, ENS Paris-Saclay, 91190, Gif-sur-Yvette, France.

This work is licensed under a Creative Commons Attribution 4.0 International License.
In this type, \( \land \) is a type combinator denoting intersection and meaning that the function has all the types given in the intersection: that is, in words, if the first argument of a function of this type is a Truthy, then the function returns a Truthy regardless of the second argument (first arrow type), while if the first argument is a Falsy, then the result is of the same type as the second argument’s type (second and third arrow type). Notice how the use of an intersection of arrow types corresponds to the typing of an “overloading” behavior (also known as, ad hoc polymorphism [Strachey 1967]), insofar as the result of an application depends on the type of the input.

In order to derive such a type, the type system must deduce that whenever the condition tested by the if holds, then \( x \) is of type Truthy and, therefore, (i) that all occurrences of \( x \) in the “then” branch (here just one) have type Truthy and (ii) that all the occurrences of the same variable \( x \) in the branch “else” (here none) have thus type Falsy. This kind of deduction is usually referred as type narrowing or occurrence typing since it requires to “narrow” the type of a variable \( x \) differently for its different occurrences. A type system such as the one for Typed Racket—defined in [Tobin-Hochstadt and Felleisen 2010] where the term occurrence typing was first introduced—is able to check that 1Or has the type in (1), meaning that the deduction requires the programmer to explicitly specify the type in the code. The system by Castagna et al. [2022b] makes a step further, since it can not only check that 1Or has type in (1), but it can also reconstruct for 1Or the intersection type ((Truthy, Any) \( \rightarrow \) Truthy) \& ((Falsy, Any) \( \rightarrow \) Any) which, although it is less precise a type than (1), it is inferred from the code of 1Or as is, without needing any type annotation. This latter work constitutes the state of the art of this kind of inference, since it is the only system that can reconstruct intersections of arrow types.

In this work we go a (big) step further, and show how to infer (i.e., reconstruct) intersections of polymorphic function types. In particular, the system we present here reconstructs for 1Or the following first order polymorphic type (where \( \alpha \) and \( \beta \) are type variables):

\[
\forall \alpha, \beta. ((\alpha \land \text{Truthy}, \text{Any}) \rightarrow \alpha \land \text{Truthy}) \land ((\text{Falsy}, \beta) \rightarrow \beta)
\]  

(2)

This type completely specifies the semantics of the function 1Or: it states that if the first argument is a Truthy, then the application of the function returns the first argument,\(^2\) otherwise it returns the second argument. This type is more precise than the one in (1), since it allows the system to deduce that, say, if the first argument of 1Or is an object, then the result will be an object of the same type (rather than just a truthy value). Not only does the system we present here infers such a precise type, but this kind of precision is compositional, yielding an accurate type also for the expressions in which the function is used. For instance, if we define the following function:

```javascript
function id (x) {
    return 1Or(x, x)
}
```

then, as we explain later on, our system infers that id has type \( \forall \alpha. \alpha \rightarrow \alpha \), viz., that id is indeed the polymorphic identity function.

This is clearly better than the current state of the art. Still, it does not seem too hard a feat to deduce that if we are testing whether \( x \) is a truthy value, then when the test succeeds we can assume that \( x \) is of type Truthy. To show the more advanced capabilities of our system let us have a look at how ECMA-Script specifies the semantics of JavaScript logical operators, as defined in

\[^1\]This type can be considered as an encoding of \( \forall (\alpha \leq \text{Truthy}). \forall (\beta). (((\alpha, \text{Any}) \rightarrow \alpha) \land ((\text{Falsy}, \beta) \rightarrow \beta) \) a type expressed in so-called bounded polymorphism: see Castagna [2023a, Section 2].

\[^2\]Strictly speaking, the type states that the function returns a result of the same type as the first argument, but by parametricity we can deduce that the result will be the first argument. Likewise for the second argument. A simple way to understand it, is by instantiating both type variables in (2) with the singleton type of the (result of the) argument.
the 2021 version of the specification [Ecma 2021, Section 13.13.1]. Since in JavaScript there are no union or intersection types, then the falsy and truthy values are defined via an (abstract) function ToBoolean which simply checks whether its argument is one of the 8 falsy values and returns false, otherwise it returns true (see its definition in row 1 of Table 1). In our system, ToBoolean has type (Truthy → true) ∧ (Falsy → false). All logical operators are then defined by ECMAScript in terms of this function: this has the advantage that any change to the specification of falsy (e.g., the addition of a new falsy value, like the addition of the built-in bigint type and its constant 0n in ES2020) requires only the modification of this function, and is automatically propagated to all operators. So the actual definition of 10r for ECMAScript is the following one:

```javascript
function 10r (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

If we feed this function to our system, then it infers for it the type in (2), that is, the same type it already deduced for the simpler version of 10r defined in lines 1-3. But here the deduction needed to perform type narrowing is more challenging, since the system must deduce from the type (Truthy → true) ∧ (Falsy → false) of ToBoolean that when the application in line 8 returns a truthy value, then the argument of ToBoolean is of type Truthy, and it is of type Falsy otherwise. More generally, we need a system which, when a test is performed on an arbitrarily complex application, can narrow the type of all the variables occurring in the application by exploiting the information provided by the overloaded behavior of the functions therein. Achieving such a degree of precision is a hard feat but, we argue, it is necessary if we want to reconstruct types for dynamic languages, that is, if we want to type their programs as they are, without requiring the addition of any type annotations. Indeed, the core operators of these languages (e.g., JavaScript’s “||”, “&&”, “typeof”,...) are characterized by an “overloaded” behavior, which is then passed over to the functions that use them. So for instance a simple use of JavaScript logical or “||” such as

in (x => x || 42) results in a function whose precise type, as reconstructed by our system, is (Falsy → 42) ∧ (Truthy ∧ α → Truthy ∧ α). JavaScript functions also routinely perform dynamic checks against constants (notably null and undefined), which our system also handles as part of its more general approach to type narrowing of arbitrary expressions.

1.1 Outline

Type System (Section 2). So, how can we achieve all this? Conceptually, it is quite simple: we just merge together three of the most expressive type systems studied in the literature, namely the Hindley-Milner (HM) polymorphic types [Hindley 1969; Milner 1978], intersection types [Coppo et al. 1981], and union types [Barbanera et al. 1995; MacQueen et al. 1986]. We achieve it simply by putting together in a controlled way the deduction rules characteristic of each of these systems (see Figure 2 in Section 2) and proving that the resulting system is sound (cf., Theorem 2.2).

More precisely, the type system we describe in Section 2 is pretty straightforward. Its core is a classic HM system with first order polymorphism: a program is a list of let-bindings that define polymorphic functions; these are typed by inferring a type for the expressions that define them, this type is then generalized, yielding a prenex polymorphic type for the function. As usual, the deduction of the type of each of these expressions is performed in a type environment that records the generic types for the previously-defined polymorphic functions, and the type system can instantiate these types differently for each use of the polymorphic functions in the expression. The novelty of our system is that when deducing the types of the expressions that define the polymorphic functions, the type system can use not only instantiations of polymorphic types (rule [Inst] in Figure 2), but also intersection and union types. More precisely, to type these
expressions the type system can decide to use the classic rules of intersection introduction (rule $\land$) and union elimination (rule $\lor$) given in Figure 2. For instance, the intersection introduction rule is used by the system to deduce that since the function $\text{lOr}$ (either versions) has both type $((\alpha \land \text{Truthy}, \text{Any}) \rightarrow \alpha \land \text{Truthy})$ and type $((\text{Falsy}, \beta) \rightarrow \beta)$, then it has their intersection, too; this intersection type is then generalized (when $\text{lOr}$ is defined at top-level) yielding the polymorphic type in (2). The union elimination rule is essentially used to fine-grainedly type branching expressions and tests involving applications of overloaded functions: for instance, to deduce that the function $\text{id}$ in lines 4–6 has type $\alpha \rightarrow \alpha$, the system can assume that $x$ has type $\alpha$ and separately infer the type of the body for $x$ : $((\alpha \land \text{Truthy})$ and for $x$ : $(\alpha \land \neg \text{Truthy})$; since the first deduction yields $(\alpha \land \text{Truthy})$ and the second yields $(\alpha \land \neg \text{Truthy})$, then the system deduces that under the hypothesis $x : \alpha$, the body has the union of these two types, that is $\alpha$. Furthermore, as observed by Castagna et al. [2022b], the combination of the union elimination with the rules of type-cases given in Figure 2 constitutes the essence of narrowing and occurrence typing.

The declarative type system given in Section 2 is all well and good, but how can we define an algorithm that infers whether a given expression can be typed in this system? Rules such as union elimination and intersection introduction are easy to understand, but they do not easily lend themselves to an implementation. In order to arrive to an effective implementation of the type system specified in Section 2 we proceed in two steps: (i) the definition of an algorithmic system and (ii) the definition of a reconstruction algorithm.

**Algorithmic System (Section 3).** The first step towards an effective implementation of our type system is taken in Section 3 where we define an algorithmic system that is sound and complete with respect to the system of Section 2. The system is algorithmic since it is composed only by syntax-directed and analytic rules\(^3\) and, as such, is immediately implementable. It is sound and complete since an expression is typable in it if and only if it is typable in the system of Section 2. To obtain this results the system is defined on pairs formed by an MSC-form (Maximal Sharing Canonical form) and an annotation tree. MSC-forms are A-normal forms [Sabry and Felleisen 1992] on steroids: they are lists of bindings associating variables to expressions in which every proper subexpression is a variable. Their characteristic is that they encode expressions and preserve typability in the sense that every expression is typable if and only if its unique MSC-form is typable. MCS-forms were introduced by Castagna et al. [2022b] to drastically reduce the range of possible applications of the union elimination rule; here we improve their definition to deal with our polymorphic setting and use them for exactly the same reason as in [Castagna et al. 2022b]. Annotation trees encode canonical derivations of the system of Section 2 for the MSC-form they are paired with. They are a generalization of type annotations inserted in the code. Instead of annotating directly an MSC-form with type-annotations we used a separate annotation tree because of the union elimination rule which types several times the same expression under different type environments; this would, thus, require different annotations for the same subexpressions, each annotation depending on the typing context: this naturally yields to tree-shaped annotations in which each branching corresponds either to the different deductions performed by a union elimination rule or to the different deductions performed by an intersection introduction rule. The soundness and completeness properties of the algorithmic systems are thus stated in terms of MSC-forms and annotation trees. They essentially state that an expression $e$ has type $t$ in the declarative system of Section 2 if and only if there exists a tree annotation for the (unique) MSC-form of $e$ that is typable in the algorithmic system with (a subtype of) $t$: see Theorem 3.4.

\(^3\)A rule is analytic (as opposed to synthetic) when the input (i.e., $\Gamma$ and $e$) of the judgment at the conclusion is sufficient to determine the inputs of the judgments at the premises (cf. [Martin-Löf 1994; Types 2019]).
Reconstruction Algorithm (Section 4). The second of the two steps to achieve an effective implementation for the type system of Section 2 is to define a reconstruction algorithm for the previous algorithmic system, which we do in Section 4. The statements of the soundness and completeness properties of the algorithmic system clearly suggest what this algorithm is expected to do: given an expression that defines a polymorphic function, the algorithm must transform it into its unique MSC-form and then try to reconstruct an annotation tree for it so that the pair MSC-form and annotation tree is typable in the algorithmic system of Section 3.

The reconstruction is performed by a system of deduction rules that incrementally refines an annotation tree (initially composed of a single node $\text{infer}$) while exploring the list of bindings of the MSC-form of the expression to type. It mixes two independent mechanisms: one that infers the domain(s) of $\lambda$-terms, and the other that performs type narrowing when a typecase is encountered.

The first mechanism is inspired by the algorithm $W$ by Damas and Milner [1982]: whenever the application of a destructor (e.g., a function application) is encountered, an algorithm finds a substitution (if any) that makes this application well-typed. In the context of a HM type system, the algorithm at issue needs to solve a unification problem (i.e., whether for two given types $s$ and $t$ there exists a substitution $\sigma$ such as $s\sigma = t\sigma$) which, if solvable, has a principal solution given by a single substitution [Robinson 1965]. In our system, which is based on subtyping, the algorithm at issue needs to solve a tallying problem (i.e., whether for two given types $s$ and $t$ there exists a substitution $\sigma$ such as $s\sigma \leq t\sigma$) which, if solvable, has a principal solution given by a finite set of substitutions [Castagna et al. 2015]. When multiple substitutions are found, they are all considered and explored in different branches by adding an intersection branching node in the current annotation tree.

The second mechanism gets inspiration from Castagna et al. [2022b] and refines decompositions made by the union-elimination rule in order to narrow the types of variables in the branches of a typecase expression. When the system encounters a typecase that tests whether some expression $e$ has type $t$, then the type $s$ of the variable bound to $e$ (recall that an MSC-form is a list of bindings) is split into $s \land t$ and $s \land \neg t$, and these splits are in turn propagated recursively in order to generate new splits for the types of the variables associated with the subexpressions composing $e$. For instance, when the algorithm encounters the test “if $(\text{ToBoolean}(x))$...” at line 8, it splits the type of (the variable bound to) ToBoolean$(x)$ in two, by intersecting it with true and $\neg$true, and this split in turn generates a new split Truthy and Falsy for the type of the variable $x$.

The reconstruction algorithm we present in Section 4 is sound: if it returns an annotation tree for an MSC-form, then the pair is typable in the algorithmic system, whose soundness implies that the expression at the origin of the MSC-form is typable in the system of Section 2. At this point, however, it should be pretty obvious that such a reconstruction algorithm cannot be complete. Our system merges three well know systems: first-order parametric polymorphism, intersection types, and union elimination. Now, even if parametric polymorphism is decidable, in our system we can encode (and type, via intersection types) polymorphic fixed-point combinators, yielding a system with polymorphic recursion whose inference has been long known to be undecidable [Henglein 1993]. More generally, a well-know property of intersection type systems is that they are undecidable, since typability is equivalent to deciding termination [Pottinger 1980]. Even worse, our system includes union elimination, which is one of the most problematic rules from an algorithmic viewpoint, not only because it is neither syntax directed nor analytic, but also because determining an inversion (a.k.a., generation) lemma for this rule is considered by experts the most important open problem in the research on union and intersection types [Dezani 2020], and an inversion lemma is somehow the first step to define a type-inference algorithm, since it tells us when and how to apply the rule.

Despite being incomplete, our reconstruction algorithm is powerful enough to handle both complicated typing use-cases and common programming patterns of dynamic languages. For instance,
for the $Z$ fixed-point combinator for strict languages $Z = \lambda f. (\lambda x. f(\lambda v.xxv)) (\lambda x. f(\lambda v.xxv))$ our algorithm reconstructs the type $\forall \alpha, \beta, \gamma. ((\alpha \rightarrow \beta) \rightarrow ((\alpha \rightarrow \beta) \land \gamma)) \rightarrow ((\alpha \rightarrow \beta) \land \gamma)$ (i.e., in bounded polymorphic notation $\forall (\alpha)(\beta)(\gamma \leq \alpha \rightarrow \beta). ((\alpha \rightarrow \beta) \rightarrow \gamma) \rightarrow \gamma$, cf. Footnote 1). The combinator can then be used as is, to define and infer the type of classic polymorphic functions such as map, fold, concat, reverse, etc., often yielding types more precise than in HM: for instance if we use $[\alpha*]$ to denote the type of the lists whose elements have type $\alpha$, then the type inferred for (a curried version of) fold_r is $\forall \alpha, \beta, \gamma. ((\alpha \rightarrow \beta \rightarrow \beta \rightarrow [\alpha*] \rightarrow \beta) \land (\text{Any} \rightarrow \gamma \rightarrow []) \rightarrow \gamma)$ where the second type in the intersection states that if the third argument is an empty list, then the result will be the second argument, whatever the type of the first argument is. Finally, we designed our algorithm so that it can take into account explicit type annotations to help it in the inference process. As an example, our algorithm can check that the classic filter function has type $\forall \alpha, \beta, \gamma. ((\alpha \land \beta \rightarrow \text{Bool}) \land (\alpha \land \neg \beta \rightarrow \text{False})) \rightarrow [\alpha*] \rightarrow [(\alpha \land \beta)*]$, stating that if we pass to filter a predicate that returns false for the elements of $\alpha$ that are not in $\beta$, then filtering a list of $\alpha$’s will return only elements also in $\beta$.

Sections 2, 3, and 4 outlined above constitute the core of our contribution. Section 5 presents our implementation. In Section 6 we discuss related work and Section 7 concludes our presentation. For space reasons we omitted in the main text some rules of the algorithmic and reconstruction systems, as well as all proofs: they are all given in the appendix of this work.

1.2 Discussion and Contributions

Intersections vs. Hindley-Milner. It is a truth universally acknowledged that intersection type systems are more powerful than HM systems: for that, one does not even need full intersections, since Rank 2 intersections suffice. Rank 2 intersection types are types that may contain intersections only to the left of a single arrow and the system of Rank 2 intersection types is able to type all ML programs (i.e., all program typable by HM), has principal typings, decidable type inference, and the complexity of type inference is of the same order as in ML [Leivant 1983].

However, intersection type systems are not compositional, and this hinders their use in a modular setting. A program that uses the polymorphic identity function to apply it to, say, an integer and a Boolean, type checks since we can infer that the polymorphic identity function has type (Int→Int)∧(Bool→Bool). But if we want to export this polymorphic identity function to use it in other unforeseen contexts, then we need for it a type that covers all its admissible usages, without the need of retyping the function every time it is applied to an argument of a new type. In other words, in a modular usage, parametric polymorphism has an edge over intersection/ad-hoc polymorphism despite being less powerful, since a type such as $\forall \alpha. \alpha \rightarrow \alpha$ synthesizes the infinitely many combinations of intersection types that can be deduced for the identity function; however in a local setting, everything that does not need to be exported can be finer-grainedly typed by intersection types. This division of roles and responsibilities is at the core of our approach. As we show in the next section, programs are lists of bindings from variables to expressions. These expressions are typed in a type environment (generated by the preceding bindings) which binds variables to polymorphic types. These expressions are typed by using instantiation, intersection introduction, and union elimination, but not generalization. Generalization is performed only at top level, that is at the level of programs and reserved to expressions to be used in other contexts.

Parametricity vs. type cases. A parametric polymorphic function (a.k.a., a generic function) is a function that behaves uniformly for all possible types of its arguments, that is, whose behavior does not depend on the type of its arguments. A common way to characterize a generic function is that it is a function that cannot inspect the parametric parts of its input, that is, those parts that are typed by a type variable: these parts can only be either returned as they are, or discarded, or passed
to another generic function. Our approach allows us to refine this characterization by shifting the attention from inputs as a whole to some particular values among all the possible inputs. This can be seen by comparing the following two function definitions:

\[
\lambda x. (x \in \text{Int}) \ ? \ x \ \ \ \ \lambda x. (x \in \text{Int}) \ ? \ x + 1 : x
\]

Both functions test whether their input is an integer. The function on the right-hand side returns the successor of the argument when this is true and the argument itself otherwise; the one on the left-hand side returns its argument in both cases, that is, it is the identity function.

Our system deduces for the function on the left the type \(\alpha \rightarrow \alpha\).\(^4\) For the function on the right it returns the type \((\text{Int} \rightarrow \text{Int}) \land (\alpha \setminus \text{Int} \rightarrow \alpha \setminus \text{Int})\), where \(s \setminus t\) denotes the set-theoretic difference of the two types, that is, \(s \land \neg t\). These two types explain how we can refine the characterization of parametricity. A generic function can inspect the parametric part of its input, but it preserves its parametricity (i.e., the type variable is in the result type) only for the values for which the outcome of this inspection is not used. The domain of both functions is Any: they can be applied to any argument. But the first function is parametric for all possible inputs, since it does not use the result of the inspection (it has type \(\forall \alpha. \alpha \rightarrow \alpha\)), while the second function is parametric only for the values of its domain that are not in Int, since it uses the result of the inspection for the other ones (by subsumption it has type \(\forall \alpha. \alpha \rightarrow \text{Int} \lor (\alpha \setminus \text{Int})\): the parametricity is maintained only for the arguments not in Int).

Contributions. The general contribution of our work is twofold. First, it proposes a way to mix parametric and intersection/ad-hoc polymorphism which, in hindsight, is natural: parametric polymorphism for everything defined at top-level and that can thus be used in other contexts (modularity); intersection polymorphism for everything that remains local (for which we can thus use more precise non-modular typing). Second, it proposes an effective way to implement this type discipline by defining a reconstruction algorithm; with respect to that, a fundamental role is played by the analysis of the (type-)tests performed by the expressions, since they drive the way in which types are split: externally, to split the domain of functions yielding intersection of arrows (intersection introduction); internally, to split the type of tested expressions, yielding a precise typing of branching (union elimination). In doing so, it provides the first system that reconstructs types that combine parametric and ad hoc polymorphism.

The technical contributions of the work can be summarized as follows:

1. We define a type system that combines parametric polymorphism with union and intersection types for a functional calculus with type-cases and prove its soundness.
2. We define an algorithmic system that we prove sound and complete with respect to the previous system.
3. We define an algorithm to reconstruct the type annotations of the previous algorithmic system and prove it sound and terminating.

The reconstruction algorithm is fully implemented. A prototype which also implements pattern matching and optional type annotations (presented only in Appendix A) is available on-line at https://poly-dl.github.io/poly-dl/.

2 SOURCE LANGUAGE AND TYPE SYSTEM

2.1 Syntax and semantics

Our core language is fully defined in Figure 1. Expressions are an untyped \(\lambda\)-calculus with constants \(c\), pairs \((e, e)\), pair projections \(\pi_i e\), and type-cases. A type-case \((e_0 \in \tau) \ ? \ e_1 : e_2\) is a dynamic type test that first evaluates \(e_0\) and, then, if \(e_0\) reduces to a value \(v\), evaluates \(e_1\) if \(v\) has type \(\tau\) or \(e_2\)

\(^4\)Precisely, it deduces for it the type \((\alpha \land \text{Int} \rightarrow \alpha \land \text{Int}) \land (\beta \setminus \text{Int} \rightarrow \beta \setminus \text{Int})\). Instantiating \(\beta\) to \(\alpha\) yields a subtype of \(\alpha \rightarrow \alpha\).
Syntax

\begin{align*}
\text{Test Type} & \quad \tau ::= b \mid 0 \to 1 \mid \tau \times \tau \mid \tau \lor \tau \mid \neg \tau \mid 0 \\
\text{Expression} & \quad e ::= c \mid x \mid \lambda x.e \mid ee \\
& \quad \mid (e, e) \mid \pi_1 e \mid (e \in \tau) \ ? e : e \\
\text{Value} & \quad v ::= c \mid \lambda x.e \mid (v, v) \\
\text{Program} & \quad p ::= \text{let } x = e ; p \mid e \\
\end{align*}

\begin{align*}
\text{Dynamic type test} \\
v \in \tau & \iff \text{typeof}(v) \leq \tau, \text{where} \\
\begin{cases}
\text{typeof}(c) &= b_c \\
\text{typeof}((v_1, v_2)) &= \text{typeof}(v_1) \times \text{typeof}(v_2) \\
\text{typeof}(\lambda x.e) &= \emptyset \to 1
\end{cases}
\end{align*}

Evaluation Contexts

\begin{align*}
E ::= \ [] \mid vE \mid Ee \mid (v, E) \mid (E, e) \mid \pi_1 E \mid (E \in \tau) \ ? e : e \\
P ::= \ [] \mid \text{let } x = \ [] ; p
\end{align*}

\begin{align*}
e \to e' & \quad E[e] \to E[e'] \\
E[e] \to P[e] & \quad P[e] \to P[e']
\end{align*}

Fig. 1. Syntax and semantics of the source language

otherwise. Type-cases cannot test arbitrary types but just ground types (i.e., types without type variables occurring in them) of the form \(\tau\) where the only arrow type that can occur in them is \(0 \to 1\), the type of all functions. This means that type-cases can distinguish functions from other values but they cannot distinguish, say, functions that have type \(\text{Int} \to \text{Int}\) from those that do not.

Programs are sequences of top-level definitions, ending with an expression that can be seen as the main entry. This notion of program is useful to capture the modularity of our type system. Indeed, top-level definitions are typed sequentially: the type we obtain for a top-level definition is considered definitive and will not be challenged by a later definition.

The reduction semantics for expressions is the one of call-by-value pure \(\lambda\)-calculus with products and with a type-case expression, together with the context rules that implement a leftmost outermost reduction strategy. We use the standard substitution operation \(e\{e'/x\}\) that denotes the capture avoiding substitution of \(e'\) for \(x\) in \(e\), whose definition we recall in Appendix B. The relation \(v \in \tau\) determines whether a \(\text{value}\) is of a given type or not and holds true if and only if \(\text{typeof}(v) \leq \tau\), where \(\leq\) is the subtyping relation defined by Castagna and Xu [2011] (we recall its definition in Appendix C). Note that \(\text{typeof}(v)\) maps every \(\lambda\)-abstraction to \(0 \to 1\) and, thus, dynamic type tests do not depend on static type inference. This approximation is allowed by the restriction on arrow types in typecases. Finally, the reduction semantics for programs sequentially reduces top-level definitions, together with a context rule that allows reducing the expression of the first definition.

2.2 Types

Types are those by Castagna and Xu [2011] who add type variables to the semantic subtyping framework of Frisch et al. [2002, 2008].

**Definition 2.1 (Types).** The set of types \(\text{Types}\) is formed by the terms \(t\) coinductively produced by the grammar:

\[
\text{Types} \quad t, s ::= b \mid \alpha \mid \alpha \rightarrow t \mid t \times t \mid t \lor t \mid t \land t \mid \neg t \mid 0
\]

and that satisfy the following conditions: (i) every term has a finite number of different sub-terms (regularity) and (ii) every infinite branch of a term contains an infinite number of occurrences of the arrow or product type constructors (contractivity).

We use the abbreviations \(t_1 \wedge t_2 \overset{\text{def}}{=} \neg(\neg t_1 \lor \neg t_2)\), \(t_1 \times t_2 \overset{\text{def}}{=} t_1 \wedge (\neg t_2)\), and \(\alpha \overset{\text{def}}{=} \neg 0\). Basic types (e.g., \(\text{Int}, \text{Bool}\)) are ranged over by \(b\), \(0\) and \(1\) respectively denote the empty (that types no value) and
top (that types all values) types. Coinduction accounts for recursive types and the condition on
infinite branches bars out ill-formed types such as \( t = t \lor t \) (which does not carry any information
about the set denoted by the type) or \( t = \neg t \) (which cannot represent any set).

For what concerns type variables, we choose not to use type-schemes but rather distinguish
two kinds of type variables. *Polymorphic type variables* ranged over by \( \alpha \), are type variables
that have been generalized and can therefore be instantiated. In a more traditional presentation, such
variables are bound by the \( \forall \) of a type-scheme ; the set of polymorphic variables is \( \forall \alpha \).
*Monomorphic type variables*, ranged over by \( \xi \) (with bold font), are variables that are not generalized and
therefore cannot be instantiated; the set of monomorphic variables is \( \forall \xi \).

For what concerns type variables, we choose not to use type-schemes but rather distinguish
two kinds of type variables.

Monomorphic types \( u, v ::= b | \alpha | u \rightarrow u | u \times u | u \lor u | \neg u | 0 \)

Our choice of using two disjoint sets for polymorphic and monomorphic type variables, instead
of the classical approach of using type schemes \( \forall \alpha_1...\alpha_n.t \), is justified by two reasons. First, type
schemes are expected to be equivalent modulo \( \alpha \)-renaming. In our case however, we do not want
polymorphic type variables to be freely renamed because of the use, in the algorithmic type system
of Section 3, of external annotations containing explicit substitutions over some polymorphic type
variables of the context. Secondly, introducing type schemes would require redefining many of
the usual set-theoretic type-related definitions, such as the subtyping relation \( \leq \), and the type
operators for application \( \circ \) and projections \( \pi_i \). Instead, we obtain a more streamlined theory by
making subtyping and these operators ignore whether a variable is polymorphic or monomorphic
in the current context and by explicitly performing instantiations in the type system when required.

The subtyping relation for these types, noted \( \leq \), is the one defined by Castagna and Xu [2011],
to which the reader may refer for the formal definition (cf. Appendix C). For this presentation, it
suffices to consider that ground types (i.e., types with no variable) are interpreted as sets of values
that have that type, and that subtyping is set containment (i.e., a type \( s \) is a subtype of a type \( t \) if
and only if \( t \) contains all the values of type \( s \)). In particular, \( s \rightarrow t \) contains all \( \lambda \)-abstractions that when
applied to a value of type \( s \), if their computation terminates, then they return a result of type \( t \) (e.g.,
\( 0 \rightarrow 1 \) is the set of all functions and \( 1 \rightarrow 0 \) is the set of functions that diverge on every argument).
Type connectives (i.e., union, intersection, negation) are interpreted as the corresponding set-
theoretic operators. For what concerns non-ground types (i.e. types with variables occurring in
them) all the reader needs to know for this work is that the subtyping relation of Castagna and Xu
[2011] is preserved by type-substitutions. Namely, if \( s \leq t \), then \( s\sigma \leq t\sigma \) for every type-substitution
\( \sigma \). We use \( \simeq \) to denote the symmetric closure of \( \leq \), thus \( s \simeq t \) (read, \( s \) is equivalent to \( t \)) means that
\( s \) and \( t \) denote the same set of values and, as such, they are semantically the same type.

2.3 Type System

Our type system is given in full in Figure 2. The typing rules for expressions are, to some extent,
the usual ones. Constants and variables are typed by the corresponding axioms \([\text{Const}]\) and
\([\text{Ax}]\). The arrow and product constructor have introduction and elimination rules. Notably, in
the case of rule \([\rightarrow I]\) the type of the argument is monomorphic. The rules for intersection \((\land)\)
and subtyping \((\leq)\) are the classical ones, and so is the rule for instantiation \((\text{Inst})\) where \( \sigma \) denotes
a substitution from polymorphic variables to types. The type-case construction is handled by three
rules: \([\emptyset] ; [e_1] ; [e_2] \). Rule \([\emptyset] \) handles the case where the tested expression is known to have the
empty type. The other two are symmetric and handle the case when the tested expression is known
to have either the type \( \tau \) or its negation, in which case the corresponding branch is typed. These
rules work together with Rule \([\lor] \), which we now describe in detail.
At first sight, the formulation of rule $[\forall]$ seems odd, since the $\lor$ connector does not appear in it. To understand it, consider the classic union elimination rule by MacQueen et al. [1986]:

$$\Gamma \vdash e : t \quad \Gamma \vdash e : t' \quad \Gamma \vdash e : t \lor t'$$

Rule $[\forall E]$ types an expression that contains occurrences of an expression $e'$ that has a union type $s_1 \lor s_2$; the rule substitutes in this expression some occurrences of $e'$ by the variable $x$ yielding an expression $e$, and then types $e$ first under the hypothesis that $x$ has type $s_1$ and then under the hypothesis that $x$ has type $s_2$. If both succeed, then the common type is returned for the expression at issue. As shown by Castagna et al. [2022b], this rule, together with the rules for type-cases, allows the system to perform occurrence typing. For instance, consider the expression $(f y \in \text{Int} ) \lor 1 : \text{false}$, in the context where $f$ has type $\text{Any} \to \text{Any}$ and $y$ is of type $\text{Int}$. This expression can be typed thanks to the rule $[\forall E]$, by considering the sub-expression $f y$. This sub-expression has type $\text{Any}$ which can be seen as the union type $\text{Any} \simeq \text{Int} \lor \neg \text{Int}$. We can then replace $x$ for $f y$ and type, using $[\epsilon_1]$, the expression $(x \in \text{Int}) \lor 1 : \text{false}$, with $x : \text{Int}$. This yields a type $\text{Int}$ (rule $[\epsilon_1]$ ignores the second branch) and by subtyping, the expression has type $\text{Int} \lor \text{false}$. Likewise for the choice $x : \neg \text{Int}$, using rule $[\epsilon_2]$ the second branch has type $\text{false}$ and therefore $\text{Int} \lor \text{false}$ (again via subtyping). The whole expression has thus the desired type $\text{Int} \lor \text{false}$.

A key element is that rule $[\forall E]$ guessed how to split the type $\text{Any}$ of $f y$ into $\text{Int} \lor \neg \text{Int}$. In a non-polymorphic setting, this is perfectly fine. But in a type-system featuring polymorphism, particular care must be taken when introducing (fresh) type variables. As it is stated, rule $[\forall E]$ could choose to split, say, $\alpha$ into a union $\alpha \lor \neg \alpha$, with $\alpha$ a polymorphic type variable. If so, then the rule becomes unsound. As a matter of facts, the premises of the $[\forall E]$ behave as in rule $[\to I]$, in that they introduce in the typing environment a fresh type whose variables must not be instantiated. In our example, however, in one premise, the rule introduces $x : \alpha$ in the typing environment which can, for instance, be instantiated by the $[\text{Inst}]$ rule. In the second premise, it introduces $x : \neg \alpha$ which can also be instantiated in a completely different way. In other words, the correlation

![Fig. 2. Declarative Type System](image-url)
between the two occurrences of the same variable $\alpha$ is lost, which amounts to commuting the (implicit) universal quantification with the $\vee$ type connective, yielding a non-prenex polymorphic type $(\forall \alpha. \alpha) \vee (\forall \alpha. \neg \alpha)$. To avoid this unsound situation, we need to ensure that when a type is split between two components of a union, no polymorphic variable is introduced. This is achieved by rule $[\lor]$ which requires the type $s$ of $e'$ to be split as $s \equiv (s \land u) \vee (s \land \neg u)$ (here is our hidden union).

The top-level definitions of a program are typed sequentially by two specific rules:

$$
\begin{align*}
\text{TopLevel-Expr} & : \quad \Gamma \vdash e : t \phi \Gamma \\
\text{TopLevel-Let} & : \quad \Gamma \vdash P \ e : t \phi \quad \Gamma, x : t \vdash P \ p : t' \phi
\end{align*}
$$

where $\phi$ denotes a generalization, that is a substitution transforming monomorphic variables into polymorphic ones and where $\phi \# \Gamma \define \text{dom}(\phi) \cap \text{vars}(\Gamma) = \emptyset$.

After typing an expression used for a definition, its type is generalized (Rule $[\text{TopLevel-Expr}]$) before being added in the environment (Rule $[\text{TopLevel-Let}]$). Note that this is the only place where generalization takes place: no rule in the type system for expressions (Figure 2) allows the generalization of a type variable. As explained at the beginning of Section 1.2, this is not a limitation, since intersection types are more powerful than HM polymorphism, and top-level generalization is of practical importance since it is necessary to the modularity of type-checking. Nevertheless, the core of our inference system is given only by the rules in Figure 2 for expressions: the above “TopLevel” rules are only useful to inhabit variables of the typing environments used in the rules for expressions, and this makes it possible to close the expressions being typed. For instance, if a typing derivation for an expression $e$ is deduced, say, under the hypothesis $x : \alpha \rightarrow \alpha$ (with $\alpha$ polymorphic), then it is possible to obtain a closed program by inhabiting $x$ by a definition such as $\text{let } x = \lambda y. y \ ; \ e$. This is the reason the rest of this works mainly focuses on typing expressions.

The type system is sound (all proofs for this work are in Appendix I):

**Theorem 2.2 (Soundess).** If $\emptyset \vdash P \ p : t$, then either $p$ diverges or $p \sim P \ v$ with $\emptyset \vdash P \ v : t$

3 ALGORITHMIC SYSTEM

As discussed in the introduction, the declarative type system is not syntax directed and some rules are not analytic. In order to make it algorithmic, we first introduce in Section 3.1 a canonical form for expressions that adds syntactic constructions (bindings) to indicate when to apply the union elimination rule and on which sub-expression. Then, in Section 3.2, we define a fully algorithmic type system that takes a canonical form together with an annotation and produces a type.

3.1 MSC Forms

3.1.1 Canonical Forms. The $[\lor]$ rule is not syntax directed since it can be applied on any expression and can split the type of any of its subexpressions. If we want an algorithmic type system, we need a syntactic way to determine when to apply this rule, and which subexpression to split. In order to achieve this, we represent our terms with a syntax called Maximal Sharing Canonical Form (MSC Form) introduced by Castagna et al. [2022b]. Let us start with defining the canonical forms, which are expressions produced by the following grammar:

$$
\begin{align*}
\text{Atomic expressions} & \quad a \ ::= \ c \ | \ x \ | \ \lambda x. \ k \ | \ (x, x) \ | \ xx \ | \ \pi x \ | \ (x \in \tau) \ ? \ x : x \\
\text{Canonical Forms} & \quad k \ ::= \ x \ | \ \text{bind } x = a \ \text{in } k
\end{align*}
$$

Canonical forms, ranged over by $k$, are binding variables (noted $x$, $y$, or $z$) possibly preceded by a list of definitions (from binding variables to atoms). Atoms are either a variable from a $\lambda$-abstraction (noted $x$, $y$, or $z$), or a constant, or a $\lambda$-abstraction whose body is a canonical form, or any other
expression in which all proper sub-expressions are binding variables. An expression in canonical form without any free binding variable can be transformed into an expression of the source language using the unwinding operator \([\_]\) that basically inlines bindings: \([\text{bind } x = a \in \kappa] = [\kappa]([a]/x)\) (see Appendix E.1 for the full definition). The inverse direction, that is, producing from a source language expression a canonical form that unwinds to it, is straightforward (see Appendix E.2). However for each expression of the source language that are several canonical forms that unwind to it. For our algorithmic type system we need to associate each source language expression to a unique canonical form, as we define next.

3.1.2 Maximal Sharing Canonical Forms. We define a congruence on canonical forms and atoms:

**Definition 3.1 (Canonical equivalence).** We denote by \(\equiv\) the smallest congruence on canonical forms and atoms that is closed by \(\alpha\)-conversion and such that:

\[
\text{bind } x_1 = a_1 \in \text{bind } x_2 = a_2 \in \kappa \equiv \text{bind } x_2 = a_2 \in \text{bind } x_1 = a_1 \in \kappa \quad x_1 \notin \text{fv}(a_2), x_2 \notin \text{fv}(a_1)
\]

To infer types for the source language, we single out canonical forms satisfying three properties:

**Definition 3.2 (MSC forms).** A maximal sharing canonical form (abbreviated as MSC-form) is (any canonical form \(\alpha\)-equivalent to) a canonical form \(\kappa\) such that:

1. if \(\text{bind } x_1 = a_1 \in \kappa_1 \) and \(\text{bind } x_2 = a_2 \in \kappa_2\) are distinct sub-expressions of \(\kappa\), then \(a_1 \neq \kappa a_2\)
2. if \(\lambda x.\kappa_1\) is a sub-expression of \(\kappa\) and \(\text{bind } y = a \in \kappa_2\) a sub-expression of \(\kappa_1\), then \(\text{fv}(a) \not\subseteq \text{fv}(\lambda x.\kappa_1)\)
3. if \(\text{bind } x = a \in \kappa'\) is a sub-expression of \(\kappa\), then \(x \in \text{fv}(\kappa')\).

MSC-forms are defined modulo \(\alpha\)-conversion.\(^5\) The first property states that distinct variables denote different definitions. The second property requires that bindings must extrude \(\lambda\) abstractions whenever possible. The third condition states that there is no useless bind (the bound variable must occur in the body of the bind).

The key property of MSC-forms is that given an expression \(e\) of the source language, all its MSC-forms (i.e., all MSC-form whose unwinding is \(e\)) are equivalent:

**Proposition 3.3.** If \(\kappa_1\) and \(\kappa_2\) are two MSC-forms and \([\kappa_1] \equiv_a [\kappa_2]\), then \(\kappa_1 \equiv [\kappa_2]\).

We denote the unique MSC-form whose unwinding is \(e\) by MSC\((e)\). It is easy to transform a canonical form into a MSC-form that has the same unwinding. The reader can refer to Appendix E for a set of rewriting rules implementing this operation.

3.2 Algorithmic Typing Rules

MSC-forms tell us when to apply the \([\vee]\) rule: a term \(\text{bind } x = a \in \kappa\) means (roughly) that it must be typed by applying the union rule to the expression \(\kappa\{a/x\}\). Putting an expression into its MSC-form to type it, thus corresponds to applying the \([\vee]\) rule on every occurrence of every subexpression of the original expression. This is a step toward a syntax-directed type system. However, there are still two issues to solve before obtaining an algorithmic type system: (i) rules \([\wedge], [\text{Inst}]\) and \([\leq]\) are still not syntax-directed, and (ii) rules \([\vee], [\text{Inst}], [\rightarrow I]\), and \([\leq]\) are not analytic, meaning that some of their premises cannot be deduced just by looking at the conclusion: the \([\vee]\) rule requires guessing a type decomposition (i.e., the monomorphic type \(u\) in the premises), the \([\text{Inst}]\) rule requires guessing a substitution, the \([\rightarrow I]\) rule requires guessing the domain \(u\) of the function, and the \([\leq]\) rule requires guessing the type \(t'\) to subsume to.

\(^5\)For instance, both \(\lambda x.\text{bind } x = x \in \text{bind } z = xy \in \text{bind } z' = zy \in z'\) and \(\lambda x.\text{bind } x = x \in \text{bind } z = xy \in z\) are two distinct atoms that can occur in the same MSC-form, even though the atom \(xy\) appears in both: an \(\alpha\)-renaming of \(x\) makes the first MSC-property hold.
The issue of \([\text{Inst}]\) and \([\leq]\) not being syntax directed can be solved by embedding them in some structural rules (in particular, in the rules for destructors). Moreover, as we will see later, the rules in which we embed \([\leq]\) can be made analytic by using some type operators. As for rule \([\land]\), making it syntax-directed, is trickier. Indeed, the usual approach of merging rules \([-\rightarrow]\) and \([\land]\) does not work here, since terms in MSC-forms may hoist a bind definition outside the function where they are used, causing rule \([\land]\) to be needed on a term that is not, syntactically, a \(\lambda\)-abstraction. Lastly, there is no easy way to guess the substitutions used by \([\text{Inst}]\) rules, or the domain used in \([-\rightarrow]\) rules, or the decompositions performed by \([\lor]\) rules. To tackle these issues, our algorithmic type system will not only take a canonical form as input, but also an annotation that will \((i)\) indicate when to apply an intersection, and \((ii)\) indicate which type decomposition (for \([\lor]\) rules) and which type substitutions (for \([\text{Inst}]\) rules) to use. Formally, our algorithmic system uses judgements of the form \(\Gamma \vdash [k \mid \kappa] : t\) for a canonical form \(\kappa\), and \(\Gamma \vdash [a \mid \alpha] : t\) for an atom \(a\) where \(k\) and \(\alpha\) are respectively form annotations and atom annotations defined as follows:

**Atom annots**
\(a := \emptyset \mid \lambda(u, k) \mid (\rho, \rho) \mid \emptyset(\Sigma, \Sigma) \mid \pi(\Sigma) \mid 0(\Sigma) \mid e_1(\Sigma) \mid e_2(\Sigma) \mid \wedge(\{\omega, \ldots, \alpha\})\)

**Form annots**
\(k := \rho \mid \text{keep}(\alpha, \{(u, k), \ldots, (u, k)\}) \mid \text{skip} k \mid \wedge(\{k, \ldots, k\})\)

where \(\rho\) ranges over renamings of polymorphic variables, that is, injective substitutions from \(V_p\) to \(V_p\), and \(\Sigma\) ranges over instantiations, that is, sets of substitutions from \(V_p\) to \(\text{Types}\). We chose to keep annotations separate from the terms, instead of embedding them in the canonical forms, since in the latter case it would be more complicated to capture the tree structure of the derivations.

The algorithmic system is defined by the rules given in Appendix G. Below we comment the most interesting rules (we just omit the rules for constants, variables and two rules for type-cases). Essentially, there is one typing rule for each annotation, the only exception being the \(\emptyset\) annotation that is used both in the rule to type constants and in the two rules for variables.

\[
\frac{}{\Gamma, x : u \vdash_{\text{Alg}} [k \mid \kappa] : t} \quad \frac{}{\Gamma \vdash_{\text{Alg}} [\lambda x. \kappa \mid \lambda(u, k)] : u \rightarrow t}
\]

To type the atom \(\lambda x. \kappa\), the annotation \(\lambda(u, k)\) provides the domain \(u\) of the function, and the annotation \(k\) for its body.

\[
\frac{}{\Gamma \vdash_{\text{Alg}} [x_1 x_2 \mid \emptyset(\Sigma_1, \Sigma_2)] : t_1 \circ t_2 \quad t_1 \leq 0 \rightarrow 1, \ t_2 \leq \text{dom}(t_1)} \quad \frac{}{\Gamma \vdash_{\text{Alg}} [\pi_1 x \mid \pi(\Sigma)] : \pi_1(t) \leq (1 \times 1) \quad \Gamma \vdash_{\text{Alg}} [\pi_2 x \mid \pi(\Sigma)] : \pi_2(t) \leq (1 \times 1)}
\]

To type an application one must apply an instantiation and a subsumption to both the type of the function and the type of the argument. Instantiations (i.e., \(\Sigma_1\) and \(\Sigma_2\)) are sets of type substitutions; their application to a type \(t\) is defined as \(t\Sigma \overset{\text{def}}{=} \bigwedge_{\sigma \in \Sigma} t\sigma\). Since they cannot be directly guessed, they are given by the annotation. Subsumption instead is embedded in two type operators. A first operator, \(\text{dom}(t)\), computes the domain of the arrow and is used to check that the application is well-typed. A second type operator, \(\circ\), computes the type of the result of the application. These type operators are defined as follows: \(\text{dom}(t) \overset{\text{def}}{=} \max\{u \mid t \leq u \rightarrow 1\}\) and \(t \circ s \overset{\text{def}}{=} \min\{u \mid t \leq s \rightarrow u\}\).

\[
\frac{}{\Gamma \vdash_{\text{Alg}} [\pi_1 x \mid \pi(\Sigma)] : t = \Gamma(x)\Sigma \quad \Gamma \vdash_{\text{Alg}} [\pi_2 x \mid \pi(\Sigma)] : \pi_2(t) \leq (1 \times 1)}
\]

The rules for projections \([\times_1]\) and \([\times_2]\) follow the same idea as the rule for application \([-\rightarrow]\), with the use of two type operators \(\pi_1(t) \overset{\text{def}}{=} \min\{u \mid t \leq u\times1\}\) and \(\pi_2(t) \overset{\text{def}}{=} \min\{u \mid t \leq 1 \times u\}\). All these type operators can be effectively computed (cf. Appendix G).

\[
\frac{}{\Gamma \vdash_{\text{Alg}} [(x_1, x_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2 \quad t_1 = \Gamma(x_1)\rho_1, \ t_2 = \Gamma(x_2)\rho_2}
\]
To type a pair \((x_1, x_2)\) it is not necessary to instantiate \(\Gamma(x_1)\) or \(\Gamma(x_2)\). However, to avoid unwanted correlations, it is necessary to rename the polymorphic type variables of its components. For instance, when typing the pair \((x, x)\) with \(x : \alpha \rightarrow \alpha\), it is better to type it with \((\alpha \rightarrow \alpha, \beta \rightarrow \beta)\) rather than \((\alpha \rightarrow \alpha, \alpha \rightarrow \alpha)\), since the former type has strictly more instances than the latter.

\[
[\epsilon_{1} \text{-Alg}] \\
\Gamma \vdash_{\alpha} [(x \in \tau) ? x_1 : x_2 \mid \epsilon_{1}(\Sigma)] : \Gamma(x_1)
\]

To type type-cases, the annotation indicates which of the three rules must be applied (here \([\epsilon_{1}]\)) and how to instantiate the polymorphic type variables occurring in the type of the tested expression, so that it satisfies the side condition of the applied rule (see also \([\epsilon_{2} \text{-Alg}]\) and \([0 \text{-Alg}]\) in Appendix G).

\[
[\text{Bind}_{1} \text{-Alg}] \\
\Gamma \vdash_{\alpha} \Gamma \vdash_{\alpha} [\text{bind } a \in \kappa \mid \text{skip } k] : t \quad x \not\in \text{dom}(\Gamma)
\]

In rule \([\text{Bind}_{1} \text{-Alg}]\) the annotation indicates to skip the definition of the current binding. This rule is used when the binding variable is not required for typing the body \(\kappa\) under the current context \(\Gamma\). For instance, this is the case when \(x\) only appears in a branch of a typecase that cannot be taken under \(\Gamma\). The side condition \(x \not\in \Gamma\) prevents a potential unsound name conflict between binding variables: as occurrences of \(x\) in \(\kappa\) denote the \(x\) binding variable that is being skipped, having the type of a former binding variable \(x\) in our environment when typing \(\kappa\) would be unsound.

\[
[\text{Bind}_{2} \text{-Alg}] \\
\Gamma \vdash_{\alpha} [a \mid \emptyset] : s \quad (\forall i \in I) \quad \Gamma, x : s \land \mathbf{u}_i \vdash_{\alpha} [k \mid k_i] : t_i \\
\Gamma \vdash_{\alpha} [\text{bind } x = a \in \kappa \mid \text{keep } (\mathbf{u}_i, \{k_i \mid k\})_{i \in I}] : \bigvee_{i \in I} t_i \quad I \neq \emptyset, \quad \forall i \in I \mathbf{u}_i \equiv \mathbb{I}
\]

This rule tries to type the bound atom and then decomposes its type according to the annotation. This decomposition corresponds to an application of the \([\lor]\) rule of the declarative type system with the only difference that the type \(s\) of the atom is split in several summands by intersecting it with the various \(\mathbf{u}_i\) (instead of just two summands as in the rule \([\lor]\)) whose union covers \(1\).

Finally, two annotations indicate when and how to apply rule \([\land]\) to atoms and canonical forms:

\[
[\land - \text{Alg}] \\
(\forall i \in I) \quad \Gamma, a \vdash_{\alpha} [k \mid k_i] : t_i \quad I \neq \emptyset \\
\Gamma \vdash_{\alpha} [\land ((\{k_i \mid k\})_{i \in I})] : \land_{i \in I} t_i \quad I \neq \emptyset
\]

An expression \(e\) is typable if and only if its unique (modulo \(\equiv_{\kappa}\)) MSC-form is typable, too:

**Theorem 3.4 (Soundness and Completeness).** For every closed term \(e\) of the source language

\[
\vdash e : t \quad \Rightarrow \quad \exists k \quad \vdash_{\alpha} [\text{MSC}(e) \mid k] : t' \leq t \quad \text{(completeness)}
\]

\[
\vdash e : t \quad \Leftrightarrow \quad \vdash_{\alpha} [\text{MSC}(e) \mid k] : t \quad \text{(soundness)}
\]

It is easy to generate the unique MSC-form associated to a closed source language expression \(e\) (cf. Appendix E). Theorem 3.4 states that this MSC-form is typable if and only if \(e\) is: we reduced the problem of typing \(e\) to the one of finding an annotation that makes the unique MSC-form of \(e\) typeable with the algorithmic type system. Figure 3 gives an example of an MSC-form and two possible annotations for it. The term "\(\lambda x. (fx \mathbin{\in} \mathbb{I}nt) ? g(fx) : x''\) (where \(f : \forall \alpha. \alpha \rightarrow \alpha\) and \(g : \mathbb{I}nt \rightarrow \mathbb{I}nt\)) is put in MSC-form (on the left). In the first annotation, the function is typed with a single \(\lambda\) annotation (l. 3). The interesting part is the annotation of the binding for \(u\) (l. 5): the corresponding keep annotation represents an application of the union elimination rule on the occurrences of the expression \(fx\) whose type \(\beta\) is split into \(\beta \land \mathbb{I}nt\) (l. 6) and \(\beta \land \mathbb{I}nt\) (l. 9). Each subcase is annotated accordingly. Notice in the second subcase that the annotation for \(v\) is \(\text{skip}\) (l. 10) which indicates that this particular variable must not be used (as \(g(fx)\) cannot be typed since in the "else" branch, \(fx\) has type \(\text{~} \mathbb{I}nt\)). A different annotation, yielding a better type, is the one on the right. This
intersection annotation (l. 2) separates the domain of the $\lambda$-abstraction into two cases, each typed independently, yielding for the whole function an intersection type. The problem of inferring such annotations, in particular the second one, is tackled in the next section.

4 RECONSTRUCTION

This section describes an algorithm to find an annotation for an expression in MSC-form, such that the pair expression and annotation is typable in the algorithmic system. Though this algorithm is not complete (inference for systems with intersection types is known to be undecidable, [Pottinger 1980]), it is sound and terminating. Experimental results are presented in Section 5.

The annotation reconstruction algorithm is composed of two systems of deduction rules: the main reconstruction algorithm (Section 4.2) which produces intermediate annotations containing information about the domains of $\lambda$-abstractions and the type decompositions to use in bindings, and the auxiliary reconstruction algorithm (Section 4.3) which converts these intermediate annotations into annotations for the algorithmic type system, by computing instantiations $\Sigma$ for destructors.

4.1 The tallying algorithm

One key ingredient used by the reconstruction algorithm is the tallying algorithm. Roughly, tallying is the equivalent of the unification used in algorithm $\mathcal{W}$ [Damas and Milner 1982], but for a type system with subtyping. The tallying algorithm was introduced by Castagna et al. [2015] to solve the following problem: given a set of pairs $\{(t_i, t'_i)\}_{i \in I}$ and a set of type variables $\Delta$ representing the monomorphic type variables, find all substitutions $\Sigma$ whose domain is disjoint from $\Delta$ and that satisfy $\forall i \in I. \ t_i \Sigma \leq t'_i \Sigma$. Castagna et al. [2015] show that this problem is decidable and give an algorithm to characterize all solutions. As for unification, for each instance of the tallying problem there is either no solution or several substitutions each of which is a solution of the problem. The difference is that while with unification all solutions are characterized by a principal substitution, with tallying they are characterized by a principal finite set of substitutions. More precisely, the solutions to a tallying instance are characterized by a set $\Sigma$ of substitutions, such that every $\Sigma \in \Sigma$ is characterized by a set containing three distinct substitutions: $\{\alpha \leadsto s, \beta \leadsto t\}$, where all substitutions that make the former type become a subtype of the latter are characterized by a set containing three distinct substitutions: $\{\alpha \leadsto s, \beta \leadsto t\}$.

6This is due to the presence of the empty type. For instance, the principal solution of unifying $\alpha \times \beta$ with $s \times t$ is the substitution $\{\alpha \leadsto s, \beta \leadsto t\}$, while all substitutions that make the former type become a subtype of the latter are characterized by a set containing three distinct substitutions: $\{\alpha \leadsto s, \beta \leadsto t\}$. More precisely, the solutions to a tallying instance are characterized by a set $\Sigma$ of substitutions, such that every $\Sigma \in \Sigma$ is characterized by a set containing three distinct substitutions: $\{\alpha \leadsto s, \beta \leadsto t\}$. More precisely, the solutions to a tallying instance are characterized by a set $\Sigma$ of substitutions, such that every $\Sigma \in \Sigma$ is characterized by a set containing three distinct substitutions: $\{\alpha \leadsto s, \beta \leadsto t\}$. More precisely, the solutions to a tallying instance are characterized by a set $\Sigma$ of substitutions, such that every $\Sigma \in \Sigma$ is characterized by a set containing three distinct substitutions: $\{\alpha \leadsto s, \beta \leadsto t\}$.
a solution, and for any solution $\sigma$, we have $\exists \sigma_1 \in \Sigma. \exists \sigma_2. \sigma \simeq \sigma_2 \circ \sigma_1$, where $\circ$ denotes the composition of substitutions and $\simeq$ is pointwise type equivalence.

In this work, all tallying instances use a single constraint, and we will note $\text{tally}(\{t_1 \triangleq t_2\})$ the set of substitutions $\Sigma$ characterizing all the solutions of the tallying instance $\{(t_1, t_2)\}$, where $\Delta = \mathcal{V}_{\mathcal{M}}$ (and thus $\forall \sigma \in \Sigma. \text{dom}(\sigma) \subseteq \mathcal{V}_{\mathcal{P}}$), and such that all new type variables introduced in the solutions are polymorphic (i.e., $\forall \sigma \in \Sigma. \forall \alpha \in \text{dom}(\sigma). \text{vars}(\sigma(\alpha)) \cap \mathcal{V}_{\mathcal{M}} \subseteq \text{vars}(t_1) \cup \text{vars}(t_2)$).

The tallying function $\text{tally()}$ finds substitutions for polymorphic type variables, but in order to infer the domain of $\lambda$-abstractions, we may need to find substitutions for monomorphic type variables. We thus introduce an additional tallying function, $\text{tally\_infer}(\{t_1 \leq t_2\})$:

**Definition 4.1.** Let $\sigma|_X$ denote the restriction of the substitution $\sigma$ to the domain $X$. We define

$$\text{tally\_infer}(\{t_1 \leq t_2\}) = \{(\sigma \circ \sigma' \circ \phi)|_{\mathcal{V}_{\mathcal{M}}} \mid \sigma' \in \text{tally}(\{\text{fresh}(t_1)\phi \leq \text{fresh}(t_2)\phi\})\}$$

where $\text{fresh}(t)$ denotes the type $t$ where polymorphic type variables have been substituted by fresh ones; $\phi$ is a renaming from $(\text{vars}(t_1) \cup \text{vars}(t_2)) \cap \mathcal{V}_{\mathcal{M}}$ to fresh polymorphic variables; and $\sigma$ is a substitution from $\mathcal{V}_{\mathcal{P}}$ to fresh monomorphic variables.

In a nutshell, polymorphic type variables in $t_1$ and $t_2$ are refreshed in order to decorrelate them, and monomorphic type variables are generalized using $\phi$ so that $\text{tally()}$ is allowed to find solutions for them. Each solution $\sigma'$ is composed with $\phi$ in order to restore the connection with the initial monomorphic type variables, and the polymorphic type variables in the image of the resulting substitution are transformed into monomorphic ones by composing $\sigma$ with it. Finally, the substitution is restricted to $\mathcal{V}_{\mathcal{M}}$ (i.e., to the domain of $\phi$).

For example, an instance such as $\text{tally\_infer}(\{\text{Int} \wedge \alpha \rightarrow \text{Int} \wedge \alpha \leq \beta \rightarrow \alpha\})$ can be generated during reconstruction, when a function of type $\text{Int} \wedge \alpha \rightarrow \text{Int} \wedge \alpha$ is applied to an argument of type $\beta$, but the $\alpha$ on the right-hand of $\leq$ is unrelated to the one on the left-hand side. Decorrelating them yields a unique solution $\{\beta \leadsto \beta' \wedge \text{Int}\}$, that is, $\beta$ must be substituted by $\beta' \wedge \text{Int}$ in our context for the application to be typeable.

### 4.2 Main reconstruction algorithm

The main reconstruction algorithm, defined in this section, infers the domains of $\lambda$-abstractions and the decompositions of types into disjoint unions to use for bindings. It works by successively refining intermediate annotations defined below. These intermediate annotations store information about the domains of lambdas and the decompositions of bindings. However, the instantiations $\Sigma$ used to type destructors (i.e., applications, projections, and typecases) in the algorithmic type system are not stored in intermediate annotations, because they might get invalidated as the reconstruction progresses: when new information is found about the domain of a lambda or the decomposition of a binding, the algorithm will retype some intermediate definitions of the MSC-form, thus invalidating the instantiations $\Sigma$ of later definitions. Thus, these instantiations $\Sigma$ will be recomputed whenever needed, using the auxiliary system (Section 4.3) that converts intermediate annotations into annotations for the algorithmic type system.

Atom and form intermediate annotations are defined by the grammar below:

- **Split annotations**: $S ::= \{(u,K),\ldots,(u,K)\}$
- **Atom intermediate annot.**: $\mathcal{A} ::= \text{infer} \mid \text{untyp} \mid \text{typ} \mid \wedge (\{\mathcal{A},\ldots,\mathcal{A}\};\{\mathcal{A},\ldots,\mathcal{A}\}) \mid \epsilon_1 \mid \epsilon_2 \mid \lambda(u,K)$
- **Form intermediate annot.**: $\mathcal{K} ::= \text{infer} \mid \text{untyp} \mid \text{typ} \mid \wedge (\{K,\ldots,K\};\{K,\ldots,K\}) \mid \text{try\_skip}(K) \mid \text{try\_keep}(\mathcal{A},K,K) \mid \text{propagate}(\mathcal{A},\Gamma,S,S) \mid \text{skip}(K) \mid \text{keep}(\mathcal{A},S,S)$
In the following, we use the metavariable \( \eta \) to range over both atoms and expressions (i.e., \( \eta := a \mid k \)). Similarly, the metavariable \( \mathcal{H} \) ranges over atom annotations \( a \) and form annotations \( k \) (i.e., \( \mathcal{H} := a \mid k \)); while the metavariable \( \mathcal{A} \) ranges over atom intermediate annotations \( \mathcal{A} \) and form intermediate annotations \( \mathcal{K} \) (i.e., \( \mathcal{H} := \mathcal{A} \mid \mathcal{K} \)).

Let \( \psi \) range over monomorphic substitution, that is, substitutions from \( V_M \) to monomorphic types, and \( \Psi \) range over finite sets of monomorphic substitutions (\( \Psi := \{\psi, \ldots, \psi\} \)). The main reconstruction algorithm is presented as a deduction rule system, for judgments of the form \( \Gamma \vdash^* \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathcal{R} \), where \( \mathcal{R} \) is a result defined as follows:

**Result**

\[
\mathcal{R} := \text{Ok}(\mathcal{H}) | \text{Fail} | \text{Split}(\Gamma, \mathcal{H}, \mathcal{H}) | \text{Subst}(\Psi, \mathcal{H}, \mathcal{H}) | \text{Var}(x, \mathcal{H}, \mathcal{H})
\]

Let us see what each result for \( \Gamma \vdash^* \langle \eta \mid \mathcal{H} \rangle \) means:

- **Ok(\( \mathcal{H} \))**: the reconstruction was successful and \( \eta \) can be typed by the algorithmic type system using the annotation \( \mathcal{H} \) (after converting it into an annotation \( \mathcal{H} \) using the auxiliary reconstruction system). This result is terminal (i.e., it is a definitive answer that cannot be further refined).

- **Fail**: the reconstruction has failed. The algorithm was not able to find an annotation that makes \( \eta \) typable with the algorithmic system. This result is terminal.

- **Subst(\( \Psi, \mathcal{H}_1, \mathcal{H}_2 \))**: the reconstruction found a set of substitutions \( \Psi \) that if applied to \( \Gamma \) may make \( \eta \) typable. In practice, for each substitution \( \psi \in \Psi \), the reconstruction will be called again on the environment \( \Gamma \psi \) and annotation \( \mathcal{H}_1 \psi \). However, this does not necessarily mean that the reconstruction will fail on the current environment \( \Gamma \): \( \eta \) might still be typeable but with a less precise type (e.g., it could yield an arrow type with a smaller domain). Thus, this default case which does not instantiate \( \Gamma \) is also explored, using the annotation \( \mathcal{H}_2 \) instead of \( \mathcal{H}_1 \).

- **Split(\( \Gamma', \mathcal{H}_1, \mathcal{H}_2 \))**: the reconstruction found some splits for the variables in \( \text{dom}(\Gamma') \) that if applied to \( \Gamma \) may make \( \eta \) typable. In practice, the system generates several new environments: one is obtained by (pointwise) intersecting \( \Gamma \) with \( \Gamma' \) and then it is used to retype \( \eta \) with the annotation \( \mathcal{H}_1 \); the others are obtained by intersecting \( \Gamma \) with all the possible pointwise negations of \( \Gamma' \) and then they are used to retype \( \eta \) with the annotation \( \mathcal{H}_2 \).

- **Var(\( x, \mathcal{H}_1, \mathcal{H}_2 \))**: the reconstruction found that in order to type \( \eta \), the definition of the bind-abstracted variable \( x \) should be typed. Any branch that successfully types it continues with the annotation \( \mathcal{H}_1 \), otherwise it continues with the annotation \( \mathcal{H}_2 \).

Initially, any form or atom \( \eta \) is annotated with \( \text{infer} \), and this annotation is then refined until it yields a terminal result (i.e., either \( \text{Ok}(\cdot) \) or \( \text{Fail} \)). The rules below are presented by decreasing priority (i.e., the first rule that applies is used). Some rules have been omitted for concision, but the reader can find the full reconstruction system in Appendix H.1.

There are two different forms of judgments: \( \Gamma \vdash^* \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathcal{R} \) and \( \Gamma \vdash \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathcal{R} \). We first define rules for the judgment \( \vdash^* \) for every canonical form and atom. The results of these judgments are not necessarily terminal and, therefore, it may be necessary to call the reconstruction again in order to refine them. This is the purpose of \( \vdash^* \) judgments which call repetitively \( \vdash \) judgments when relevant, so that in the end we get a terminal result. Let us first focus on \( \vdash \) judgments.

\[
\begin{align*}
\text{[Ok]} & \quad \Gamma \vdash \langle \eta \mid \text{typ} \rangle \Rightarrow \text{Ok(\text{typ})} \\
\text{[Fail]} & \quad \Gamma \vdash \langle \eta \mid \text{untyp} \rangle \Rightarrow \text{Fail}
\end{align*}
\]

If a canonical form or atom \( \eta \) is annotated with \( \text{typ} \), then reconstruction is finished for \( \eta \), and it is typeable in the current context \( \Gamma \). The annotation \( \text{typ} \) is never used on lambdas and bindings because the system needs to store more information for them. Likewise, if a form or atom \( \eta \) is
annotated with untyp, then reconstruction is finished for $\eta$ by failing in the current context.

\[
\begin{align*}
[AxOk] & \quad x \in \text{dom}(\Gamma) \\
\Gamma \vdash R \langle x \mid \text{infer} \rangle \Rightarrow \text{Ok}(\text{typ})
\end{align*}
\]

\[
\begin{align*}
[AxFail] & \quad \Gamma \vdash R \langle x \mid \text{infer} \rangle \Rightarrow \text{Fail}
\end{align*}
\]

If a $\lambda$-abstracted variable $x$ is in the environment, then it is typeable and thus the algorithm returns $\text{Ok}(\text{typ})$. Otherwise, $x$ is undefined and $\text{Fail}$ is returned.

\[
\begin{align*}
[AppVar_i] & \quad x_i \notin \text{dom}(\Gamma) \\
\Gamma \vdash R \langle x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Var}(x_i, \text{infer, untyp})
\end{align*}
\]

To type the application $x_1 x_2$, we must first ensure that $\{x_1, x_2\} \subseteq \text{dom}(\Gamma)$. If it is not the case, then the two rules $[AppVar_i]$ (for $i = 1, 2$) try to remedy it by returning $\text{Var}(x_i, \text{infer, untyp})$, which is the result that asks the system to try to type the atom bound to $x_i$ for $x_i \notin \text{dom}(\Gamma)$. If the attempt is successful, then the algorithm will continue the reconstruction for the application with the annotation $\text{infer}$ and $x_i \in \text{dom}(\Gamma)$, otherwise it will continue with the annotation untyp making the reconstruction fail on this application.

\[
\begin{align*}
[AppInf] & \quad \Psi = \text{tally\_infer}(\{\Gamma(x_i) \leq \Gamma(x_2) \rightarrow \alpha\}) \\
\Gamma \vdash R \langle x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Subs}((\Psi, \text{typ, untyp}) \quad \alpha \in V_p \text{ fresh})
\end{align*}
\]

If $\{x_1, x_2\} \subseteq \text{dom}(\Gamma)$, then the rule $[AppInf]$ tries to find all instances of the current context in which the application $x_1 x_2$ is typeable, by subsuming $\Gamma(x_1)$ (the type of the function) to $\Gamma(x_2) \rightarrow \alpha$ (a function type whose domain is the type of the argument). For that, it calls the tallying algorithm which returns a set of substitutions $\Psi$. Then, $\text{Subs}((\Psi, \text{typ, untyp})$ is returned, meaning that this application should be typeable under every instance $\Gamma\psi$ of the current context $\Gamma$ (with $\psi \in \Psi$). The default case (i.e., when the current context is unchanged, for example, when $\Psi = \emptyset$) cannot be typed, so it is annotated with untyp (see rule $[Iterate_2]$ later on). The rules for pairs are similar and have been omitted.

\[
\begin{align*}
[Casesplit] & \quad \Gamma(x) \not\subseteq \tau \quad \Gamma(x) \not\subseteq \neg\tau \\
\Gamma \vdash R \langle (x \in \tau) \mid x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Split}((\{(x : \tau)\}), \text{infer, infer})
\end{align*}
\]

\[
\begin{align*}
[CaseThen] & \quad \Gamma(x) \leq \tau \quad \Psi = \text{tally\_infer}(\{\Gamma(x) \leq 0\}) \\
\Gamma \vdash R \langle (x \in \tau) \mid x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Subs}((\Psi, \text{typ, } \epsilon_1)
\end{align*}
\]

\[
\begin{align*}
[CaseElse] & \quad \Gamma(x) \leq \neg\tau \quad \Psi = \text{tally\_infer}(\{\Gamma(x) \leq 0\}) \\
\Gamma \vdash R \langle (x \in \tau) \mid x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Subs}((\Psi, \text{typ, } \epsilon_2)
\end{align*}
\]

\[
\begin{align*}
[CaseVar_i] & \quad x_i \notin \text{dom}(\Gamma) \\
\Gamma \vdash R \langle (x \in \tau) \mid x_1 x_2 \mid \epsilon_i \rangle \Rightarrow \text{Var}(x_i, \text{typ, untyp})
\end{align*}
\]

The key rule for type-cases is $[Casesplit]$, corresponding to the case where $x$ is in $\Gamma$, but with a type that does not allow the selection of a specific branch. Thus, we need to partition the type of $x$ in two, one part being a subtype of $\tau$ and the other a subtype of $\neg\tau$. This is achieved by returning $\text{Split}((\{(x : \tau)\}), \text{infer, infer})$: this result is backtracked up to the binding of $x$, where it will split the associated type, accordingly.

When the type of $x$ allows the selection of a branch, then either the rule $[CaseThen]$ or the rule $[CaseElse]$ applies. If we are in the case of $[CaseThen]$, that is $\Gamma(x) \leq \tau$, then we have to determine whether we will apply the algorithmic rule $[0\text{-Alg}]$ or the algorithmic rule $[\epsilon_1\text{-Alg}]$. To determine it, the $[CaseThen]$ rule calls tally_infer($\{\Gamma(x) \leq 0\}$) which returns the set of contexts
\(\Gamma \psi\) (for \(\psi \in \Psi\)) under which the algorithmic rule \([\emptyset]\text{-}\text{ALG}\) is to be applied, that is, the contexts under which the tested expression \(x\) has an empty type. The default case, corresponding to the case in which the type of \(\Gamma(x)\) is not guaranteed to be empty and, thus, in which the algorithmic rule \([\epsilon_1]\text{-}\text{ALG}\) must be applied, is annotated with \(\epsilon_1\). This annotation is handled by the rule \([\text{CASEVAR}_1]\) which forces the system to type \(x_1\), the binding variable associated to the first branch. The case for \([\text{CASEELSE}]\) and \([\text{CASEVAR}_2]\) is analogous.

We omitted the remaining rules for type-cases since they are straightforward: the rule for \(x \not\in \text{dom}(\Gamma)\), which triggers a \(\text{Var}\ (x, \text{infer}, \text{untyp})\) result; two rules similar to \([\text{CASEVAR}_1]\), but where \(x_i \in \text{dom}(\Gamma)\), which simply return \(0k(\text{typ})\).

The rules for \(\lambda\)-abstractions mimic algorithm \(\mathcal{W}\). Rule \([\text{LAMBDAINFERENCE}]\) transforms the initial \text{infer} annotation into a \(\lambda(\alpha, \text{infer})\) annotation. As in \(\mathcal{W}\), \(\lambda\)-abstracted variables are initially given a fresh type variable, which will then be substituted as needed while reconstructing the type of the body; here we use a fresh monomorphic variable, but \(\text{tally}_{\text{infer}}()\) will transform it into a polymorphic—thus, instantiable—one, just for the reconstruction in the body. Rule \([\text{LAMBDA}]\) adds the \(\lambda\)-abstracted variable to the environment with the type specified in the annotation, recursively calls reconstruction on the body, and reestablishes the variable type annotation on the result. The notation \(\text{map}(X \mapsto f(X), R)\) denotes the result \(R\) where \(f\) has been applied to every annotation \(X\).

The \([\text{BINDINFERENCE}]\) rule transforms an initial \text{infer} annotation into a \text{try-skip (infer)} annotation which skips the binding and annotates the body \(\kappa\) with \text{infer}. We do not try to type the definition of a binding until it is actually used, because its variable might appear only in unreachable positions (e.g., in an unreachable branch of a type-case). In other words, we implement a lazy typing discipline for bind-abstracted variables. If the variable is used at some point, then an attempt to type it will be initiated by the \([\text{BINDTRYSKIP}_1]\) rule below:

\[
\frac{\Gamma \vdash R (\text{bind } a \text{ in } \kappa \ | \ \text{try-skip (infer)}) \Rightarrow R}{\Gamma \vdash R (\text{bind } a \text{ in } \kappa \ | \ \text{infer}) \Rightarrow R}
\]

This rule tries to type the body of the binding, starting with the annotation \(\mathcal{K}\) (initially \text{infer}). If the result is a \(\text{Var}\ (x, \mathcal{K}_1, \mathcal{K}_2)\), then it means that the current binding is used in the body \(\kappa\) and, thus, the system should try to type it. Consequently, the annotation for the current binding is changed into a \text{try-keep (infer, } \mathcal{K}_1, \mathcal{K}_2)\) so that, at the next iteration, its definition will be reconstructed.
If typing the body of the binding yields a result different from \( \text{Var} (x, \mathcal{K}_1, \mathcal{K}_2) \), then this result is just propagated as in [LAMBDA] (the corresponding rules have been omitted).

\[
\begin{align*}
\Gamma \vdash^* \langle a \mid \mathcal{A} \rangle & \Rightarrow \text{Ok}(\mathcal{A}') \\
\text{[BindTryKeep1]} & \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{keep} (\mathcal{A}', \{(1, \mathcal{K}_1), \emptyset\}) \rangle \Rightarrow \mathcal{R} \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{try-keep} (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathcal{R}
\end{align*}
\]

\[
\begin{align*}
\text{[BindTryKeep2]} & \\
\Gamma \vdash^* \langle a \mid \mathcal{A} \rangle & \Rightarrow \text{Fail} \quad \Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{skip} (\mathcal{K}_2) \rangle \Rightarrow \mathcal{R} \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{try-keep} (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathcal{R}
\end{align*}
\]

As expected, if the current annotation for the binding is a \( \text{try-keep} (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \), then the system tries to reconstruct the annotation for the definition. If it succeeds, then it becomes possible to type the definition and to continue the reconstruction of the body using \( \mathcal{K}_1 \). This is what [BindTryKeep1] does by changing the current annotation to \( \text{keep} (\mathcal{A}', \{(1, \mathcal{K}_1), \emptyset\}) \) (more details below). If the reconstruction of the definition fails (rule [BindTryKeep2]), then we have no choice but to skip this definition and use the default annotation \( \mathcal{K}_2 \) to type the body.

In an annotation \( \text{keep} (\mathcal{A}, \mathcal{S}, \mathcal{S}') \) for the binding of a variable \( x \), \( \mathcal{A} \) is the annotation for typing the definition of \( x \), while the two other arguments describe the type decomposition to use for \( x \) and, for each part of the decomposition, the annotation to use for the body. More precisely, \( \mathcal{S} \) contains the parts of the type decomposition that have yet to be explored, and \( \mathcal{S}' \) contains the parts that have already been fully explored. In particular, the annotation \( \text{keep} (\mathcal{A}', \{(1, \mathcal{K}_1), \emptyset\}) \) used in rule [BindTryKeep1] means that the type of the definition does not need to be partitioned: there is only one part, covering \( 1 \), associated with an annotation \( \mathcal{K}_1 \) for typing the body.

\[
\begin{align*}
\text{[BindOk]} & \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{keep} (\mathcal{A}, \emptyset, \mathcal{S}) \rangle \Rightarrow \text{Ok}(\text{keep} (\mathcal{A}, \emptyset, \mathcal{S}))
\end{align*}
\]

If all the parts of the type decomposition have already been explored (i.e., if \( \mathcal{S} = \emptyset \)), then the reconstruction is successful. Otherwise, the following rules are applied:

\[
\begin{align*}
\text{[BindSplit1]} & \\
\Gamma \vdash \langle a \mid \mathcal{A} \rangle & \Rightarrow \varnothing \quad \Gamma \vdash a \mid \mathcal{A} : s \quad \Gamma, x : s \land \mathcal{U} \vdash^* \langle \kappa \mid \mathcal{K} \rangle & \Rightarrow \text{Ok}(\mathcal{K}') \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{keep} (\mathcal{A}, \mathcal{S}, \{(\mathcal{U}, \mathcal{K}') \cup \mathcal{S}'\}) \rangle \Rightarrow \mathcal{R} \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{keep} (\mathcal{A}, \{(\mathcal{U}, \mathcal{K}) \cup \mathcal{S}, \mathcal{S}'\}) \rangle \Rightarrow \mathcal{R}
\end{align*}
\]

\[
\begin{align*}
\text{[BindSplit2]} & \\
\Gamma \vdash \langle a \mid \mathcal{A} \rangle & \Rightarrow \varnothing \quad \Gamma \vdash a \mid \mathcal{A} : s \quad \Gamma, x : s \land \mathcal{U} \vdash^* \langle \kappa \mid \mathcal{K} \rangle & \Rightarrow \text{Split}(\mathcal{I}', \mathcal{K}_1, \mathcal{K}_2) \\
x \in \text{dom}(\mathcal{I}') \quad \Gamma \vdash e \langle a : \neg (\mathcal{U} \land \mathcal{I}'(x)) \rangle \Rightarrow \mathcal{I}_1 \quad \Gamma \vdash e \langle a : \neg (\mathcal{U} \land \mathcal{I}'(x)) \rangle \Rightarrow \mathcal{I}_2 \\
\Gamma \vdash \langle \text{bind} x = a \text{ in } \kappa \mid \text{keep} (\mathcal{A}, \{(\mathcal{U}, \mathcal{K}) \cup \mathcal{S}, \mathcal{S}'\}) \rangle & \Rightarrow \text{Split}(\mathcal{I}' \setminus x, \mathcal{K}_1', \mathcal{K}_2') \\
\text{with, in the last rule, } \mathcal{K}_1' = \text{propagate} (\mathcal{A}, \mathcal{I}_1 \cup \mathcal{I}_2, \{(\mathcal{U} \land \mathcal{I}'(x), \mathcal{K}_1), (\mathcal{U} \land \mathcal{I}'(x), \mathcal{K}_2) \cup \mathcal{S}, \mathcal{S}'\}) \text{ and } \mathcal{K}_2' = \text{keep} (\mathcal{A}, \{(\mathcal{U}, \mathcal{K}) \}) \cup \mathcal{S}, \mathcal{S}'
\end{align*}
\]

In both rules, the definition of the binding is typed using the annotation \( \mathcal{A} \). For that, it is first converted into an annotation \( \varnothing \) of the algorithmic type system, using the deduction rules for the judgment \( \Gamma \vdash \langle a \mid \mathcal{A} \rangle \Rightarrow \varnothing \), defined in Section 4.3. Then, the type \( s \) obtained for the definition is partitioned according to the split annotation in the second argument of \( \text{keep}() \) (i.e., \( \{(\mathcal{U}, \mathcal{K}) \} \cup \mathcal{S} \) for both rules), and each part is explored one after the other, using its own annotation for the body. Note that, since split annotations are sets, the order in which the branches are explored is arbitrary.

The rule [BindSplit1] for an annotation \( \text{keep} (\mathcal{A}, \mathcal{S}, \mathcal{S}') \) is responsible for moving a branch from \( \mathcal{S} \) to \( \mathcal{S}' \) when the result for the branch is \( \text{Ok}() \). If instead the reconstruction of the body requires to further split the type of \( x \), then the rule [BindSplit2] splits the current branch into two
branches. However, before exploring these two branches, some information about the split needs to be propagated, to ensure that when a split is explored, it is under a context as precise as possible.

Let us explain this by an example. Assume we have a polymorphic primitive function \( \text{id} \) of type \( \alpha \to \alpha \) and an initial environment \( \Gamma = \{ x : \text{Bool} \} \). We want to type the following canonical form, and deduce for it the type \( \text{True} \) (since \( x \) and \( y \) are always bound to the same value):

\[
\text{bind} x = \text{bind} y = \text{id} \quad x \in \text{bind} z = (y \in \text{True}) \quad ? x : \text{true in z }
\]

At some point, the partition associated to \( y \) will change from \( \{1\} \) to \( \{\text{True}, \neg \text{True}\} \) because of the type-case (rule \[\text{CaseSplit}\]). However, if the case corresponding to \( (y : \text{True}) \) is immediately explored, it will yield for the body the type \( \text{Bool} \), because \( x \) still has the type \( \text{Bool} \) in the environment.

In order to obtain the more precise type \( \text{True} \), we must deduce, before exploring the case \( (y : \text{True}) \), that when \( \text{id} x \) (the definition of \( y \)) has type \( \text{True} \), then \( x \) also has type \( \text{True} \). Knowing that, the type of \( x \) should be split accordingly into \( \{\text{True}, \neg \text{True}\} \).

This mechanism of backward propagation of splits is initiated in the \[\text{BindSplit}_2\] rule with the two premises \( \Gamma \vdash (a : (u \land \Gamma'(x))) \Rightarrow \Gamma_1 \) and \( \Gamma \vdash (a : (u \land \Gamma'(x))) \Rightarrow \Gamma_2 \). This auxiliary judgment \( \Gamma \vdash (a : u) \Rightarrow \Gamma \), defined in Appendix H.3, can be read as follows: refining the current environment \( \Gamma \) with one of the \( \Gamma' \in \Gamma \) ensures that the atom \( a \) will have type \( u \). The refinements we obtain are stored in the annotation of the binding, using an annotation \( \text{propagate} \) \( (\mathcal{A}, \Gamma, S, S') \). This annotation is handled by two other rules (omitted here) whose role is to propagate these refinements one after the other using successive \( \text{Split}(\Gamma', \mathcal{K}_1, \mathcal{K}_2) \) results (with \( \Gamma'' \in \Gamma \)), before finally restoring a keep \( (\mathcal{A}, S, S') \) annotation.

\[
\begin{align*}
\text{[InterEmpty]} & \quad \Gamma \vdash \langle \eta \mid \text{\land}(\emptyset, \emptyset) \rangle \Rightarrow \text{Fail} \\
\text{[InterOk]} & \quad \Gamma \vdash \langle \eta \mid \text{\land}(\emptyset, S) \rangle \Rightarrow \text{Ok}(\emptyset \leq \emptyset, S) \\
\text{[Inter1]} & \quad \Gamma \vdash^* \langle \eta \mid \mathcal{H} \rangle \Rightarrow \text{Ok}(\mathcal{H}') \quad \Gamma \vdash \langle \eta \mid \text{\land}(S, \{\mathcal{H}' \cup S', S') \rangle \Rightarrow \mathbb{R} \\
\text{[Inter2]} & \quad \Gamma \vdash \langle \eta \mid \text{\land}(\{\mathcal{H} \cup S, S') \rangle \Rightarrow \mathbb{R} \\
\text{[Inter3]} & \quad \Gamma \vdash \langle \eta \mid \text{\land}(\{\mathcal{H} \cup S, S') \rangle \Rightarrow \text{map}(X \mapsto (\text{\land}(X \cup S, S')) \rangle \Rightarrow \mathbb{R})
\end{align*}
\]

Intersection annotations are introduced by the \( \vdash^* \) judgments defined below. In an intersection annotation \( \text{\land}(S, S') \), the annotations in \( S' \) are fully processed (i.e., the associated reconstruction returned \( \text{Ok}(\cdot) \)), while the annotations in \( S \) are not: they still have to be refined one after the other (rule \[\text{Inter3}\]). If one of them becomes fully processed, it is moved in \( S' \) (rule \[\text{Inter1}\]). Conversely, if one of them fails, it is removed (rule \[\text{Inter2}\]). The process stops when \( S \) is empty: then, the reconstruction fails if \( S' \) is empty (rule \[\text{InterEmpty}\]), and succeed otherwise (rule \[\text{InterOk}\]).

Finally, we formalize the rules for the judgments \( \vdash^* \). As said earlier, the purpose of \( \vdash^* \) is to repeatedly call \( \vdash R \) judgments so that, in the end, we obtain a terminal result.

\[
\begin{align*}
\text{[Iterate1]} & \quad \Gamma \vdash \langle \eta \mid \mathcal{H} \rangle \Rightarrow \text{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2) \quad \Gamma \vdash^* \langle \eta \mid \mathcal{H}_1 \rangle \Rightarrow \mathbb{R}' \\
\text{[Iterate2]} & \quad \Gamma \vdash \langle \eta \mid \text{\land}(\text{\{fresh}(\mathcal{H}_1 \psi_i)\}_{i \in I} \cup \{\mathcal{H}_2, \emptyset\}) \Rightarrow \mathbb{R}' \quad \forall i \in I. \psi_i \# \Gamma
\end{align*}
\]
4.3 Auxiliary reconstruction algorithm

The auxiliary reconstruction algorithm defined in this section converts an intermediate annotation of the main reconstruction system into an annotation for the algorithmic type system. For that, it needs to retrieve the polymorphic substitutions $\Sigma$ needed to type the atoms.

Formally, the algorithm takes as input an environment $\Gamma$, an atom or canonical form $\eta$, and an intermediate annotation $\mathcal{H}$, and produces an annotation $\h$ for the algorithmic type system. It is presented as a deduction rule system for judgments of the form $\Gamma \vdash \langle \eta \mid \mathcal{H} \rangle \Rightarrow \h$. Some rules have been omitted for concision (they can be found in Appendix H.2): for instance, the rules for constants and axioms are omitted since straightforward, as they just transform an intermediate annotation typ into an annotation $\emptyset$ for the algorithmic type system; likewise, the rules for lambdas and intersections are straightforward and have been omitted, since they just proceed recursively on their children annotations. The most important rule for this system is the one for applications:

$$
\begin{align*}
\text{[App]} & \quad t_1 = \Gamma(x_1) \quad t_2 = \Gamma(x_2) \quad \rho_1 = \text{refresh}(t_1) \\
\quad \rho_2 = \text{refresh}(t_2) \quad \Sigma = \text{tally}(\{t_1 \rho_1 \leq t_2 \rho_2 \to \alpha\}) \\
\Rightarrow \quad \Gamma \vdash \langle x_1 \mid x_2 \mid \text{typ} \rangle \Rightarrow \Theta(\{\sigma \circ \rho_1 \mid \sigma \in \Sigma\}, \{\sigma \circ \rho_2 \mid \sigma \in \Sigma\}) \quad \Sigma \neq \emptyset \quad \alpha \in \mathcal{V}_p \text{ fresh}
\end{align*}
$$

where $\text{refresh}(t)$ returns a renaming from $\text{vars}(t) \cap \mathcal{V}_p$ to fresh polymorphic variables.

For applications, an annotation of the form $\Theta(\Sigma_1, \Sigma_2)$ must be produced. In order to find some instantiations $\Sigma_1$ and $\Sigma_2$ (for $x_1$ and $x_2$ respectively) that make the application typable, the [App] rule solves the tallying instance $\text{tally}(\{t_1 \rho_1 \leq t_2 \rho_2 \to \alpha\})$. The purpose of $\rho_1$ and $\rho_2$ is to decorrelate type variables in $\Gamma(x_1)$ and in $\Gamma(x_2)$. For instance, assume we want to reconstruct the instantiations for the atom $x \times x$ with $\Gamma(x) = \beta \to \beta$. The tallying instance $\text{tally}(\{\beta \to \beta \leq \beta \to \beta \to \alpha\})$ has no solution, because the use of the same type variable $\beta$ on both sides of $\leq$ makes the types incompatible. However, $\text{tally}(\{\beta' \to \beta' \leq \beta \to \beta \to \alpha\})$ has solutions, in particular $\{\beta' \sim \beta \to \beta \vdash \alpha \to \beta \to \beta\}$. The side-condition $\Sigma \neq \emptyset$ ensures that the tallying instance has at least one solution (otherwise the annotation produced would be invalid). The rules for projections, pairs,
We have implemented the reconstruction presented in Section 4, using the CDuce \cite{CDuce} API for\footnote{\url{https://poly-dl.github.io/poly-dl/}} (where \cite{Ocsigen}, and is about 8 times slower than the native version. The web version is compiled to JavaScript using js_of_ocaml \cite{Ocsigen}, and is about 8 times slower than the native version.

For each function we report its inferred type and the time to infer it. To enhance readability we manually curated the types which, thus, may be syntactically different from (but are semantically equivalent to) the types printed by the prototype. The experiments were performed on an Intel Core i9-10900KF 3.70GHz CPU. The code was compiled natively using OCaml 4.14.1. All these examples (and more) can be easily tested with the web-based interactive prototype hosted at \url{https://poly-dl.github.io/poly-dl/}. The web version is compiled to JavaScript using js_of_ocaml \cite{Ocsigen}, and is about 8 times slower than the native version.

<table>
<thead>
<tr>
<th>Code</th>
<th>Inferred type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Falsy = False</td>
<td>(Falsy \to False) \land (Truthy \to True)</td>
<td>4.90</td>
</tr>
<tr>
<td>type Truthy = ~Falsy</td>
<td>((\alpha \land Truthy) \times 1) \to \alpha \land Truthy)\land</td>
<td>23.30</td>
</tr>
<tr>
<td>let toBoolean x = if x is Truthy then true else false</td>
<td>((Falsy \times \beta) \to \beta)</td>
<td>11.52</td>
</tr>
<tr>
<td>let lOr (x,y) = if toBoolean x then x else y</td>
<td>((\beta \to \alpha) \to (\beta \to \alpha) \land y) \to (\beta \to \alpha) \land y</td>
<td>42.44</td>
</tr>
<tr>
<td>let id x = lOr (x,x)</td>
<td>(\alpha \to \alpha</td>
<td>92.05</td>
</tr>
<tr>
<td>let fixpoint = fun f -&gt; let delta = fun x -&gt; ( fun y -&gt; ( x x y ) ) in delta</td>
<td>((\alpha \to \beta) \to (([\alpha]\to[\beta]\to)) \land (1 \to [1] \to [1])</td>
<td>172.13</td>
</tr>
<tr>
<td>let map = fixpoint map_stub</td>
<td>((\alpha \to \beta) \to ([\alpha] \to [\beta])\to ([1] \to [1]) \to [1])</td>
<td>84.30</td>
</tr>
<tr>
<td>let filter = fixpoint filter_stub (f: ('a-&gt;Any) &amp; ('b -&gt; ~True))</td>
<td>((\alpha \to \beta) \to ([\alpha]\to[\beta]\to)\to ([1] \to [1]) \to [1])</td>
<td>29.38</td>
</tr>
<tr>
<td>let rec flatten x = match x with</td>
<td>((\alpha \to [1]) \land (\beta \to \land True) \to (\alpha \land \beta)\to ([\alpha]\to[\beta]\to)\to ([1] \to [1]) \to [1]) \to [1])</td>
<td>955.15</td>
</tr>
<tr>
<td></td>
<td>([1] \to [1] \land \land True) \to (\alpha \land \beta)\to ([\alpha]\to[\beta]\to)\to ([1] \to [1]) \to [1]) \to [1])</td>
<td>955.15</td>
</tr>
</tbody>
</table>

Note: The table above is printed in the form of a natural language sentence. The natural language sentence is not a valid OCaml code snippet. The natural language sentence is intended to provide a summary of the table.

Table 1. Types inferred by the implementation (times are in ms)

and type-cases are similar and, thus, omitted.

\[
\begin{align*}
\lambda & \vdash_P \langle a \mid A \rangle \Rightarrow \varnothing & \lambda \vdash_\forall \langle a \mid \varnothing \rangle : s \quad (\forall i \in I) & \lambda, x : s \land \forall i \vdash_P \langle \kappa_i \mid K_i \rangle \Rightarrow k_i \\
\lambda & \vdash_P \langle \text{bind x} = a \in \kappa \mid \text{keep} (A, \varnothing, \{(u_i, K_i)\}_{i \in I}) \Rightarrow \text{keep} (\varnothing, \{(u_i, k_i)\}_{i \in I}) \end{align*}
\]

(\text{bind x} = a \in \kappa \mid \text{keep} (A, \varnothing, \{(u_i, K_i)\}_{i \in I}) \Rightarrow \text{keep} (\varnothing, \{(u_i, k_i)\}_{i \in I})) \Rightarrow \text{keep} (\varnothing, \{(u_i, k_i)\}_{i \in I})

(where (\ast) is I \neq \varnothing and \bigvee_{\varnothing \in \varnothing} u_i \approx \varnothing). The rule [BINDKEEP] takes as input an intermediate annotation keep (A, S, S'), with S = \varnothing, since all branches must have been fully explored by the main reconstruction algorithm. The two side-conditions (\ast) check the validity of the intermediate annotation. The rule recursively transforms the intermediate annotation A for the definition a into an annotation a for the algorithmic type system, and uses it to type a. It can then update the environment and proceed recursively on the body \kappa, for each branch in S'.

5 IMPLEMENTATION

We have implemented the reconstruction presented in Section 4, using the CDuce \cite{CDuce} API for the subtyping and the tallying algorithms. The prototype is 4500 lines of OCaml code and features several extensions such as optional type annotations, pattern matching (cf. Appendix A), records, and a more user-friendly syntax. It implements some optimisations not discussed in this paper, for instance to avoid typing redundant branches when inferring the domains of \lambda-abstractions.

We give in Table 1 the code of several functions, using a syntax similar to OCaml, where uppercase identifiers (e.g., True, Truthy) denote types and lowercase identifiers denote variables or constants. For each function we report its inferred type and the time to infer it. To enhance readability we manually curated the types which, thus, may be syntactically different from (but are semantically equivalent to) the types printed by the prototype. The experiments were performed on an Intel Core i9-10900KF 3.70GHz CPU. The code was compiled natively using OCaml 4.14.1. All these examples (and more) can be easily tested with the web-based interactive prototype hosted at \url{https://poly-dl.github.io/poly-dl/}. The web version is compiled to JavaScript using js_of_ocaml \cite{Ocsigen}, and is about 8 times slower than the native version.
Code 1 features the examples used in the introduction.

Code 2 implements Curry’s fix-point combinator in a call-by-value setting. Though it is traditionally given the type \(((β → α) → (β → α)) → (β → α)\), our prototype infers a slightly more precise type by intersecting the co-domain of the argument with \(γ\).

Code 3 shows how to use the fix-point combinator to type recursive functions. The map\_stub function implements a step of the traditional map function. The type inferred for this function has been omitted for simplicity. Then, map is obtained by applying the fixed-point combinator to map\_stub. Note that \([α∗]\) denotes a list of elements of type \(α\), and \([]\) denotes an empty list. The branch \(1 → [] \rightarrow []\) may be surprising, but it is correct since the map function does not use its first argument if the second argument is an empty list.

Code 4 shows how type annotations (cf. Appendix A.2) can be used to infer more precise types: when the filter function applied to a (precisely typed) characteristic function for the set \(α ∨ β\), the inferred type removes from the result the elements that do not satisfy the predicate.

Defining recursive functions by using a fixpoint combinator is but a stress test for our reconstruction algorithm. For recursive functions we implemented classic let rec definitions, for which the reconstruction takes the arity of the function into account. This may improve the speed of reconstruction of unannotated functions. This is the case, for instance, for the let rec version of filter for which our implementation returns a type that subsumes the one inferred by ML (it is a big intersection which, besides the ML type, enumerates some specific cases for the input, e.g., empty list, non-empty list, function always returning true etc.), but which fails with the fixpoint combinator. Code 5 shows the use of let rec and of pattern matching (cf. Appendix A.3). It defines the deep flatten function that transforms arbitrary nested lists into the list of their elements (where concat is a function of type \([α∗] → [β∗] → [(α∗)(β∗)]\), the result being the type of lists starting with \(α\) elements and ending with \(β\) ones). This function is the ultimate test for any type system: as explained by Greenberg [2019] this simple polymorphic function defies all type systems since of all existing languages, none can reconstruct a type for it and only a couple of languages can check its explicitly typed version: CDue and Haskell (the latter by resorting to complex metaprogramming constructions). Our system reconstructs a precise type for flatten as shown by the first arrow in its intersection type, which states that flatten is a function that takes a tree (i.e., a list of elements that are either trees or values different from non-empty lists) and returns the list of elements of the tree that are not lists; the other two arrows of the intersection, state that when flatten is applied to an element different from a non-empty list, or to a list that contains only other lists but no other element, then it returns the singleton list of that element or the empty list, respectively.

6 RELATED WORK

This work can be seen as a polymorphic extension of [Castagna et al. 2022b] from which it borrows some key notions, such as (i) the combination of the union elimination rule (from [Barbanera et al. 1995]) with three rules for type-cases, in order to capture the essence of occurrence typing ([Tobin-Hochstadt and Felleisen 2008]), (ii) the use of MSC forms to drive the application of the union elimination rule, and (iii) the use of annotations in the algorithmic type system. However, the introduction of polymorphic types greatly modifies the meta-theory. Besides its influence on the union elimination rule, the interplay between intersection, union elimination and instantiation suggests a different style of type annotations, to be amenable to type inference. We use external annotations while [Castagna et al. 2022b] annotates terms. Further, the presence of type variables imposes to use tallying in an inference algorithm inspired by ‘W’ by Damas and Milner [1982] and from [Castagna et al. 2015], where tallying was first introduced to type polymorphic applications. This yields a clear improvement over [Castagna et al. 2022b] which is unable to infer higher-order types for function arguments, while our algorithm is able to do so even for recursive functions.
The use of trees to annotate calculi with full-fledged intersection types is common. In the presence of explicitly-typed overloaded functions, one must be able to precisely describe how the types of nested \(\lambda\)-abstractions relate to the various “branches” of the outermost function. The work most similar to ours is [Liquori and Ronchi Della Rocca 2007], since the deductions are performed on pairs of marked term and proof term. A marked term is an untyped term where variables are marked with integers and a proof term is a tree that encodes the structure of the typing derivation and relates marks to types. Other approaches, such as [Bono et al. 2008; Ronchi Della Rocca 2002; Wells et al. 2002], duplicate the term typed with an intersection, such that each copy corresponds exactly to one member of the intersection. Lastly, the work of [Wells and Haack 2002] does not duplicate terms but rather decorate \(\lambda\)-abstractions with a richer concept of branching shape which essentially allows one to give names to the various branches of an overloaded function and to use these names in the annotations of nested \(\lambda\)-abstraction. Note that none of these works feature type reconstruction, which was our main motivation to eschew annotations within terms, since the backtracking nature of our reconstruction would imply rewriting terms over and over.

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]. Both works trade expressivity for principality. They define a lattice of types and an algebraic subtyping relation that ensures principality, but forbids the intersection of arrow types. This precludes them from expressing overloaded functions, but allows them to define a principal polymorphic type inference with unions and intersections. We justify our choice of set-theoretic types, with no type principality and a complex inference, by our aim to type dynamic languages, such as Erlang or JavaScript, where overloading plays an important role. We favour the expressivity necessary to type many idioms of these languages, and rely on user-defined annotations when necessary to compensate for the incompleteness of type inference. Lastly, both works implement some form of type simplifications (e.g., Dolan and Mycroft [2017] use automata techniques to simplify types), a problem of practical importance that we did not tackle, yet.

Ângelo and Florido [2022] provide a principal type inference for a type system with rank-2 intersection types. In their work, overloaded behaviors are expressible using intersection types, but they are limited by the rank-2 restriction. Union types are not supported, nor are equi-recursive types (actually, it does not feature a general notion of subtyping between two arbitrary types). Their inference does not require backtracking: it generates a set of constraints that are then solved using a set unification algorithm. This approach for inference has some similarities with the one by Castagna et al. [2016] improved and further developed by Petrucciani [2019] in a context with set-theoretic types, where the set unification algorithm is replaced by tallying in the presence of subtyping. However, while [Petrucciani 2019] does support intersection types with no ranking limitation, it is not able to infer intersection types for overloaded functions. Our work aims to improve this aspect, as well as providing a more precise typing of type-cases (occurrence typing).

Work by Oliveira et al. [2016] and Rioux et al. [2023] study disjoint intersection and union types. They allow expressing overloaded behaviors by a general deterministic merge operator. In our work, we do not have a general merge operator: overloaded behaviors only emerge through the use of type-case expressions (or the application of an overloaded function). Our work can be extended with pattern-matching, in which case the first matching branch is selected. This is a different approach than the one used with disjoint intersection types, where branches are disjoint and have no priority and where ambiguous programs are rejected using a notion of mergeability and distinguishability, allowing to define a general merge operator and to support nested composition, which may be useful in some contexts such as compositional programming [Zhang et al. 2021].

Finally, set-theoretic types are starting to be integrated into real-world languages, for instance by Schimpf et al. [2023] for Erlang, by Jeffrey [2022] for Luau, and by Castagna et al. [2023] for Elixir.
We believe that, in the future, our work could be used in these systems in order to benefit from a more precise typing of type-cases and pattern matching, as well as by providing an optional type inference that can be used in conjunction with explicit type annotations.

7 CONCLUSION

This work aims to provide a formal and expressive type system for dynamic languages, where type-cases can be used to give functions an overloaded behavior. It features a type inference that mixes both parametric polymorphism (for modularity) and intersection polymorphism (to capture overloaded behaviors). While we believe our work to be an important step towards a better static typing of dynamic languages, several pieces are still missing. First, the union elimination rule is only valid for pure subexpressions. To circumvent this problem, a prior analysis of side-effects seems necessary to determine which terms are pure (and therefore can be shared in the MSC-form). A challenging aspect is that this analysis should probably occur before our type inference, at a point when no type information would be available. Second, while the performance of our prototype is reasonable, it can certainly be improved by using more sophisticated implementations techniques and heuristics. Third, the interactions between code that is exported and code that is local must be better studied and understood: using intersection for local polymorphic functions and generalization for global ones, may not always be entirely satisfactory since the types of the global functions may be “polluted” by the types of the local applications, yielding less a precise reconstruction for the former. One solution can be to hoist the definition of polymorphic functions at toplevel whenever possible. Lastly, an important future work is the support of row-polymorphism: while records can be easily added to the present work, the precise typing of functions operating on records requires row-polymorphism. This is especially important for dynamic languages where records are seamlessly used to encode both objects and dictionaries. A first step in that direction may be to integrate the work by Castagna [2023b], which unifies dictionaries and records.

REFERENCES

The CDuce Compiler. CDuce https://www.cduce.org


Mariangiola Dezani. 2020. Personal communication.


Facebook. Flow. Facebook https://flow.org/


Microsoft. TypeScript. Microsoft https://www.typescriptlang.org/


A EXTENSIONS

In this appendix, we present some extensions for the source language, in particular let-bindings (not to be confused with the top-level definitions composing a program: the let-bindings presented in this section can be used anywhere in an expression and do not generalize the type of their definition) and pattern matching.

This section gives an overview of these extensions together with some explanations, but the full semantics and typing rules can be found in the next appendices.

A.1 Let Bindings

A.1.1 Declarative Type System. Let bindings can be added to the syntax of our language:

\[
\text{Expressions} \quad e \quad ::= \quad \cdots \mid \text{let } x = e \text{ in } e
\]  

(4)

with the following notion of reduction:

\[
\text{let } x = v \text{ in } e \quad \rightarrow \quad e[v/x]
\]

At first sight, we could think of adding this typing rule to the declarative type system:

\[
\frac{
\Gamma \vdash e_1 : t_1 \quad \Gamma, (x : t_1) \vdash e_2 : t_2
}{
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2
}\]

\text{[LET]}

However, this extension of the declarative type system has one issue: let-bindings can introduce aliasing, preventing in some cases the \([\lor]\) rule from applying. For instance, consider the following expression:

\[
\lambda x. \text{let } y = x \text{ in } (f \ x \in \text{Int} \ ? \ f \ y) \colon 42
\]

with \(f : \mathbb{1} \rightarrow \mathbb{1}\).

Though for any argument \(x\) this function yields an integer, it is not possible to derive for it the type \(\mathbb{1} \rightarrow \text{Int}\) using this extension of the declarative type system. Indeed, \(f \ x\) and \(f \ y\) are not syntactically equivalent and thus the \([\lor]\) rule can only decompose their types independently, losing the correlation between these two expressions.

One way to fix this issue is to remove this kind of aliasing before applying the declarative type system. For that, we can introduce an intermediate language featuring an alternative version of let-bindings:

\[
\text{Expressions} \quad e \quad ::= \quad \cdots \mid \text{let } e \text{ in } e
\]  

(5)

Let-bindings of the source language can be transformed into this alternative version using a transformation \([\cdot]\) defined as follows (the other cases are straightforward):

\[
[\text{let } x = e_1 \text{ in } e_2] = \text{let } [e_1] \text{ in } [e_2]\{[e_1]/x\}
\]

Finally, the declarative type system can be extended with this rule:

\[
\frac{
\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2
}{
\Gamma \vdash \text{let } e_1 \text{ in } e_2 : t_2
}\]

\text{[LET]}
A.1.2 Algorithmic type system. Let-bindings are added to MSC forms as a new atom construction:

\[
\text{Atomic expr } a ::= \cdots | \text{let } x \text{ in } x
\]  

(6)

The intuition is the same as for the declarative type system: we want to get rid of the aliasing caused by let-bindings, while still using bindings to factorize each subexpression. Indeed, to produce an atom for the expression \( \text{let } x = e_1 \text{ in } e_2 \) we must replace each subexpression by a binding variable, which would yield something of the form \( \text{let } x = x_1 \text{ in } x_2 \). Since the body of the let-expression is a variable, then the variable \( x \) is only an alias for \( x_1 \) and thus is undesirable. Consequently, only the other two variables are specified, which yields \( \text{let } x_1 \text{ in } x_2 \) and which explains the definition of the atom for let expressions.

For instance, the example expression \( \text{let } x = \lambda y. y \text{ in } (x, x) \) has the following canonical form:

\[
\begin{align*}
\text{bind } x_1 &= \lambda y. \text{bind } y = y \text{ in } y \text{ in } y \\
\text{bind } x_2 &= (x_1, x_1) \text{ in } x_2 \\
\text{bind } x_0 &= (\text{let } x_1 \text{ in } x_2) \text{ in } x_0
\end{align*}
\]

Note that, as explained above, the variable \( x \) is no longer present in the canonical form.

The algorithmic type system can then be extended with the following rule:

\[
\begin{array}{c}
\text{[LET-Alg]} \\
\Gamma \vdash a \quad x_1 \in \text{dom} (\Gamma)
\end{array}
\]

It is straightforward to extend the reconstruction with additional rules in order to support this new construction (c.f. appendix H).

A.2 Type constraints

A new construction \((e : \tau)\) can be added to our source language. This construction acts as a type constraint: if the expression \( e \) does not reduce to a value of type \( \tau \) (and does not diverge), then the reduction will be stuck. In a sense, it could be seen as a cast, but we will not use this terminology in order to avoid confusions with gradual typing. Actually, we only introduce this construction because it will be used later to encode more general type-cases.

We add the following construction to our source language:

\[
\text{Expressions } e ::= \cdots | (e : \tau)
\]  

(7)

with the following notion of reduction:

\[
(u : \tau) \leadsto v \quad \text{if } v \in \tau
\]

The declarative type system can trivially be extended by adding this rule:

\[
\text{[Constr]} \\
\Gamma \vdash e : \tau \quad \Gamma \vdash e : t
\]

\[
\Gamma \vdash (e : \tau) : t
\]

The same construction is added to the atoms of canonical forms:

\[
\text{Atomic expr } a ::= \cdots | x : \tau
\]  

(8)

The annotations of the algorithmic type system also need to be extended:

\[
\text{Atoms annotations } a ::= \cdots | \delta(\Sigma)
\]  

(9)

and the algorithmic type system is extended with the following rule:

\[
\text{[Constr-Alg]} \\
\Gamma \vdash a \quad \delta(\Sigma) \leq \tau
\]

\[
\Gamma(\delta(\Sigma)) : \Gamma(x)
\]

It is also straightforward to extend the reconstruction with additional rules in order to support this new construction (c.f. appendix H).
A.3 Pattern matching

Pattern matching is a fundamental feature of functional languages, and even some dynamic languages such as Python have started to implement it. In this section, we show how this feature can be added in our source language. We proceed in two steps: first, a more general typecase construct with arbitrary arity is introduced, and secondly, this construct is generalized again so that branches can be decorated with patterns instead of just types.

A.3.1 Extended typecases. We start by adding a generalized version of the typecase, that can have any number of branches:

\[ \text{Expressions } \mathcal{E} ::= \cdots | (\text{tcase } \mathcal{E} \text{ of } \tau \rightarrow \mathcal{E} | \ldots | \tau \rightarrow \mathcal{E}) \]  

with the following notion of reduction:

\[ \text{tcase } v \text{ of } \tau_1 \rightarrow \mathcal{E}_1 | \ldots | \tau_n \rightarrow \mathcal{E}_n \leadsto \mathcal{E}_k \text{ if } v : \tau_k \setminus (\bigvee_{i \in 1..k-1} \tau_i) \text{ for any } k \in 1..n \]  

In term of typing, however, we choose not to extend the type system with additional rules in order to preserve its minimality. Instead, we transform expressions with extended typecases into expressions of the source language presented in section 2, with the let-binding and type constraints extensions (A.1 and A.2). For that, we use the following transformation:

\[ \langle \text{tcase } \mathcal{E} \text{ of } \tau_1 \rightarrow \mathcal{E}_1 | \ldots | \tau_n \rightarrow \mathcal{E}_n \rangle = \begin{cases} \lambda x.\langle \mathcal{E} \rangle & \text{if } x \in \tau \setminus (\bigvee_{i \in 1..n} \tau_i) \\ \langle \mathcal{E}_1 \rangle & \text{if } \tau_1 \rightarrow \mathcal{E}_1 \\ \ldots \\ \langle \mathcal{E}_n \rangle & \text{if } \tau_n \rightarrow \mathcal{E}_n \end{cases} \]

A.3.2 Pattern matching. Now, we introduce patterns and a pattern matching construct in the source language:

\[ \text{Patterns } p ::= \tau | x | p \& p | p | (p, p) | x := c \]

\[ \text{Expressions } \mathcal{E} ::= \cdots | (\text{match } \mathcal{E} \text{ with } p \rightarrow e | \ldots | p \rightarrow e) \]  

The associated reduction rule can be found in Appendix B.

In term of typing, we proceed as before by transforming an expression with pattern matching into an expression without pattern matching (but with extended typecases and let-bindings), using the following transformation:
\( \langle e \rangle = c \)
\( \langle x \rangle = x \)
\( \langle \lambda x . e \rangle = \lambda x . \langle e \rangle \)
\( \langle \pi_i e \rangle = \pi_i \langle e \rangle \)
\( \langle (e_1, e_2) \rangle = (\langle e_1 \rangle, \langle e_2 \rangle) \)
\( \langle (e_1 : e_2) \rangle = (\langle e_1 \rangle : \langle e_2 \rangle) \)
\( \langle (e \in \tau) ? e_1 : e_2 \rangle = (\langle e \rangle \in \tau) ? \langle e_1 \rangle : \langle e_2 \rangle \)
\( \langle \text{let } x = e_1 \text{ in } e_2 \rangle = \text{let } x = \langle e_1 \rangle \text{ in } \langle e_2 \rangle \)
\( \langle (e \in \tau) \rangle = \langle e \rangle \in \tau \)
\( \langle \text{tcase } e \text{ of } \tau_1 \rightarrow e_1 | ... | \tau_n \rightarrow e_n \rangle = \text{tcase } \langle e \rangle \text{ of } \tau_1 \rightarrow \langle e_1 \rangle | ... | \tau_n \rightarrow \langle e_n \rangle \)
\( \langle \text{match } e \text{ with } p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n \rangle = \text{let } x = \langle e \rangle \text{ in } \text{tcase } x \text{ of } \tau_1 \rightarrow \langle e_1 \rangle | ... | \tau_n \rightarrow \langle e_n \rangle \)

with \( x \) fresh, and where for every \( i \in 1 \ldots m \):
\( e'_i = \text{let } x_1 = d_x(p_1,x) \text{ in } ... \text{let } x_m = d_x(p_1,x) \text{ in } \langle e_i \rangle \) for \( \{x_1, ..., x_m\} = \text{vars}(p_i) \) with

\[
\begin{align*}
    d_x(x,e) &= e \\
    d_x(x := c, e) &= c \\
    d_x((p_1, p_2), e) &= d_x(p_1, \pi_1 e) \\
    d_x(p_1 \& p_2, e) &= d_x(p_1, e) \\
    d_x(p_1 | p_2, e) &= (e \in \{ p_1 \} ) ? d_x(p_1, e) : d_x(p_2, e) \\
    d_x(p, e) &= \text{undefined} \\
\end{align*}
\]

otherwise

\[\text{if } x \in \text{vars}(p_i) \]

\[\text{if } x \in \text{vars}(p_i) \]

\[\text{otherwise} \]

\section*{B \ FULL SEMANTICS WITH EXTENSIONS}

Expressions of the source language with extensions of Appendix A are defined as follows:

\[\begin{array}{ll}
\text{Test Types} & \tau ::= b \mid 0 \rightarrow 1 | \tau \times \tau | \tau \lor \tau | \neg \tau | 0 \\
\text{Patterns} & p ::= \tau \mid x \mid p \& p \mid p \mid p \mid (p, p) \mid x := c \\
\text{Expressions} & e ::= c \mid x \mid \lambda x . e \mid ee \mid (e, e) \mid \pi_i e \mid (e \in \tau) ? e \mid \text{let } x = e \text{ in } e \mid (e \in \tau) \\
& \text{if } (\text{tcase } e \text{ of } e \rightarrow e | ... | e \rightarrow e) \mid (\text{match } e \text{ with } p \rightarrow e | ... | p \rightarrow e) \\
\text{Values} & v ::= c \mid \lambda x . e \mid (u, v)
\end{array}\]
The associated reduction rules are:

\[(\lambda x. e) v \rightarrow e[v/x] \quad (13)\]
\[\pi_1(v_1, v_2) \rightarrow v_1 \quad (14)\]
\[\pi_2(v_1, v_2) \rightarrow v_2 \quad (15)\]
\[(v \in \tau) ? e_1 : e_2 \rightarrow e_1 \quad \text{if } v \in \tau \quad (16)\]
\[(v \in \neg \tau) ? e_1 : e_2 \rightarrow e_2 \quad \text{if } v \in \neg \tau \quad (17)\]
\[\text{let } x = v \text{ in } e \rightarrow e[v/x] \quad (18)\]
\[(v \equiv \tau) \rightarrow v \quad \text{if } v \in \tau \quad (19)\]
\[\text{tcase } v \text{ of } r_1 \rightarrow e_1 | \ldots | r_n \rightarrow e_n \rightarrow e_k \quad \text{if } v : \tau_k \setminus (\bigvee_{i=1..k-1} \tau_i) \quad (20)\]
\[\text{match } v \text{ with } p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n \rightarrow e_k(v/p_k) \quad \text{if } v : (\bigvee_{i=1..n} \tau_i) \quad (21)\]

together with the context rules that implement a leftmost outermost reduction strategy, that is, \(E[e] \rightarrow E[e']\) if \(e \rightarrow e'\) where the evaluation contexts \(E[\cdot]\) are defined as follows:

**Evaluation Context**  \[E ::= [] | vE | Ee | (v,E) | (E,e) | \pi_iE | (E \notin \tau) ? e : e \]
\[| \text{let } x = E \text{ in } e | (E \equiv \tau) | (\text{tcase } E \text{ of } \ldots) \mid (\text{match } E \text{ with } \ldots)\]

Capture-avoiding substitutions are defined as follows (cases for extended typecases and pattern-matchings have been omitted for concision):

\[c(e'/x) = c \]
\[x(e'/x) = e' \]
\[y(e'/x) = y \quad x \neq y \]
\[(\lambda x. e)(e'/x) = \lambda x. e \]
\[(\lambda y. e)(e'/x) = \lambda y. e(e'/x) \quad x \neq y, y \notin \text{fv}(e') \]
\[(\lambda y. e)(e'/x) = \lambda z. (e(z/y){e'/x}) \quad x \neq y, y \in \text{fv}(e'), z \text{ fresh} \]
\[(e_1 e_2)(e'/x) = (e_1(e'/x))(e_2(e'/x)) \]
\[(e_1, e_2)(e'/x) = (e_1(e'/x), e_2(e'/x)) \]
\[(\pi_i e)(e'/x) = \pi_i(e(e'/x)) \]
\[((e_1 \in \tau) ? e_2 : e_3)(e'/x) = (e_1(e'/x) \in \tau) ? e_2(e'/x) : e_3(e'/x) \]
\[(\text{let } x = e_1 \text{ in } e_2)(e'/x) = \text{let } x = e_1(e'/x) \text{ in } e_2 \]
\[(\text{let } y = e_1 \text{ in } e_2)(e'/x) = \text{let } y = e_1(e'/x) \text{ in } e_2(e'/x) \quad x \neq y, y \notin \text{fv}(e') \]
\[(\text{let } y = e_1 \text{ in } e_2)(e'/x) = \text{let } y = e_1(e'/x) \text{ in } e_2(z/y){e'/x} \quad x \neq y, y \in \text{fv}(e'), z \text{ fresh} \]

The relation \(v \in \tau\) that determines whether a value is of a given type or not and holds true if and only if \(\text{typeof}(v) \leq \tau\), where

\[\text{typeof}(\lambda x. e) = 0 \rightarrow 1 \]
\[\text{typeof}(c) = b_c \]
\[\text{typeof}((v_1, v_2)) = \text{typeof}(v_1) \times \text{typeof}(v_2) \]
Finally, the operators used in the reduction rule for pattern matching are defined as follows:

\[
\begin{align*}
\{\tau\} &= \tau \\
\{x\} &= 1 \\
\{p_1 &\& p_2\} &= \{p_1\} \land \{p_2\} \\
\{p_1 | p_2\} &= \{p_1\} \lor \{p_2\} \\
\{(p_1, p_2)\} &= \{p_1\} \times \{p_2\} \\
\{x := c\} &= 1
\end{align*}
\]

and

\[
\begin{align*}
v/\tau &= \text{id} & \text{if } v : \tau \\
v/x &= \{v/x\} \\
v/(p_1 &\& p_2) &= \sigma_1 \cup \sigma_2 & \text{if } \sigma_1 = v/p_1 \text{ and } \sigma_2 = v/p_2 \\
v/(p_1 | p_2) &= v/p_1 & \text{if } v/p_1 \neq \text{fail} \\
v/(p_1 | p_2) &= v/p_2 & \text{if } v/p_1 = \text{fail} \\
v/(p_1, p_2) &= \sigma_1 \cup \sigma_2 & \text{if } v = (v_1, v_2), \ \sigma_1 = v_1/p_1 \text{ and } \sigma_2 = v_2/p_2 \\
v/(x := c) &= \{c/x\} \\
v/p &= \text{fail} & \text{otherwise}
\end{align*}
\]

C SUBTYPING RELATION

Subtyping is defined by giving a set-theoretic interpretation of the types of Definition 2.1 into a suitable domain \(D\). In case of polymorphic types, the domain at issue must satisfy the property of convexity [Castagna and Xu 2011]. A simple model that satisfies convexity was proposed by [Gesbert et al. 2015]. We succinctly present it in this section. The reader may refer to [Castagna 2023a, Section 3.3] for more details.

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

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

where \(c\) ranges over the set \(C\) of constants, \(L\) ranges over finite sets of type variables, and where \(\Omega\) is such that \(\Omega \notin D\).

The elements of \(D\) 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 \(\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L\), where \(\Omega\) (which is not in \(D\)) 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 \(\partial\) which ranges over \(D_{\Omega} = D \cup \{\Omega\}\) (we reserve \(d\) to range over \(D\), thus excluding \(\Omega\)). This constant \(\Omega\) is used to ensure that \(\text{1} \rightarrow \text{1}\) is not a supertype of all function types: if we used \(d\) instead of \(\partial\), then every well-typed function could be subsumed to \(\text{1} \rightarrow \text{1}\) and, therefore, every application could be given the type \(\text{1}\), independently from its argument as long as this argument is typable (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 $\alpha$ by the set of all elements that are labeled by $\alpha$, that is $\llbracket\alpha\rrbracket = \{d \mid \alpha \in \text{tags}(d)\}$ (where we define $\text{tags}(c^\downarrow) = \text{tags}(\langle d, d' \rangle^\downarrow) = \text{tags}(\langle (d_1, \partial_1), \ldots, (d_n, \partial_n) \rangle^\downarrow) = \mathcal{L}$).

We define the interpretation $\llbracket t \rrbracket$ 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 $\mathcal{B}$ denotes the function that assigns to each basic type the set of constants of that type, so that for every constant $c$ we have $c \in \mathcal{B}(b_c)$ (we use $b_c$ to denote the basic type of the constant $c$):

\[
\begin{align*}
\llbracket 0 \rrbracket &= \emptyset \\
\llbracket a \rrbracket &= \{d \mid \alpha \in \text{tags}(d)\} \\
\llbracket b \rrbracket &= \mathcal{B}(b) \\
\llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
\llbracket t_1 \land t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
\llbracket t_1 \lor t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
\llbracket t_1 \rightarrow t_2 \rrbracket &= \{R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}) \mid \forall (d, \partial) \in R. \ d \in \llbracket t_1 \rrbracket \implies \partial \in \llbracket t_2 \rrbracket\}
\end{align*}
\]

We cannot take the equations above directly as an inductive definition of $\llbracket \rrbracket$ because types are not defined inductively but coinductively. Notice however that the contractivity condition of Definition 2.1 ensures that the binary relation $\triangleright \subseteq \text{Types} \times \text{Types}$ defined by $t_1 \lor t_2 \triangleright t_1, t_1 \land t_2 \triangleright t_1, \neg t \triangleright t$ is Noetherian. This gives an induction principle\(^7\) on Types that we use combined with structural induction on $\mathcal{D}$ to give the following definition, which validates these equalities.

**Definition C.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 $t \in \text{Types}$, by induction on the pair $(d, t)$ ordered lexicographically. The predicate is defined as follows:

\[
\begin{align*}
(c : b) &= c \in \mathcal{B}(b) \\
(d : a) &= \alpha \in \text{tags}(d) \\
\langle (d_1, d_2) : t_1 \times t_2 \rangle &= \langle d_1 : t_1 \rangle \text{ and } \langle d_2 : t_2 \rangle \\
\langle \{d_1, \partial_1\}, \ldots, \{d_n, \partial_n\} \rangle : t_1 \rightarrow t_2 &= \forall i \in \llbracket 1 \ldots n \rrbracket. \text{ if } \langle d_i : t_1 \rangle \text{ then } \langle \partial_i : t_2 \rangle \\
(d : t_1 \lor t_2) &= (d : t_1) \text{ or } (d : t_2) \\
(d : \neg t) &= \text{ not } (d : t) \\
(\partial : t) &= \text{ false otherwise}
\end{align*}
\]

We define the set-theoretic interpretation $\llbracket \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$ as $\llbracket t \rrbracket = \{d \in \mathcal{D} \mid (d : t)\}$.

Finally, we define the subtyping preorder and its associated equivalence relation as follows.

**Definition C.3 (Subtyping relation).** We define the subtyping relation $\leq$ and the subtyping equivalence relation $\equiv$ as $t_1 \leq t_2 \overset{\text{def}}{\iff} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$ and $t_1 \equiv t_2 \overset{\text{def}}{\iff} (t_1 \leq t_2)$ and $(t_2 \leq t_1)$.

**D DECLARATIVE TYPE SYSTEM WITH EXTENSIONS**

The declarative type system extended with the extensions of Appendix A uses expressions produced by the following grammar:

**Expressions**

\[
e \ ::= \ c \mid x \mid \lambda x.e \mid ee \mid (e, e) \mid \pi_i e \mid (e \in \tau) \mid \text{let } e \in e \mid (e \in \tau)
\]

Note that extended typecases and pattern matching are absent because they are encoded using let-bindings and type constraints before typing. Similarly, we use the construction $\text{let } e \in e$ for let-bindings instead of the initial construction $\text{let } x = e \in e$ in order to avoid aliasing. You should refer to Section A.1 for more details on this transformation.

\(^7\)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 basic types are the base cases for the induction.
The deduction rules for the declarative type system are:

\[
\begin{align*}
&\text{[Const]} \quad \Gamma ⊢ c : b_c \\
&\text{[Ax]} \quad \Gamma ⊢ x : \Gamma(x)
\end{align*}
\]

\[
\begin{align*}
&\text{[→I]} \quad \Gamma, x : u ⊢ e : t \\&\text{[→E]} \quad \Gamma ⊢ e_1 : t_1 \rightarrow t_2 \quad \Gamma ⊢ e_2 : t_1 \\
&\text{[×I]} \quad \Gamma ⊢ e_1 : t_1 \quad \Gamma ⊢ e_2 : t_2 \\
&\text{[×E1]} \quad \Gamma ⊢ e : t_1 \times t_2 \\
&\text{[×E2]} \quad \Gamma ⊢ e : t_1 \times t_2
\end{align*}
\]

\[
\begin{align*}
&\text{[0]} \quad \Gamma ⊢ e : 0 \\
&\text{[e1]} \quad \Gamma ⊢ e : \tau \quad \Gamma ⊢ e_1 : t_1 \\
&\text{[e2]} \quad \Gamma ⊢ e : \tau \quad \Gamma ⊢ e_2 : t_2 \\
&\text{[v]} \quad \Gamma ⊢ e' : s \quad \Gamma, x : s \land u ⊢ e : t \\
&\text{[∧]} \quad \Gamma ⊢ e : t_1 \quad \Gamma ⊢ e : t_2 \\
&\text{[Inst]} \quad \Gamma ⊢ e : t \quad \Gamma ⊢ e : t \sigma \\
&\text{[≤]} \quad \Gamma ⊢ e : t \quad \Gamma ⊢ e : t' \quad t ≤ t'
\end{align*}
\]

with these additional rules for the extensions of Appendix A (let-bindings and type constraints):

\[
\begin{align*}
&\text{[Let]} \quad \Gamma ⊢ e_1 : t_1 \quad \Gamma ⊢ e_2 : t_2 \\
&\text{[Constr]} \quad \Gamma ⊢ e \in \tau \quad \Gamma ⊢ e : t
\end{align*}
\]

E  COMPUTATION OF MSC-FORMS

E.1 From canonical forms to source language expressions

We recall the grammar for canonical forms, with the extensions presented in Appendix A:

Atomic expressions \( a ::= c \mid x \mid \lambda x.x \mid (x, x) \mid xx \mid \pi_1 x \mid (x \in \tau) \mid x : x \mid \text{let } x \text{ in } x \mid x \in \tau \)

Canonical Forms \( \kappa ::= x \mid \text{bind } x = a \in \kappa \)

Any canonical form can be transformed into an expression of the source language using the unwinding operator \( [\cdot] \) defined as follows:

\[
\begin{align*}
[c] &= c \\
[x] &= x \\
[\lambda x.x] &= \lambda x.[x] \\
[x_1 x_2] &= x_1 x_2 \\
[(x_1, x_2)] &= (x_1, x_2) \\
[\pi_1 x] &= \pi_1 x \\
[(x \in \tau) \ ? x_1 : x_2] &= (x \in \tau) \ ? x_1 : x_2 \\
[\text{let } x_1 \text{ in } x_2] &= \text{let } x = x_1 \text{ in } x_2\{x/x_1\} \text{ with } x \text{ fresh} \\
[x \in \tau] &= x \in \tau \\
[\text{bind } x = a \in \kappa] &= [\kappa][\{a\}/x] \\
[x] &= x
\end{align*}
\]
E.2 From source language expressions to canonical forms

Any expression $e$ of the source language can be transformed into a canonical form whose unwinding is $e$. Let $\Delta$ denote a possibly empty list of mappings from binding variables to atoms. We note these lists extensionally by separating elements by a semicolon, that is, $x_1\mapsto a_1;\ldots;x_n\mapsto a_n$ and use $e$ to denote the empty list. We define an operation $\text{term}(\Delta, x)$ which takes a list of mappings $\Delta$ and a binding variable $x$ and constructs the canonical form whose bindings are those listed in $\Delta$ and whose body is $x$, that is:

$$\text{term}(e, x) \overset{\text{def}}{=} x$$

$$\text{term}(y\mapsto a; \Delta, x) \overset{\text{def}}{=} \text{bind } y = a \text{ in } \text{term}(\Delta, x)$$

We can now define the function $[e]$ that transforms an expression $e$ into a pair $(\Delta, x)$ formed by a list of mappings $\Delta$ and a binding variable $x$ that will be bound to the atom representing $e$. The definition is as follows, where $x_0$ is a fresh binding variable.

$$[e] = ((x_0\mapsto e), x_0)$$

$$[x] = ((x_0\mapsto x), x_0)$$

$$[\lambda x.e] = ((x_0\mapsto \lambda x.\text{term}[e]), x_0)$$

$$[\pi_1 e] = (\text{where } (\Delta, x) = [e])$$

$$[e_1 e_2] = ((\Delta_1; \Delta_2; x_0\mapsto x_1 x_2), x_0) \text{ where } (\Delta_1, x_1) = [e_1], \ (\Delta_2, x_2) = [e_2]$$

$$[\text{let } x = e_1 \text{ in } e_2] = ((\Delta_1; \Delta_2; x_0\mapsto (x \in \tau) \ ? x_1 : x_2), x_0) \text{ where } (\Delta, x) = [e], \ (\Delta_1, x_1) = [e_1], \ (\Delta_2, x_2) = [e_2]$$

$$[\epsilon \in \tau] = (\Delta; x_0\mapsto x \in \tau, x_0) \text{ where } (\Delta, x) = [e]$$

It is easy to see that, for any term of the source language $e$, $\lfloor \text{term}(\lfloor e \rfloor) \rfloor = e$.

E.3 From canonical forms to a MSC form

It is easy to transform a canonical form into a MSC-form that has the same unwinding. This can be done by applying the rewriting rules below, that are confluent and normalizing.

$$\text{bind } x_1 = a_1 \text{ in } \kappa \rightarrow \text{bind } x_1 = a_1 \text{ in } \kappa\{x_1/x_2\} \quad a_1 \equiv_{\kappa} a_2 \quad (22)$$

$$\text{bind } x = a \text{ in } \kappa \rightarrow \kappa \quad x \not\in \text{fv}(\kappa) \quad (23)$$

$$\text{bind } x = \lambda y. (\text{bound in } \kappa_0) \rightarrow \text{bind } z = a \text{ in } \kappa \text{ in } \kappa \quad y \not\in \text{fv}(a), z \not\in \text{fv}(\kappa) \quad (24)$$

$$\kappa_1 \rightarrow \kappa_2 \quad \exists \kappa_1'. \kappa_1 \equiv_{\kappa} \kappa_1' \rightarrow \kappa_2 \quad (25)$$

Rule (22) implements the maximal sharing: if two variables bind atoms with the same unwinding (modulo $\alpha$-conversion), then the variables are unified. Rule (23) removes useless bindings. Rule (24) extrudes bindings from abstractions of variables that do not occur in the argument of the binding. Rule (25) applies the previous rule modulo the canonical equivalence: in practice it applies the swap
of binding defined in Definition 3.1 as many times as it is needed to apply one of the other rules. As customary, these rules can be applied under any context.

The transformation above transforms every canonical form into an MSC-form that has the same unwinding. It thus allows to compute $\text{MSC}(\epsilon)$ for any expression $\epsilon$ of the source language.

**F TYPE OPERATORS**

The algorithmic type system presented in this work use the following type-operators:

$$\text{dom}(t) = \max\{u \mid t \leq u \rightarrow 1\}$$

$$t \circ s = \min\{u \mid t \leq s \rightarrow u\}$$

$$\pi_1(t) = \min\{u \mid t \leq u \times 1\}$$

$$\pi_2(t) = \min\{u \mid t \leq 1 \times u\}$$

In words, $t \circ s$ is the best (i.e., smallest wrt $\leq$) type we can deduce for the application of a function of type $t$ to an argument of type $s$. Projection and domain are standard. All these operators can be effectively computed as shown below (see Castagna et al. [2022a]; Frisch et al. [2008] for details and proofs).

For $t \overset{\text{def}}{=} \bigwedge_{i \in I} \left( \bigwedge_{p' \in P_i} \alpha_{p'} \land \bigwedge_{n' \in N_i} -\alpha'_{n'} \land \bigwedge_{p \in P_i} (s_p \rightarrow t_p) \land \bigwedge_{n \in N_i} -(s'_n \rightarrow t'_n) \right)$, the first two operators are computed by:

$$\text{dom}(t) = \bigwedge_{i \in I} \bigvee_{p \in P_i} s_p$$

$$t \circ s = \bigvee_{i \in I} \left( \bigvee_{Q \subseteq P_i} \left( \bigvee_{s \notin \bigcup_{q \in Q} s_q} \left( \bigwedge_{p \in P_i \setminus Q} t_p \right) \right) \right) \quad \text{(for } s \leq \text{dom}(t))$$

For $t \overset{\text{def}}{=} \bigwedge_{i \in I} \left( \bigwedge_{p' \in P_i} \alpha_{p'} \land \bigwedge_{n' \in N_i} -\alpha'_{n'} \land \bigwedge_{p \in P_i} (s_p, t_p) \land \bigwedge_{n \in N_i} -(s'_n, t'_n) \right)$ the last two operators are computed by

$$\pi_1(t) = \bigvee_{i \in I} \bigvee_{N' \subseteq N_i} \left( \bigwedge_{p \in P_i} s_p \land \bigwedge_{n \in N'} -s'_n \right)$$

$$\pi_2(t) = \bigvee_{i \in I} \bigvee_{N' \subseteq N_i} \left( \bigwedge_{p \in P_i} t_p \land \bigwedge_{n \in N'} -t'_n \right)$$

**G ALGORITHMIC TYPE SYSTEM**

**Atom annots**

$$\emptyset ::= \emptyset \mid \lambda(u,k) \mid (\rho,\rho) \mid \emptyset(\Sigma,\Sigma) \mid \pi(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma) \mid \emptyset(\Sigma)$$

**Form annots**

$$\emptyset ::= \rho \mid \text{keep}(\emptyset,\{(u,k),\ldots,(u,k)\}) \mid \text{skip} k \mid \emptyset(\{(k,\ldots,k)\})$$

The algorithmic type system is defined by the following deduction rules:
To extend the system to type the extensions presented in Appendix A the following rules must be added:

\[
\begin{align*}
\text{[Let-Alg]} & : \quad \Gamma \vdash \text{let } x_1 \text{ in } x_2 : \varnothing : \Gamma(x_2) & x_1 \in \text{dom}(\Gamma) \\
\text{[Constr-Alg]} & : \quad \Gamma \vdash \text{let } \delta : \varnothing : \varnothing \vdash \Gamma(x) : \Gamma(x) \Sigma \leq \tau
\end{align*}
\]
\section*{H FULL RECONSTRUCTION SYSTEM}

\subsection*{H.1 Monomorphic reconstruction}

\begin{description}
\item[Split annotations] \( S ::= \{ (u, \mathcal{K}), \ldots, (u, \mathcal{K}) \} \)
\item[Atoms intermediate annot.] \( \mathcal{A} ::= \text{infer} | \text{untyp} | \text{typ} | \wedge (\{\mathcal{A}, \ldots, \mathcal{A}\}, \{\mathcal{A}, \ldots, \mathcal{A}\}) \)
\item[Forms intermediate annot.] \( \mathcal{K} ::= \text{infer} | \text{untyp} | \text{typ} | \wedge (\{\mathcal{K}, \ldots, \mathcal{K}\}, \{\mathcal{K}, \ldots, \mathcal{K}\}) \)
\end{description}

\subsection*{H.1.1 Deduction rules for \( \vdash_R \) judgements.}

\begin{align*}
\text{[Ok]} & \quad \Gamma \vdash_R \langle \eta \mid \text{typ} \rangle \Rightarrow \text{Ok(\text{typ})} \\
\text{[Fail]} & \quad \Gamma \vdash_R \langle \eta \mid \text{untyp} \rangle \Rightarrow \text{Fail} \\
\text{[Const]} & \quad \Gamma \vdash_R \langle c \mid \text{infer} \rangle \Rightarrow \text{Ok(\text{typ})} \\
\text{[AxOk]} & \quad \Gamma \vdash_R \langle x \mid \text{infer} \rangle \Rightarrow \text{Ok(\text{typ})} \\
\text{[AxFail]} & \quad \Gamma \vdash_R \langle x \mid \text{infer} \rangle \Rightarrow \text{Fail} \\
\text{[PairVar]} & \quad \Gamma \vdash_R \langle (x_1, x_2) \mid \text{infer} \rangle \Rightarrow \text{Var}(x_i, \text{infer}, \text{untyp}) \\
\text{[PairOk]} & \quad \{x_1, x_2\} \subseteq \text{dom}(\Gamma) \quad \Gamma \vdash_R \langle (x_1, x_2) \mid \text{infer} \rangle \Rightarrow \text{Ok(\text{typ})} \\
\text{[ProjVar]} & \quad \Gamma \vdash_R \langle \pi_i x \mid \text{infer} \rangle \Rightarrow \text{Var}(x, \text{infer}, \text{untyp}) \\
\text{[ProjInfer]} & \quad \Psi = \text{tally_infer}(\{\Gamma(x) \leq \alpha \times \beta\}) \quad \alpha, \beta \in \mathcal{V}_p \text{ fresh} \quad \Gamma \vdash_R \langle \pi_i x \mid \text{infer} \rangle \Rightarrow \text{Subst}(\Psi, \text{typ}, \text{untyp}) \\
\text{[AppVar]} & \quad \Gamma \vdash_R \langle x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Var}(x_i, \text{infer}, \text{untyp}) \\
\text{[AppInfer]} & \quad \Psi = \text{tally_infer}(\{\Gamma(x_1) \leq \Gamma(x_2) \to \alpha\}) \quad \alpha \in \mathcal{V}_p \text{ fresh} \quad \Gamma \vdash_R \langle x_1 x_2 \mid \text{infer} \rangle \Rightarrow \text{Subst}(\Psi, \text{typ}, \text{untyp}) \\
\text{[CaseVar]} & \quad \Gamma \vdash_R \langle (x \in \tau) \mid \text{infer} \rangle \Rightarrow \text{Var}(x, \text{infer}, \text{untyp}) \\
\text{[CaseSplit]} & \quad \Gamma(x) \not\leq \tau \quad \Gamma(x) \not\leq \neg \tau \quad \Gamma \vdash_R \langle (x \in \tau) \mid \text{infer} \rangle \Rightarrow \text{Split}(\{(x : \tau)\}, \text{infer}, \text{infer})
\end{align*}
\[\Gamma(x) = \emptyset\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid \text{infer}) \Rightarrow \text{Ok}(\text{typ})\]

\[\Gamma(x) \leq \tau \quad \Psi = \text{tally}_{\text{infer}}(\{\Gamma(x) \leq \emptyset\})\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid \text{infer}) \Rightarrow \text{Subst}(\Psi, \text{typ}, e_1)\]

\[\Gamma(x) \leq \neg \tau \quad \Psi = \text{tally}_{\text{infer}}(\{\Gamma(x) \leq \emptyset\})\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid \text{infer}) \Rightarrow \text{Subst}(\Psi, \text{typ}, e_2)\]

\[x_i \notin \text{dom}(\Gamma)\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid e_i) \Rightarrow \text{Var}(x_i, \text{typ}, \text{untyp})\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid e_i) \Rightarrow \text{Ok}(\text{typ})\]

\[\Gamma \vdash_R (\lambda x. \lambda(a, \text{infer})) \Rightarrow R\]

\[\Gamma \vdash_R (\lambda x. \text{infer}) \Rightarrow \lambda \in \mathcal{V}_M \text{ fresh}\]

\[\Gamma \vdash_R (\lambda x. \text{infer}) \Rightarrow \text{Fail}\]

\[\Gamma, x : u \vdash_R^* (x \mid \mathcal{K}) \Rightarrow R\]

\[\Gamma \vdash_R (\lambda x. \mathcal{K}) \Rightarrow \text{map}(X \mapsto \lambda(u, X), R)\]

with \(\text{map}(X \mapsto f(X), R)\) an auxiliary function that applies \(f\) to each intermediate annotation in \(R\):

\[\text{map}(X \mapsto f(X), \text{Ok}(\mathcal{H})) \overset{\text{def}}{=} \text{Ok}(f(\mathcal{H}))\]

\[\text{map}(X \mapsto f(X), \text{Fail}) \overset{\text{def}}{=} \text{Fail}\]

\[\text{map}(X \mapsto f(X), \text{Split}(\Gamma, \mathcal{H}_1, \mathcal{H}_2)) \overset{\text{def}}{=} \text{Split}(\Gamma, f(\mathcal{H}_1), f(\mathcal{H}_2))\]

\[\text{map}(X \mapsto f(X), \text{Subst}(\Psi, \mathcal{H}_1, \mathcal{H}_2)) \overset{\text{def}}{=} \text{Subst}(\Psi, f(\mathcal{H}_1), f(\mathcal{H}_2))\]

\[\text{map}(X \mapsto f(X), \text{Var}(x, \mathcal{H}_1, \mathcal{H}_2)) \overset{\text{def}}{=} \text{Var}(x, f(\mathcal{H}_1), f(\mathcal{H}_2))\]

\[x \notin \text{dom}(\Gamma)\]

\[\Gamma \vdash_R (x \mid \text{infer}) \Rightarrow \text{Var}(x, \text{infer}, \text{untyp})\]

\[\Gamma \vdash_R (x \mid \text{infer}) \Rightarrow \text{Ok}(\text{typ})\]

\[\Gamma \vdash_R (\text{bind} x = a \mid \text{try-skip}(\text{infer})) \Rightarrow R\]

\[\Gamma \vdash_R (\text{bind} x = a \mid \text{infer}) \Rightarrow R\]

\[\Gamma(x) = \emptyset\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid \text{infer}) \Rightarrow \text{Ok}(\text{typ})\]

\[\Gamma(x) \leq \tau \quad \Psi = \text{tally}_{\text{infer}}(\{\Gamma(x) \leq \emptyset\})\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid \text{infer}) \Rightarrow \text{Subst}(\Psi, \text{typ}, e_1)\]

\[\Gamma(x) \leq \neg \tau \quad \Psi = \text{tally}_{\text{infer}}(\{\Gamma(x) \leq \emptyset\})\]

\[\Gamma \vdash_R ((x \in \tau) \mathbin{?} x_1 : x_2 \mid \text{infer}) \Rightarrow \text{Subst}(\Psi, \text{typ}, e_2)\]
where $\mathcal{K}' = \text{propagate} (\mathcal{A}, \Gamma_1 \cup \Gamma_2, \{(u \land \Gamma'(x), \mathcal{K}_1), (u \land \Gamma'(x), \mathcal{K}_2)\} \cup S, S')$

$\mathcal{K}'_2 = \text{keep} (\mathcal{A}, \{(u, \mathcal{K}_2)\} \cup S, S')$
where: £ \text{compatible$(\Gamma,\Gamma')$}

\[ \Gamma \vdash R (\text{bind } x = a \in \kappa | \text{propagate } (\mathcal{A}, \Gamma, S, S')) \Rightarrow \text{Split} (\Gamma'', \mathcal{K}, \mathcal{K'}) \]

where:

- \text{compatible($\Gamma,\Gamma'$) } \Leftrightarrow (\text{dom($\Gamma'$) } \subseteq \text{dom($\Gamma$)}) \land (\forall x \in \text{dom($\Gamma'$)}). (\Gamma(x) \land \Gamma'(x) \neq 0) \lor (\Gamma(x) \land \Gamma'(x) = 0))
- $\mathcal{K'}$ = propagate $(\mathcal{A}, \Gamma \setminus \{\Gamma'\}, S, S')$
- $\Gamma'' = \{(x : u) \in \Gamma' | \Gamma(x) \not\in u\}$

\[ \Gamma \vdash R (\text{bind } x = a \in \kappa | \text{propagate } (\mathcal{A}, \Gamma, S, S')) \Rightarrow \text{R} \]

\[ \Gamma \vdash R (\text{bind } x = a \in \kappa | \text{keep } (\mathcal{A}, \Gamma, S, S')) \Rightarrow \text{R} \]

H.1.2 Deduction rules for $\Gamma \vdash$ judgements.

\[ \Gamma \vdash R (\eta | \mathcal{H}) \Rightarrow \text{Split} (\Gamma', \mathcal{H}_1, \mathcal{H}_2) \]

\[ \Gamma \vdash R (\eta | \mathcal{H}_1) \Rightarrow R' \]

\[ \Gamma = \emptyset \]

\[ \Gamma \vdash R (\eta | \mathcal{H}) \Rightarrow \text{Subst} \{(\psi_i)_{i \in I}, \mathcal{H}_1, \mathcal{H}_2\} \]

\[ \Gamma \vdash R (\eta | \text{fresh($\mathcal{H}_1\psi_i$) \cup \{\mathcal{H}_2, \emptyset\}}) \Rightarrow R' \]

\[ \forall i \in I. \psi_i \not\in \Gamma \]

with $\mathcal{H}_\psi$ denoting the intermediate annotation $\mathcal{H}$ in which the substitution $\psi$ has been applied recursively to every type (in lambdas and binding annotations), and fresh($\mathcal{H}$) denotes $\mathcal{H}$ where all the monomorphic type variables not in $\Gamma$ have been substituted by fresh ones (in order to decorrelate the different branches).

The following rules can be added to support the extensions presented in Appendix A:
\[\{x_1, x_2\} \subseteq \text{dom}(\Gamma)\]

\[\Gamma \vdash \mathcal{R} \langle \text{let } x_1 \text{ in } x_2 | \text{infer} \rangle \Rightarrow \text{Var} (x_1, \text{infer}, \text{untyp})\]

[LetOK]

\[\Gamma \vdash \mathcal{R} \langle \text{let } x_1 \text{ in } x_2 | \text{infer} \rangle \Rightarrow \text{0k} (\text{typ})\]

[LetVar]

\[x_i \not\in \text{dom}(\Gamma)\]

\[\Gamma \vdash \mathcal{R} \langle \text{let } x_1 \text{ in } x_2 | \text{infer} \rangle \Rightarrow \text{Var} (x_i, \text{infer}, \text{untyp})\]

[ConstrInfer]

\[\Sigma = \text{tally}_{\text{infer}} (\{\Gamma(x) \leq \tau\})\]

\[\Gamma \vdash \mathcal{R} \langle x \not\in \tau | \text{infer} \rangle \Rightarrow \text{Subst} (\Sigma, \text{typ}, \text{untyp})\]

[ConstrVar]

\[\Gamma \vdash \mathcal{R} \langle x \not\in \tau | \text{infer} \rangle \Rightarrow \text{Var} (x, \text{infer}, \text{untyp})\]

H.2 Polymorphic reconstruction

In the following, refresh(t) denotes a renaming from vars(t) \cap V_P to fresh polymorphic variables.

[Const]

\[\Gamma \vdash_{\mathcal{P}} \langle c | \text{typ} \rangle \Rightarrow \emptyset\]

[Ax]

\[\Gamma \vdash_{\mathcal{P}} \langle x | \text{typ} \rangle \Rightarrow \emptyset\]

\[x \in \text{dom}(\Gamma)\]

[Pair]

\[\rho_1 = \text{refresh}(\Gamma(x_1)) \quad \rho_2 = \text{refresh}(\Gamma(x_2))\]

\[\Gamma \vdash_{\mathcal{P}} \langle (x_1, x_2) | \text{typ} \rangle \Rightarrow (\rho_1, \rho_2)\]

[Proj]

\[\Sigma = \text{tally}_{\text{infer}} (\{\Gamma(x) \leq \alpha \times \beta\})\]

\[\Gamma \vdash_{\mathcal{P}} \langle \pi_i x | \text{typ} \rangle \Rightarrow \pi(\Sigma)\]

\[\Sigma \neq \emptyset\]

\[\alpha, \beta \in V_P \text{ fresh}\]

\[t_1 = \Gamma(x_1) \quad t_2 = \Gamma(x_2) \quad \rho_1 = \text{refresh}(t_1) \quad \rho_2 = \text{refresh}(t_2)\]

\[\Sigma = \text{tally}(\{(t_1 \rho_1 \leq t_2 \rho_2 \rightarrow \alpha)\})\]

\[\Gamma \vdash_{\mathcal{P}} \langle x_1 x_2 | \text{typ} \rangle \Rightarrow \circ (\{\sigma \circ \rho_1 | \sigma \in \Sigma\}, \{\sigma \circ \rho_2 | \sigma \in \Sigma\})\]

\[\Sigma \neq \emptyset\]

\[\alpha \in V_P \text{ fresh}\]

[Case0]

\[\sigma \in \text{tally}(\{\Gamma(x) \leq \emptyset\})\]

\[\Gamma \vdash_{\mathcal{P}} \langle (x \in \tau) ? x_1 : x_2 | \text{typ} \rangle \Rightarrow \emptyset (\{\sigma\})\]

[Case1]

\[\sigma \in \text{tally}(\{\Gamma(x) \leq \tau\})\]

\[\Gamma \vdash_{\mathcal{P}} \langle (x \in \tau) ? x_1 : x_2 | \text{typ} \rangle \Rightarrow \epsilon_1 (\{\sigma\})\]

\[x_1 \in \text{dom}(\Gamma)\]

[Case2]

\[\sigma \in \text{tally}(\{\Gamma(x) \leq \tau\})\]

\[\Gamma \vdash_{\mathcal{P}} \langle (x \in \tau) ? x_1 : x_2 | \text{typ} \rangle \Rightarrow \epsilon_2 (\{\sigma\})\]

\[x_2 \in \text{dom}(\Gamma)\]

[Case2]

\[\Gamma, x : u \vdash_{\mathcal{P}} \langle \kappa | \mathcal{K} \rangle \Rightarrow \emptyset\]

\[\Gamma \vdash_{\mathcal{P}} \langle \lambda x.\kappa | \lambda (u.\mathcal{K}) \rangle \Rightarrow \lambda (u, \emptyset)\]

[Lambda]
The split propagation system defined in this section tries to deal with the following problem: given a current environment $\Gamma$, an atom $a$ and a type $t$, what additional assumptions can I make on $\Gamma$ in order to ensure that $a$ has type $t$? It is used by the main reconstruction system in order to propagate splits made by bindings.

We note $[t]$ the instance of $t$ obtained by applying the following substitutions:

- Any polymorphic variable in $t$ only appearing in covariant positions is substituted by $\underline{1}$,
- Any polymorphic variable in $t$ only appearing in contravariant positions is substituted by $\underline{0}$.

H.3 Split propagation

The following rules can be added to support the extensions presented in Appendix A:

**[VAR]**  \[ \rho = \text{refresh}(\Gamma(x)) \]

\[ \Gamma \vdash_P (x : \text{typ}) : \rho \]

**[BINDSKIP]**  \[ \Gamma \vdash_P (\text{bind } x = a \in \kappa \mid \text{skip } (K)) \Rightarrow \text{skip } \kappa \]

\[ x \notin \text{dom}(\Gamma) \]

**[BINDKEEP]**  \[ \Gamma \vdash_P (a \mid \mathcal{A}) \Rightarrow a \] \[ \Gamma \vdash_\pi [a \mid a] : s \quad (\forall i \in I) \]

\[ \Gamma, x : s \cup u, \vdash_P (\kappa \mid \mathcal{K}) \Rightarrow \kappa_i \]

where (*) is $I \neq \emptyset$ and $\bigvee_{i \in I} u_i = 1$.

**[INTER]**  \[ (\forall i \in I) \quad \Gamma \vdash_P (\eta \mid \mathcal{H}_i) \Rightarrow \eta_i \]

\[ \Gamma \vdash_P (\eta \mid \wedge (\emptyset, \{\mathcal{H}_i\}_{i \in I})) \Rightarrow \wedge (\{\eta_i\}_{i \in I}) \]

$I \neq \emptyset$

The following rules can be added to support the extensions presented in Appendix A:

**[LET]**  \[ \Gamma \vdash_P (\text{let } x_1 \text{ in } x_2 \mid \text{typ}) \Rightarrow \emptyset \]

**[CONSTR]**  \[ \Sigma = \text{tally}(\{\Gamma(x) \leq \tau\}) \]

\[ \Gamma \vdash_P (x \vdash \tau \mid \text{typ}) \Rightarrow \Downarrow(s(\Sigma)) \]

\[ \Sigma = \text{tally}(\{\Gamma(x) \leq \tau\}) \]

\[ \Gamma \vdash_P (x \vdash \tau \mid \text{typ}) \Rightarrow \Downarrow(s(\Sigma)) \]

H.3 Split propagation

The split propagation system defined in this section tries to deal with the following problem: given a current environment $\Gamma$, an atom $a$ and a type $t$, what additional assumptions can I make on $\Gamma$ in order to ensure that $a$ has type $t$? It is used by the main reconstruction system in order to propagate splits made by bindings.

We note $[t]$ the instance of $t$ obtained by applying the following substitutions:

- Any polymorphic variable in $t$ only appearing in covariant positions is substituted by $\underline{1}$,
- Any polymorphic variable in $t$ only appearing in contravariant positions is substituted by $\underline{0}$.

**[CONST1]**  \[ b_c \leq u \]

\[ \Gamma \vdash_E (c : u) \Rightarrow \{\emptyset\} \]

**[CONST2]**  \[ \Gamma \vdash_E (c : u) \Rightarrow \{\emptyset\} \]

**[AX1]**  \[ \Gamma(x) \leq u \]

\[ \Gamma \vdash_E (x : u) \Rightarrow \{\emptyset\} \]

**[AX2]**  \[ \Gamma \vdash_E (x : u) \Rightarrow \{\emptyset\} \]

**[PROJ1]**  \[ \Gamma \vdash_E (\pi_1 x : u) \Rightarrow \{\{x : u \times \underline{1}\}\} \]

**[PROJ2]**  \[ \Gamma \vdash_E (\pi_2 x : u) \Rightarrow \{\{x : \underline{1} \times u\}\} \]

**[PAIR]**  \[ u \wedge (\underline{1} \times \underline{1}) \overset{\text{DHF}}{=} \bigvee_{i \in I} (u_i \times v_i) \]

\[ \Gamma \vdash_E ((x_1, x_2) : u) \Rightarrow \{\{x_1 : u_i\} \wedge \{x_2 : v_i\} \mid i \in I\} \]

**[CASE]**  \[ \Gamma \vdash_E ((x : \tau) \wedge x_1 \times x_2 : u) \Rightarrow \{\{x : \tau, x_1 : u\}, \{x : \neg \tau, x_2 : u\}\} \]

**[CASE]**  \[ \{\sigma_i\}_{i \in I} = \text{tally}(\{\Gamma(x_1) \leq \alpha \to u\}) \]

\[ \Gamma = \{\{x_2 : [\alpha \sigma_i]\} \mid i \in I\} \]

$\alpha$ fresh

\[ \alpha \text{ fresh } \]

**[LAMBDA]**  \[ \Gamma \vdash_E (\lambda x. \kappa : u) \Rightarrow \{\} \]
The following rules can be added to support the extensions presented in Appendix A:

\[
\begin{align*}
\text{[LET]} & \quad \Gamma \vdash E \ (\text{let} \ x_1 \ \text{in} \ x_2 : u) \Rightarrow \{x_2 : u\} \\
\text{[CONSTR]} & \quad \Gamma \vdash E \ (x : \tau : u) \Rightarrow \{x : u\}
\end{align*}
\]

1 PROOFS

The proofs are for the source language presented in section 2 without extension:

Expressions \[ e ::= c \mid x \mid \lambda x. e \mid ee \mid (e, e) \mid \pi_i e \mid (e \in \tau) ? e : e \]

We recall some notations relative to substitutions:

- \( \rho \) ranges over renamings of polymorphic variables, that is, injective substitutions from \( \mathcal{V}_P \) to \( \mathcal{V}_P \)
- \( \sigma \) ranges over substitutions from polymorphic type variables \( \mathcal{V}_P \) to types
- \( \Sigma \) ranges over sets of substitutions from polymorphic type variables \( \mathcal{V}_P \) to types
- \( \psi \) ranges over substitutions from monomorphic type variables \( \mathcal{V}_M \) to monomorphic types
- \( \Psi \) ranges over sets of substitutions from monomorphic type variables \( \mathcal{V}_M \) to monomorphic types

I.1 Declarative type system

I.1.1 Modifications to the declarative type system. See Appendix D for the full declarative system, without the rules for extensions.

In the next subsection (I.1.2), we will define normalized forms for derivations of the declarative type system. These normalized form are not unique, but still satisfies properties that will be used to prove the safety of the declarative type system, as well as the completeness of the declarative type system towards the declarative one.

In order to be able to express these normalized forms, we first need to slightly modify the declarative type system (of course, all the changes made are admissible). All the proofs (in this section and the following ones) will refer to this modified version of the declarative type system.

First, we modify the \([\text{Ax}]\) rule so that it can perform a renaming of the polymorphic type variables of the returned type:

\[
\begin{align*}
\text{[Ax]} & \quad \Gamma \vdash x : \Gamma(x) \rho
\end{align*}
\]

This new \([\text{Ax}]\) rule can be derived in the initial declarative type system by composing a \([\text{Ax}]\) rule and a \([\text{Inst}]\) rule. However, it makes sense to allow the \([\text{Ax}]\) rule to perform a renaming directly so that it can avoid correlation between two types without resorting to the \([\text{Inst}]\) rule (that will only be used before destructor). For instance, we want to be able to type the pair \((f, f)\), with \(f\) being a variable with type \(\alpha \rightarrow \alpha\), with the type \((\alpha \rightarrow \alpha) \times (\beta \rightarrow \beta)\) and without having to use a \([\text{Inst}]\)

rule.

Secondly, we will use \([\lor]\) and \([\land]\) rules of multiple arity instead of the binary ones:

\[
\begin{align*}
\text{[\lor]} & \quad \Gamma \vdash e' : s' \quad (\forall i \in I) \quad \Gamma, x : s \land u_i \vdash e : t \\
& \quad \Gamma \vdash e[e'/x] : t \quad \sqrt{i \in I} u_i \equiv 1 \\
\text{[\land]} & \quad (\forall i \in I) \quad \Gamma \vdash e : t_i \\
& \quad \Gamma \vdash e : \land_{i \in I} t_i \quad I \neq \emptyset
\end{align*}
\]

The new \([\land]\) rule can be derived in the previous system by composing several \([\land]\) rules. The new \([\lor]\) rule, however, is admissible in a more subtle way:
From any decomposition $\bigvee_{i \in I} u_i$ covering $\bot$, we can construct a partition $\bigvee_{i \in J} u'_i$ of $\bot$ with $|J| \geq 2$ and such that $\forall j \in J. \exists i \in I. u'_j \leq u_i$.

A decomposition into a partition $\{u'_1, u'_2, u'_3\}$ of $\bot$ can easily be obtained by composing two binary $[\lor]$ rules:

\[
\frac{\Gamma \vdash e' : \sigma}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{{}}}}}}
\]

with $X$ being the following derivation:

\[
\frac{\Gamma \vdash e' : s \quad \frac{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{{}}}}}{{}}}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{{}}}}}
\]

This construction can be generalized for any partition of $\bot$ of cardinality at least 2.

Every premise in the construction above is derivable from one of the $\Gamma, x : s \land u_j \vdash e : t$ of the initial derivation by alpha-renaming and monotonicity (see I.4 below).

Lastly, we will distinguish variables that are introduced by a $[\rightarrow I]$ from variables introduced by a $[\lor]$; we will use binding variables $(x, y, z)$ for the latter. More formally, the syntax of expressions is extended, and the rules changed accordingly:

Expressions $e ::= c \mid x \mid x \land \lambda x. e \mid ee \mid (e, e) \mid \pi_i e \mid (e \in \tau) \mid ? : e$ (26)

\[
\frac{\frac{\frac{\Gamma \vdash e' : s \quad \frac{\frac{\frac{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{{}}}}}{{}}}}{}}}{{}}}{\frac{\frac{\frac{\frac{\frac{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{\frac{\Gamma, y : s \land u'_1 \vdash e\{y/x\} : t}{{}}}}}{{}}}}{}}}{{}}}}
\]

Note that this is still equivalent to the initial type system as both binding variables and lambda variables are treated the same way ($[\text{Ax}_\lambda]$ and $[\text{Ax}_\lor]$ both correspond to the $[\text{Ax}]$ rule we defined earlier).

Also, we will call structural rules the rules $[\text{Const}]$, $[\text{Ax}_\lambda]$, $[\rightarrow I]$, $[\lor I]$, $[\land E]$, $[\land E_1]$, $[\land E_2]$, $[\land 0]$, $[\land e_1]$ and $[\land e_2]$. Note that the rule $[\text{Ax}_\lor]$ is not considered structural.

I.1.2 Normalisation Lemmas. In the proofs below, we will sometimes omit the guardians of some rules in the derivations. This is only for concision and clarity, and only trivially verified guardians will be omitted.

**Lemma I.1.** Any derivation $\Gamma \vdash e : t$ can be transformed into a derivation $\Gamma \vdash e : t \rho$ (for any renaming $\rho$ on polymorphic type variables) without changing the structure of the derivation.

**Proof.** Any polymorphic type variable in $t$ must be introduced either by an axiom or a $[\text{Inst}]$ rule. Thus, we can derive $\Gamma \vdash e : t \rho$ with the following transformations:

- The renaming $\rho'$ of any axiom rule is replaced by the renaming $\rho \circ \rho'$
- The substitution $\sigma$ of any $[\text{Inst}]$ rule is replaced by the substitution $\rho \circ \sigma \circ \rho^{-1}$

**Definition I.2.** We define the order relation $\leq_\rho$ on types as follows (with $\Sigma$ a set of substitutions from $\mathcal{V}_p$ to types):

$$t_1 \leq_\rho t_2 \iff \exists \Sigma. t_1 \Sigma \leq t_2$$
**Definition I.3.** We define the order relation \( \leq_\rho \) on environments as follows:

\[
\Gamma_1 \leq_\rho \Gamma_2 \iff \forall x \in \text{dom}(\Gamma_2). x \in \text{dom}(\Gamma_1) \text{ and } \Gamma_1(x) \leq_\rho \Gamma_2(x)
\]

where \( x \) denotes both lambda variables and binding variables (usually denoted by \( x \)).

**Lemma I.4 (Monotonicity).** If \( \Gamma \vdash e : t \) and \( \Gamma' \leq_\rho \Gamma \), then \( \Gamma' \vdash e : t \).

**Proof.** We apply the rules \([\text{Inst}], [\land]\) and \([\leq]\) just after the axiom rules of the original derivation whenever required. \(\square\)

**Lemma I.5 (Generation of an arbitrary [\lor] rule).** Let \( \Gamma \) a type environment, \( e \) and \( e_x \) two expressions, and \( \{u_i\}_{i \in I} \) a set of monomorphic types such that \( \lor_{i \in I} u_i \simeq 1 \). Let \( D \) be a derivation for the judgement \( \Gamma \vdash e \{e_x/x\} : t \) such that \( D \) does not contain any [\lor] rule that performs a substitution \( \{e_y/y\} \) with \( e_y \) being a strict subexpression of \( e_x \). If \( e_x \) is typable under the context \( \Gamma \), then \( D \) can be transformed so that it ends with an application of the [\lor] rule of the following form:

\[
\begin{align*}
\vdots & \quad [\lor] & \quad [\land] & \quad [\lor] & \quad [\land] & \quad [\lor] \\
\Gamma \vdash e_x : s & \quad \Gamma, x : s \land u_i \vdash e : t & \forall i \in I \\
\Gamma \vdash e \{e_x/x\} : t
\end{align*}
\]

for some type \( s \) that cannot be choosen arbitrarily.

This transformation does not add any new structural rule nor [\lor] rule (except the one at the root) to the derivation.

**Proof.** Let’s call \( C \) a derivation for \( \Gamma \vdash e_x : 1 \). First, we collect in \( D \) the set \( \{C_k\}_{k \in K} \) of all the subderivations for \( e_x \). By noticing that no variable in \( \text{fv}(e_x) \) could have been introduced by a lambda in \( e \) (we recall that all our substitutions are capture-avoiding), we know that these derivations are valid under the environment \( \Gamma \).

Then, we build the following derivation:

\[
\begin{align*}
\vdots & \quad [\land] & \quad [\lor] & \quad [\land] & \quad [\lor] & \quad [\land] & \quad [\lor] \\
\Gamma \vdash e_x : 1 & \quad \forall k \in K & \quad \forall k \in K & \quad \forall k \in K & \quad \forall k \in K & \quad \forall k \in K \\
C & \quad C_k & \quad \forall k \in K & \quad \forall k \in K & \quad \forall k \in K & \quad \forall k \in K \\
\Gamma \vdash e_x : \land_{k \in K} t_k & \quad \Gamma \vdash e_x : \land_{k \in K} t_k & \quad \Gamma \vdash e_x : \land_{k \in K} t_k & \quad \Gamma, x : (\land_{k \in K} t_k) \land u_i \vdash e : t & \forall i \in I \\
\Gamma \vdash e \{e_x/x\} : t
\end{align*}
\]

with each \( D'_k \) being a derivation easily derived from \( D \) by substituting \( e_x \) by \( x \) when relevant, using an axiom rule on \( x \) instead of a subderivation for \( e_x \) when necessary, and by using monotonicity (I.4). The hypothesis on the derivation \( D \) ensures that it does not contain any conflicting [\lor] rule that would become inapplicable due to the fact that \( e_x \) has been substituted by \( x \). \(\square\)

**Lemma I.6 (Deletion of aliasing).** In any derivation, any occurrence of a [\lor] rule applying a substitution \( \{x/y\} \) can be removed (without adding any other [\lor] rule nor structural rule in the derivation).

**Proof.** The following transformation can be performed:
where $C_j$ is easily derived from $C_j$ by replacing occurrences of $[Ax_v]$ on $y$ by $[Ax_v]$ on $x$ and applying the monotonicity lemma (I.4). Note that we have $s \cup u_k \leq s''$ as the derivation $B$ can only derive for $x$ a type larger than $\Gamma''(x)$, and thus $s \cup u_k \cup u'' \leq s'' \cup u''$.

For the following lemmas, we fix an arbitrary total order $\leq$ over the expressions of the source language modulo alpha-renaming. This order must be an extension of the subexpression order (i.e. if $e_1$ is a subexpression of $e_2$ modulo alpha-renaming, then we should have $e_1 \leq e_2$).

Then, we introduce for convenience the notation $[\forall \cap \text{Pat}]$ for denoting, given a derivation $D$, a segment of a branch of $D$ that is only composed of a succession of $[\forall]$ rules, the one connected to the next by any of its premises except the first one, and that is compatible with the $\leq$ order in the following way: if a rule of this segment is doing the substitution $\{e_i/x\}$ and if its premise is also in the segment and is doing the substitution $\{e_j/y\}$, then $[e_i] \leq [e_j]$, where $[e]$ is abusively used here to denote the expression obtained by applying to $e$ all the substitutions made by the $[\forall]$ rules crossed when going toward the root of $D$.

We also introduce two new notations that take the form of two new rules, but are actually just shortands for a specific pattern:

\[
\frac{A}{\Gamma \vdash e : t'} \quad \frac{\exists \sigma. \; \Lambda \sigma \in \Sigma \; t' \sigma \leq t}{\Gamma \vdash e : t}
\]

\[
\frac{A_i}{\Gamma, \lambda x. e : \bigwedge_{i \in I} u_i \rightarrow t_i} \quad \frac{\Lambda_i}{\Gamma \vdash \lambda x. e : \bigwedge_{i \in I} u_i \rightarrow t_i}
\]
The normalisation lemmas that follow are cumulative: each one can be combined with the next ones (the transformations introduced by a lemma do not break the properties of the previous lemmas).

**Lemma I.7 (Normalisation of $\lor$).** Any derivation of $\Gamma \vdash e : t$ can be transformed so that:

- every occurrence of a structural rule except $[\to I]$ does not have any structural rule (including $[\to I]$) in the subderivations of its premises, and
- the premise of any occurrence of a $[\to I]$ rule is either a $[\vee]$ rule or its subderivation does not contain any structural rule, and
- any premise except the first of any occurrence of a $[\vee]$ rule is either another $[\vee]$ rule or its subderivation does not contain any structural rule

and such that every occurrence of $[\vee]$ rule doing the substitution $e'\{e_x/x\}$ is such that:

- $e_x$ is not a subexpression of $e'$ (maximal sharing), and
- $e_x$ is not a binding variable (no aliasing), and
- $x$ is a subexpression of $e'$ (no useless binding)

and is part of a $[\vee\text{Pat}]$ pattern that starts either:

- At the root of the derivation (optionally preceded by a $[\text{Inst}\land\leq]$ pattern), or
- At the premise of a $[\to I]$ rule that introduces a variable $y$ such that $y \in \text{fv}(e_x)$, or
- At the $n$-th premise ($n \geq 2$) of a $[\vee]$ rule that introduces a variable $y$ such that $y \in \text{fv}(e_x)$.

**Proof.** We can transform any derivation into a derivation that satisfies these properties. First, we can remove any aliasing by applying I.6 as needed. We can also trivially remove useless bindings (i.e. those doing a substitution $e'\{e_x/x\}$ where $e'$ does not contain $x$).

Then, let’s consider, in the whole derivation, all the rules that satisfy one of these:

- It is a structural rule that is in the subderivation of a premise of another structural rule except $[\to I]$,  
- It is a structural rule in the subderivation of the premise of a $[\to I]$ rule with no $[\vee]$ rule at the root,  
- It is a structural rule in the subderivation of the $n$-th premise ($n \geq 2$) of a $[\vee]$ rule with no other $[\vee]$ rule at the root,  
- It is a $[\vee]$ rule such that $e_x$ is a subexpression of $e'$,  
- It is a $[\vee]$ rule that is not part of a $[\vee\text{Pat}]$ pattern as described in the statement.

If there is no such rule, then we are done (the properties of this lemma are satisfied). Otherwise, to each of these faulty rules, we associate an expression:

- For a structural rule applied on an expression $e'$, we associate $e'$  
- For a $[\vee]$ rule doing the substitution $\{e_x/x\}$, we associate $e_x$

Now, we select among those rules the one that minimizes (according to the order $\leq$) $[e']$, with $e'$ its associated expression and $[e]$ denoting the expression $e$ to which we apply all the substitutions made by the $[\vee]$ rules crossed when going toward the root of the derivation. Let’s call this rule $R$ and the associated expression $e'$.

Now, let’s locate, in the segment from the root to $R$, the rule $R'$ nearest from the root such that:

- All lambda variables and binding variables in $e'$ are defined in the environment of the judgement of $R'$,  
- $R'$ is not a $[\vee]$ rule, or it makes a substitution $\{e_y/y\}$ with $[e'] \leq [e_y]$

Let’s note $\Gamma_{R'} \vdash e_{R'} : t_{R'}$ the judgement of $R'$, and let’s consider the associated subderivation. Note that, in this subderivation, there is no $[\vee]$ rule that makes a substitution $\{e_y/y\}$ with $e_y$ a
strict subexpression of $e'$, because $[e']$ would be smaller than $[e''']$, thus breaking the properties of the lemma and contradicting the minimality of $[e''']$. Also note that this subderivation contains $R$.

We apply the lemma I.5 on the root of this subderivation ($R'$) so that it performs the substitution $e_{R'}(e'/z)$, with $z$ fresh and $e_{R'} = e_{R'}(z/e')$, and using the decomposition $\{u_i\}_{i \in I} = \{1\}$. This lemma can be applied as:

- There cannot be in our subderivation any $[\lor]$ rule substituting a strict subexpression of $e'$,
- We know that $\Gamma_{R'} \vdash e' : 1$ holds: we can derive it from the first premise of $R$ if $R$ is a $[\lor]$ rule, or from $R$ itself if $R$ is a structural rule.

Now, in this new subderivation, if $R$ is a $[\lor]$ rule, then it can be removed by using lemma I.6 (as well as other aliasing that would have been introduced in other branches). Not that, if $R$ is a structural rule, it has already been eliminated by the application of I.5.

Finally, we put our subderivation back in its place in the original derivation. Note that this process does not introduce any new rule that would break the properties of the lemma. However, an already existing $[\lor]$ rule could start breaking the lemma due to this process if it was part of a $[\lor\text{PAT}]$ pattern starting on $R$, but in this case we know that its associated expression $e''$ is such that $[e'] \leq [e'']$. Thus, we can conclude by repeating this process until all the rules satisfy the properties of the lemma.

\[ \square \]

**Lemma I.8 (Normalisation of $[\text{Inst}]$).** Any derivation of $\Gamma \vdash e : t$ can be transformed so that every application of $[\text{Inst}]$ is part of a $[\text{Inst} \land \leq]$ pattern that is either:

- At the root of the derivation, or
- The first premise of a $[0]$, $[\in_1]$ or $[\in_2]$ rule, or
- The premise of a $[\times E_1]$ or $[\times E_2]$ rule, or
- One of the premises of a $[\rightarrow E]$ rule.

**Proof.** First, we apply the normalisation lemma I.7 on the derivation. Now, we proceed by induction on $(n_\lor, n)$ (using the lexicographic order), with $n_\lor$ the number of $[\lor]$ rules in the derivation, and $n$ the total number of rules in the derivation.

The base case is trivial.

For the inductive case, we consider the root of the derivation.

If the root is a $[\text{Inst}]$ or $[\leq]$, we apply the induction hypothesis to its premise:

- If the new premise does not end with a $[\text{Inst} \land \leq]$ pattern, then we are already done (note that a single $[\text{Inst}]$ or $[\leq]$ is a particular cases of a $[\text{Inst} \land \leq]$ pattern).
- If the new premise ends with a $[\text{Inst} \land \leq]$ pattern, then we can trivially merge them together into a single $[\text{Inst} \land \leq]$ pattern.

If the root is a $[\land]$, we apply the induction hypothesis to all its premises:

- If none of its new premises end with a $[\text{Inst} \land \leq]$ pattern, then we are already done.
- If (at least) one of its new premises ends with a $[\text{Inst} \land \leq]$ pattern, we can consider without loss of generality that all its premises end with a $[\text{Inst} \land \leq]$ pattern. We then apply the following transformation (with $\Sigma = \bigcup_{i \in I} \Sigma_i$):
If the root is a $[\lor]$:  

- We first apply the induction hypothesis to its first premise. If its new first premise ends with a $[\text{Inst} \land \leq]$ pattern, we apply the following transformation (otherwise, we continue to the next step):

\[
\frac{A_i}{\Gamma \vdash e : t_i'} \quad (\land_{\sigma \in \Sigma_i} t_i' \sigma \leq t_i) \quad \forall i \in I
\]

\[
[\land]
\frac{A_i}{\Gamma \vdash e : t_i} \quad \forall i \in I
\]

\[
[\text{Inst} \land \leq]
\frac{A_i}{\Gamma \vdash e : \land_{i \in I} t_i'} \quad \forall i \in I
\]

\[
\frac{\land_{\sigma \in \Sigma} (\land_{i \in I} t_i') \sigma \leq \land_{i \in I} \land_{\sigma \in \Sigma_i} (\land_{i \in I} t_i') \sigma}{\Gamma \vdash e : \land_{i \in I} t_i}
\]

• The next step is to apply the induction hypothesis on the other (new) premises of this $[\lor]$ rule. If at least one of these new premises is ending with a $[\text{Inst} \land \leq]$ pattern, we can consider without loss of generality that it is the case for all of them. We can also suppose that the types in the conclusion of these premises all have disjoint polymorphic type variables: if it is not the case, it can be ensured by applying I.1 to these premises and adding an instantiation at the root to compensate. Then, we apply the following transformation:

\[
\frac{A}{\Gamma \vdash e' : s'} \quad (\land_{\sigma \in \Sigma} s' \sigma \leq s) \quad B_i
\]

\[
[\lor]
\frac{\Gamma \vdash e' : s}{\Gamma \vdash e : t}
\]

\[
\frac{B_i}{\Gamma, x : s \land u_i \vdash e : t \quad \forall i \in I}
\]

\[
\frac{\Gamma, x : s' \land u_i \leq_p s \land u_i \vdash e : t \quad \forall i \in I}{\Gamma \vdash e' \{e' / x\} : t}
\]

with $B'_i$ a derivation easily derived from $B_i$ by monotonicity (I.4). Note that the application of the monotonicity lemma might add unwanted occurrences of a $[\text{Inst} \land \leq]$ pattern, which will be eliminated with the next step.
We justify this \([\text{INST} \land \leq]\) application by using \(\Sigma = \{\sigma_1 \cup \cdots \cup \sigma_n \mid \sigma_1 \in \Sigma_1, \ldots, \sigma_n \in \Sigma_n\}\) with \(I = \{1, \ldots, n\}\), and where \(\cup\) designates the composition of disjoint substitutions (their disjointness can be guaranteed by the fact that we assumed every \(t_i\) to have disjoint polymorphic vars):

\[
\begin{align*}
\land_{\sigma \in \Sigma}(\lor_{i \in 1..n} t_i)\sigma & \\
\cong \land_{(\sigma_1, \ldots, \sigma_n) \in \Sigma_1 \times \cdots \times \Sigma_n}(\lor_{i \in 1..n} t_i)(\sigma_1 \cup \cdots \cup \sigma_n) & \\
\cong \land_{(\sigma_1, \ldots, \sigma_n) \in \Sigma_1 \times \cdots \times \Sigma_n} \lor_{i \in 1..n} t_i(\sigma_1 \cup \cdots \cup \sigma_n) & \\
\cong \lor_{i \in 1..n} \land_{\sigma_i \in \Sigma_i} t_i\sigma_i & \quad \text{(distributivity of} \lor \text{over} \land) \\
\leq \lor_{i \in 1..n} t_i & \leq t
\end{align*}
\]

The new \([\leq]\) rule that appears as premise of the \([\lor]\) rule could interrupt a \([\lor]\text{Pat}\) pattern, thus breaking the normalisation lemma I.7. In this case, we move this \([\leq]\) rule up as follows:

\[
\begin{align*}
\land_{\sigma \in \Sigma}(\lor_{i \in 1..n} t_i)\sigma & \\
\cong \land_{(\sigma_1, \ldots, \sigma_n) \in \Sigma_1 \times \cdots \times \Sigma_n}(\lor_{i \in 1..n} t_i)(\sigma_1 \cup \cdots \cup \sigma_n) & \\
\cong \land_{(\sigma_1, \ldots, \sigma_n) \in \Sigma_1 \times \cdots \times \Sigma_n} \lor_{i \in 1..n} t_i(\sigma_1 \cup \cdots \cup \sigma_n) & \\
\cong \lor_{i \in 1..n} \land_{\sigma_i \in \Sigma_i} t_i\sigma_i & \quad \text{(distributivity of} \lor \text{over} \land) \\
\leq \lor_{i \in 1..n} t_i & \leq t
\end{align*}
\]

The other cases are similar or straightforward.

\begin{itemize}
  \item Is part of a \([-I\land]\) pattern, or
  \item Is part of a \([\text{INST} \land \leq]\) pattern
\end{itemize}
Proof. First, we apply the normalisation lemmas I.7 and I.8. In particular, this ensures that our derivation does not contain any \([\land]\) rule that has a \([\lor]\) rule as a premise, and that if a \([\land]\) rule has a \([\text{Inst}]\) rule as a premise, then it is part of a \([\text{Inst} \land \leq]\) pattern.

Now, we proceed by induction on the size of the derivation.

The base case is trivial.

For the inductive case, we consider the root of the derivation.

If the root is not a \([\land]\), we can directly conclude by induction on its premises. Thus, let’s assume that the root is a \([\land]\).

If one of its premises is a \([\leq]\) rule, we move it towards the root by applying the following transformation, and we conclude by induction:

\[
\begin{align*}
\Gamma \vdash e : t' & \quad B_i \quad \forall i \in I \\
\Gamma \vdash e : t & \quad \forall i \in I \\
\Gamma \vdash e : t \land \land_{i \in I} t_i & \quad \rightarrow \\
\Gamma \vdash e : t' \land \land_{i \in I} t_i & \\
\end{align*}
\]

If one of its premises is another \([\land]\) rule, we can easily merge them together and conclude by induction.

If one of its premises is a \([\rightarrow E]\) rule, we know that it will also be the case for the other premises (previous normalisation lemmas prevent the cases \([\lor]\) and \([\text{Inst}]\), and the cases for \([\land]\) and \([\leq]\) have been treated previously). Thus, we can apply the following transformation and conclude by induction:

\[
\begin{align*}
A_i & \quad B_i \\
\Gamma \vdash e_1 : s_i \rightarrow t_i & \quad \forall i \in I \\
\Gamma \vdash e_2 : s_i & \quad \forall i \in I \\
\Gamma \vdash e_1 e_2 : t_i & \quad \forall i \in I \\
\Gamma \vdash e_1 e_2 : \land_{i \in I} t_i & \quad \rightarrow \\
\Gamma \vdash e_1 e_2 : (\land_{i \in I} s_i) \rightarrow (\land_{i \in I} t_i) & \\
\end{align*}
\]

The other cases are similar or straightforward. □

Lemma I.10 (Normalisation of \([\leq]\)). Any derivation of \(\Gamma \vdash e : t\) can be transformed so that every application of \([\leq]\) is either:

- At the root of the derivation, or
- The first premise of a \([\in_1]\) or \([\in_2]\) rule, or
- The \(n\)th premise (\(n \geq 2\)) of a \([\lor]\) rule, or
- The premise of a \([\times E_1]\) or \([\times E_2]\) rule, or
- The first premise of a \([\rightarrow E]\) rule
Proof. First, we apply the normalisation lemmas I.7, I.8 and I.9.
Now, we proceed by induction on \((n_v, n)\) (using the lexicographic order), with \(n_v\) the number of \([\lor]\) rules in the derivation, and \(n\) the total number of rules in the derivation.

The base case is trivial.

For the inductive case, we consider the root of the derivation.

If the root is a \([\leq]\), we apply the induction hypothesis on its premise. If this new premise ends with a \([\leq]\), then we can trivially merge these two consecutive \([\leq]\) rules into one \([\leq]\) rule.

If the root is a \([\rightarrow\land]\) pattern, we apply the induction hypothesis to the premises of this \([\rightarrow\land]\) pattern. If one of these new premises end with a \([\leq]\) rule, we can assume without loss of generality that all of them do and we apply the following transformation:

\[
\frac{A_i}{\Gamma, x : u_i \vdash e : t'_i \leq t_i} \quad \text{if } \forall i \in I
\]

\[
\frac{\rightarrow I \land}{\Gamma, x : u_i \vdash e : t_i} \quad \text{if } \forall i \in I
\]

\[
\frac{A_i}{\Gamma \vdash \lambda x.e : \land_{i \in I} u_i \rightarrow t'_i} \quad \text{if } \forall i \in I
\]

\[
\frac{\leq}{\Gamma \vdash \land_{i \in I} u_i \rightarrow t'_i \leq \land_{i \in I} u_i \rightarrow t_i}
\]

If the root is a \([\lor]\):

- We first apply the induction hypothesis on its first premise. If the new first premise ends with a \([\leq]\), we apply the following transformation:

\[
\frac{A_i}{\Gamma, x : u_i \vdash e : s'} \quad \text{if } \forall i \in I
\]

\[
\frac{B_i}{\Gamma \vdash \lambda x.e \land_{i \in I} u_i \rightarrow t_i \rightarrow \land_{i \in I} u_i \rightarrow t'_i \leq \land_{i \in I} u_i \rightarrow t_i}
\]

\[
\frac{A}{\Gamma \vdash e' : s'} \quad \text{if } \forall i \in I
\]

\[
\frac{B'_i}{\Gamma, x : s \land u_i \vdash e : t \vdash e' / x : t}
\]

\[
\frac{A}{\Gamma \vdash e' : s'} \quad \text{if } \forall i \in I
\]

\[
\frac{B'_i}{\Gamma, x : s' \land u_i \rightarrow s \land u_i \vdash e : t}
\]

with \(B'_i\) a derivation easily derived from \(B_i\) by monotonicity (I.4). If the application of the monotonicity lemma breaks the normalisation lemma I.8 or I.9, we can apply I.8 and I.9 again on \(B'_i\). Note that this might add unwanted occurrences of a \([\leq]\) rule, which will be eliminated with the next point.

- Then, we apply the induction hypothesis on the other (new) premises. If one of those new premises end with a \([\leq]\) rule that breaks a \([\lor\)PAT] pattern (and thus the normalisation lemma I.7), we can move it up as done in the proof of I.8.
If the root is a \([\text{Inst} \land \leq] \) pattern, we apply the induction hypothesis on its premise. If this new premise ends with a \([\leq] \), we merge it with the root using the following transformation:

\[
\frac{\frac{\frac{A}{\Gamma \vdash e : t''}}{\leq} \quad \Gamma \vdash e : t'}{\leq} \quad \Gamma \vdash e : t \quad (\text{\(\bigwedge_{\sigma \in \Sigma} t' \sigma \leq t\})}
\]

\[
\frac{\frac{\frac{A}{\Gamma \vdash e : t''}}{\leq} \quad (\text{\(\bigwedge_{\sigma \in \Sigma} t'' \sigma \leq t\})}}{\Gamma \vdash e : t}
\]

The other cases are similar or straightforward. □

**Lemma I.11 (Normalisation).** Any derivation can be transformed into a derivation satisfying the properties of all the normalisation lemmas above (I.7, I.8, I.9, I.10).

**Proof.** The normalisation lemmas above can be applied the one after the other in order to cumulate their properties: the transformations introduced by a one of them preserve the properties of the ones above. □

**I.1.3 Parallel Semantics.** One technical difficulty in the proof of the subject reduction property is that reducing an expression \(e\) might break the use of a \([\lor] \) rule. Indeed, if in the original typing derivation a rule \([\lor] \) substitutes multiple occurrences of the expression \(e\) by a variable \(x\), reducing one occurrence of \(e\) but not the others would alter the application of this rule (the correlation between the reduced \(e\) and the other occurrences of \(e\) will be lost).

To circumvent this issue, we introduce a notion of parallel reduction which forces to reduce all occurrences of a sub-expression at the same time.

The idea is to first define reduction rules that only apply at top-level, and then define a context rule (rule \([\kappa] \) below) that allows reducing under an evaluation context, but that will apply this reduction everywhere in the term.

A step of reduction for the top-level semantics is noted \(\sim_\top\), and a step of reduction for the parallel semantics is noted \(\sim_\parallel\).

**Top-level reductions:**

\[
\begin{align*}
(\lambda x. e) \, v & \sim_\top \ e \{v/x\} & \quad (27) \\
\pi_1(v_1, v_2) & \sim_\top \ v_1 & \quad (28) \\
\pi_2(v_1, v_2) & \sim_\top \ v_2 & \quad (29) \\
(v \in \tau) \ ? \ e_1 : e_2 & \sim_\top \ e_1 & \quad \text{if } v \in \tau & \quad (30) \\
(v \in \tau) \ ? \ e_1 : e_2 & \sim_\top \ e_2 & \quad \text{if } v \in \neg \tau & \quad (31)
\end{align*}
\]

**Parallel reductions:**

\[
[\kappa] \quad e \sim_\parallel e'
\]

\[
E[e] \sim_\parallel (E[e])\{e'/e\}
\]

**Evaluation Context** 

\[
E ::= \ [] \mid v E \mid E e \mid (v, E) \mid (E, e) \mid (E \in \tau) \ ? \ e : e
\]

Here is an example of a reduction step using the parallel semantics:

\[
[\kappa] \quad (\lambda x. \ 42) \sim_\top \ 42
\]

\[
\text{if } (\lambda x. \ 42) \in \text{Int} \text{ then } (\lambda x. \ 42) \text{ else } 0 \sim_\parallel 0 \text{ if } 42 \in \text{Int} \text{ then } 42 \text{ else } 0
\]

Here is an example of a reduction step using the parallel semantics: \[
[\kappa] \quad (\lambda x. \ 42) \sim_\top \ 42
\]

\[
\text{if } (\lambda x. \ 42) \in \text{Int} \text{ then } (\lambda x. \ 42) \text{ else } 0 \sim_\parallel 0 \text{ if } 42 \in \text{Int} \text{ then } 42 \text{ else } 0
\]
Notice that the rule \([\kappa]\) applies a substitution from an expression to an expression. This is formally defined as follows:

**Definition I.12 (Expression Substitutions).** Expression substitutions, ranged over by \(\rho\), map an expression into another expression. The application of an expressions substitution \(\rho\) to an expression \(e\), noted \(e\rho\), is the capture avoiding replacement defined as follows:

- If \(e' \equiv_\kappa e''\), then \(e''\{e/e'\} = e\).
- If \(e' \not\equiv_\kappa e''\), then \(e''\{e/e'\}\) is inductively defined as
  \[ x\{e/e'\} = x \]
  \[(\lambda x.e_0)\{e/e'\} = \lambda x.e_0 \quad x \in \text{fv}(e')\]
  \[(\lambda x.e_0)\{e/e'\} = \lambda x.(e_0\{e/e'\}) \quad x \notin \text{fv}(e) \cup \text{fv}(e')\]
  \[(\lambda x.e_0)\{e/e'\} = \lambda y._{(e_0\{y/x\})}\{e/e'\} \quad x \notin \text{fv}(e), x \in \text{fv}(e'), y \text{ fresh}\]
  \[(\pi_i e_0)\{e/e'\} = \pi_i(e_0\{e/e'\})\]
  \[(e_1, e_2)\{e/e'\} = (e_1\{e/e'\}, e_2\{e/e'\})\]

\((e_1 \in t) \? e_2 : e_3\{e/e'\} = (e_1\{e/e'\} \in t) \? e_2\{e/e'\} : e_3\{e/e'\}\)

Notice that the expression substitutions are up to alpha-renaming and perform only one pass.

**I.1.4 Subject reduction.**

**Property I.13.** If \(\Gamma \vdash v : \tau\), then \(v \in \tau\).

**Proof.** Straightforward, by induction on the derivation of the judgement \(\Gamma \vdash v : \tau\). Note that the case of lambda-abstractions is trivial as \(\tau\) can only be \(\emptyset \rightarrow \vdash\).

**Lemma I.14 (Atomicity of Value Types).** For any derivation \(\Gamma \vdash v : s\) that does not use any \([\vee]\) rule nor \([\leq]\) rule except in the subderivation of a \([\rightarrow I]\) rule, we can deduce that \(s\) cannot be decomposed into a non-trivial union (i.e. \(s \leq \bigvee_{i \in I} s_i \Rightarrow \exists i \in I. s \leq s_i\)).

**Proof.** As \(v\) is a value, we know that the derivation does not use any destructor nor axiom rule (except maybe in the subderivation of a \([\rightarrow I]\) rule). It does not use any \([\vee]\) rule nor \([\leq]\) rule neither (hypothesis). In particular, this implies that \(s\) cannot contain any type variable, except under an arrow. More precisely, we can deduce that \(s\) can be constructed with the following syntax:

**Value Type** \(\bar{i} \quad ::= \quad b \mid t \rightarrow t \mid \bar{i} \times \bar{i} \mid \bar{i} \wedge \bar{i}\)

By starting from the fact that a type \(t_1 \rightarrow t_2\) cannot be decomposed into a non-trivial union (nor can a base type \(b\) be), we can deduce that a type constructed with the syntax above cannot be decomposed into a non-trivial union neither.

**Theorem I.15 (Subject Reduction).** If \(\Gamma \vdash e : t\) and \(e_o \sim_\tau e'_o\), then \(\Gamma \vdash e\{e'_o/e_o\} : t\).

**Proof.** We apply the normalisation lemma (I.11) to the derivation of the judgement \(\Gamma \vdash e : t\), and we proceed by induction on \((d_1, n_{\vee}, n)\) (using the lexicographic order), with \(d_1\) the maximum number of imbricated lambdas in \(e\), \(n_{\vee}\) the number of top-level \([\vee]\) rules in the derivation (i.e. that are not in the subderivation of a \([\rightarrow I]\) rule), and \(n\) the total number of rules in the derivation. Note that, even if the derivation is initially normalized, some binding variables will be substituted by values in some inductive calls, thus the properties of I.7 about structural rules will not hold. The other properties of normalisation, however, will be preserved.
We denote by $\rho$ the substitution $\{e'/e_0\}$ and by $e'$ the expression $e \rho$. If $e$ contains no occurrence of $e_0$ (modulo alpha-renaming), this theorem is trivial. Thus, we will suppose in the following that $e$ contains at least one occurrence of $e_0$.

We consider the root of the derivation:

- **[CONST]** Impossible case ($e$ cannot contain any reducible expression).
- **[Ax$\land$]** Impossible case ($e$ cannot contain any reducible expression).
- **[Ax$\lor$]** Impossible case ($e$ cannot contain any reducible expression).
- **[$\leq$]** By induction on the premise $\Gamma \vdash e : t'$ (with $t' \leq t$), we get a derivation for $\Gamma \vdash e' : t'$, thus we can derive $\Gamma \vdash e' : t$ by using $[\leq]$.
- **[Inst]** Similar to the previous case (by induction on the premise).
- **[$\land$]** Similar to the previous case (by induction on the premises).
- **[$\rightarrow$I]** We have $e' = \lambda x. (e_1, \rho)$. We can derive $\Gamma, x : u_1 \vdash e_1 : t_2$ by induction on the premise $\Gamma, x : u_1 \vdash e_1 : t_2$ (with $t_2 \approx t$), and conclude by using $[\rightarrow$I$]$.
- **[$\times$I]** We have $e' = (e_1, \rho, e_2, \rho)$. We can conclude by induction on the premises similarly to the previous case.
- **[$\rightarrow$E]** We have $e = e_1 e_2$. If $e_0$ is a subexpression of $e_1$ and/or $e_2$, we conclude trivially by induction (as in the previous cases).

Otherwise, $e_0 = e_1 e_2$ and thus the reduction $e_0 \leadsto e_0'$ uses the rule 27. Consequently, we know that $e_0 = e = (\lambda x. e_1) u$ (for some expression $e_1$ and value $u$) and $e_0' = e' = e_1 u/x$.

We have the following premises:

1. $\Gamma, x : u \vdash e : t_1 \rightarrow t_2$ (with $t_2 \approx t$)
2. $\Gamma \vdash v : t_1$

As the derivation is normalized, we know that the premise (1) ends with a $[\text{Inst} \land \leq]$ pattern preceded by a $[\rightarrow$I$\land$] pattern (in particular, there cannot be $[\lor]$ rules in between). Thus, we can extract from it a collection of derivations of the judgements $\Gamma, x : u_i \vdash e_i : s_i$ for $i \in I$, such that $\exists \Sigma. \forall x \in I. (\land_i \Sigma = \Sigma, \land_i e_i \rightarrow s_i \leq t_1 \rightarrow t_2$ (with $\forall \sigma \in \Sigma. \text{dom}(\sigma) \subseteq \mathcal{V}_p$). This is equivalent to $\forall \Sigma \in \mathcal{V}_p, \land_i e_i \rightarrow s_i \leq t_1 \rightarrow t_2$. We define $\{(u'_i, s'_i)\}_{i \in I} = \{(u_i, s_i) | i \in I, \Sigma \in \mathcal{V}_p\}$.

Now, let’s consider a partition $\{v_k\}_{k \in K}$ of $\land_{j \in J} u'_j$ of minimal cardinality that satisfies the following property: $\forall k \in K, \forall j \in J. v_k \leq u'_j$ or $v_k \cap u'_j = \emptyset$.

We can suppose that $K$ is not empty: the case $t_1 \leq \land_{j \in J} u'_j = 0$ is straightforward. For every $k \in K$, we define $J_k = \{ j \in J | v_k \land u'_j \neq \emptyset \}$ ($J_k$ cannot be empty as the partition has minimal cardinality). Note that for all $k \in K$ and $j \in J_k$, we have $v_k \leq u'_j$.

According to the monotonicity lemma (I.4), for every $k \in K$ and $j \in J_k$ we can derive the judgement $\Gamma, x : v_k \vdash e_l : s_j$. Thus, using a $[\land]$ rule, for every $k \in K$ we can derive $\Gamma, x : v_k \vdash e_l : \land_{j \in J_k} s'_j$. Moreover, as $\land_{j \in J} u'_j \rightarrow s'_j \leq t_1 \rightarrow t_2$, we have for every $k \in K$:

$$\land_{j \in J_k} s'_j \leq t_2.$$ Consequently, for every $k \in K$, we can derive the judgement $\Gamma, x : v_k \vdash e_l : t_2$ using a $[\leq]$ rule.

As $u$ is a value and as the derivation is normalized, we can apply I.14 on the premise (2) and deduce that $t_1$ cannot be decomposed into a non-trivial union. As $\{v_k\}_{k \in K}$ is a partition covering $t_1$, we can therefore find $k \in K$ such that $t_1 = v_k \land t_1$. Thus, from $\Gamma, x : v_k \vdash e_l : t_2$, we can easily derive $\Gamma \vdash e_l (v/x) : t_2$ by replacing $[\text{Ax}_\land]$ rules by the derivation (1) and using monotonicity (I.4).

- **[$\times$E]** We have $e = \pi_1 e_1$. If $e_0$ is a subexpression of $e_1$, we conclude trivially by induction.
Otherwise, the reduction \( e_0 \leadsto \tau e'_0 \) uses the rule 28 and thus we know that \( e_0 \equiv e \equiv \pi_1(v_1, v_2) \) and \( e'_0 \equiv e' \equiv v_1 \).

Similarly to the previous case, as the derivation is normalized, we can extract from the derivation of the premise \( \Gamma \vdash (v_1, v_2) : t_1 \times t_2 \) a collection of derivations of the judgements \( \Gamma \vdash v_i : s_i \) and \( \Gamma \vdash v_2 : s_i' \) for \( i \in I \), such that \( \exists \Sigma. \land_{\sigma \in \Sigma} \land_{i \in I}(s_i \times s_i')\sigma \leq t_1 \times t_2 \). In particular, this last property implies \( \land_{\sigma \in \Sigma} \land_{i \in I} s_i\sigma \leq t_1 \).

Therefore, we can conclude this case by using a \([\text{INST} \land \leq] \) pattern with the premises \( \{\Gamma \vdash v_i : s_i\}_{i \in I} \) in order to derive \( \Gamma \vdash v_1 : t_1 \).

\[ \times E_2 \quad \text{Similar to the previous case.} \]

\[ \lor \quad \text{We have } e \equiv e_1\{e_2/x\} \text{ (conclusion of the } [\lor] \text{ rule), and thus } e' \equiv (e_1\{e_2/x\})\{e'_2/e_0\}. \]

If \( e_2 \) is a value, then we can apply I.14 on the first premise \( \Gamma \vdash e_2 : s \) (as the derivation is normalised), which gives that there exists \( i \in I \) such that \( s \land u_i \Rightarrow s \). The corresponding premise, \( \Gamma, x : s \vdash e_1 : t \), can be transformed into a derivation \( \Gamma \vdash e_1\{e_2/x\} : t \) by replacing any occurrence of \( x \) by \( e_2 \) and replacing occurrences of an \([\text{Ax}_x] \) rule on \( x \) by the derivation of the first premise, \( \Gamma \vdash e_2 : s \) (where \( \Gamma \) can be extended as needed to match the current environment). Then, we conclude by applying the induction hypothesis on this new derivation.

We can now assume that \( e_2 \) is not a value. We know that \( e_0 \) does not contain \( x \) as a free variable (otherwise there would be no occurrence of \( e_0 \) in \( e_1\{e_2/x\} \)). Moreover, \( e'_2 \) does not contain \( x \) neither, because a reduction step cannot introduce a new free variable.

There are several cases:

- \( e_0 \) does not contain \( c_2 \) and \( e_2 \) does not contain \( e_0 \). In this case, we have:
  \( e' \equiv (e_1\{e_2/x\})\{e'_2/e_0\} \equiv (e_1\{e'_2/e_0\})\{e_2/x\} \). Thus, we can easily conclude by keeping the first premise of the \([\lor] \) rule and applying the induction hypothesis on the others.

- \( e_2 \) contains \( e_0 \). In this case, we pose \( e'_2 = e_2\{e'_2/e_0\} \).
  We have \( e' \equiv (e_1\{e_2/x\})\{e'_2/e_0\} \equiv (e_1\{e'_2/e_0\})\{e'_2/x\} \). We can easily derive \( \Gamma \vdash e'_2 : s \) by induction on the first premise, and \( \Gamma, x : s \land u_i \vdash e_1\{e'_2/e_0\} : t \) for all \( i \in I \) by induction on the others. Thus, we can derive \( \Gamma \vdash (e_1\{e'_2/e_0\})\{e'_2/x\} : t \) using a \([\lor] \) rule.

- \( e_0 \) contains \( e_2 \) as a strict subexpression. In this case, we pose \( e_\star = e_0\{x/e_2\} \) and \( e'_\star = e'_0\{x/e_2\} \). We know that \( e_1 \) does not contain any occurrence of \( e_2 \) (because the derivation is normalised), and thus no occurrence of \( e_0 \) neither. Consequently, we have \( e' \equiv (e_1\{e_2/x\})\{e'_2/e_0\} \equiv (e_1\{e'_0/e_0\})\{e_2/x\} \).
  As \( e_2 \) is not a value, it can only appear in \( e_0 \) inside a lambda-abstraction, and/or inside a branch of a typecase: otherwise, \( e_2 \) would necessarily be a value for \( e_0 \) to be reducible. Thus, we can deduce that \( e_\star = e_0\{x/e_2\} \leadsto \tau e'_\star\{x/e_2\} = e'_\star \).
  Consequently, we can easily conclude by keeping the first premise of the \([\lor] \) rule and applying the induction hypothesis on the others.

\[ \emptyset \quad \text{We have } e \equiv (e_1\in\tau) \Rightarrow e_2 : e_3. \text{ As values cannot have the type } \emptyset, \text{ we know that } e_1 \text{ is not a value.} \]

Thus, \( e' \equiv (e_1\in\tau) \Rightarrow e_2\rho : e_3\rho \). We can derive \( \Gamma \vdash e_1\rho : \emptyset \) by induction on the premise, and then we can derive \( \Gamma \vdash e' : \emptyset \) by using \([\emptyset] \). 

\[ \epsilon_1 \quad \text{We have } e \equiv (e_1\in\tau) \Rightarrow e_2 : e_3. \text{ There are three cases:} \]

\( e' \equiv (e_1\rho\in\tau) \Rightarrow e_2\rho : e_3\rho \). We can easily conclude by induction on the premises.

\( e' \equiv e_2 \). We can conclude with the second premise.

\( e' \equiv e_3 \). This case is impossible. Indeed, it implies that \( e_1 \) is a value, and as \( \Gamma \vdash e_1 : \tau \) (first premise), we can deduce using the property I.13 that \( e_1 \in \tau \), which contradicts \( e \leadsto \tau e_3 \).

\[ \epsilon_2 \quad \text{Similar to the previous case.} \]

\[ \square \]
Corollary I.16 (Subject reduction). If $\Gamma \vdash e : t$ and $e \leadsto^P e'$, then $\Gamma \vdash e' : t$.

Proof. The root of the derivation of $e \leadsto^P e'$ is a $[x]$ rule, with its premise being of the form $e_0 \leadsto^\tau e'_0$. Additionally, we have $e' \equiv e\{e'_0/e_0\}$. Thus, by using I.15, we obtain $\Gamma \vdash e' : t$. \qed

1.1.5 Progress.

Theorem I.17 (Progress). If $\Gamma \vdash e : t$ and if there is no evaluation context $E$ and variable $x$ (resp. $x$) such that $e \equiv E[x]$ (resp. $e \equiv E[x]$), then either $e$ is a value or $\exists e'$. $e \leadsto^P e'$.

Proof. For convenience, quantifications on a variable $x$ (like in $\forall E, x. e \not\equiv E[x]$) will denote both lambda variables and binding variables.

We apply the normalisation lemma (I.11) to the derivation of the judgement $\Gamma \vdash e : t$, and we proceed by induction on $(n_{\lor}, n)$ (using the lexicographic order), with $n_{\lor}$ the number of top-level $[\lor]$ rules in the derivation (i.e. that are not in the subderivation of a $[\to\land]$ rule), and $n$ the total number of rules in the derivation.

Note that, even if the derivation is initially normalized, some binding variables will be substituted by values in some inductive calls, thus the properties of I.17 about structural rules will not hold. The other properties of normalisation, however, will be preserved.

We consider the root of the derivation:

[\textbf{Const}] Trivial ($e$ is a value).

[\textbf{Ax}_x] Impossible case ($e$ cannot be a variable).

[\textbf{Ax}_\lor] Impossible case ($e$ cannot be a variable).

[\leq] Trivial (by induction on the premise).

[\textbf{Inst}] Trivial (by induction on the premise).

[\land] Trivial (by induction on one of the premises).

[$\to\land]$ Trivial ($e$ is a value).

[$\times$] We have $e \equiv (e_1, e_2)$.

- If $e_1$ is not a value, and as we have $\forall E, x. e_1 \not\equiv E[x]$, we know by applying the induction hypothesis that $e_1$ can be reduced. Thus, $e$ can also be reduced with the context ($[$, $e_2$).

- If $e_1$ is a value, then we can then apply the induction hypothesis on the second premise (as $e_1$ is a value, we know that $\forall E, x. e_2 \not\equiv E[x]$). 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 with the context $(e_1, [])$).

[$\to\lor$] We have $e \equiv e_1 e_2$, with $\Gamma \vdash e_1 : s \to t$ and $\Gamma \vdash e_2 : s$.

- If $e_1$ is not a value, and as we have $\forall E, x. e_1 \not\equiv E[x]$, we know by applying the induction hypothesis that $e_1$ can be reduced. Thus, $e$ can also be reduced with the context $[ ] e_2$.

- If $e_1$ is a value, we can apply the property I.13 on it. As $\Gamma \vdash e_1 : 0 \to 1$, it gives that $e_1 \in 0 \to 1$ and thus $e_1 \equiv \lambda x. e_0$. Moreover, we can apply the induction hypothesis on the second premise (as $e_1$ is a value, we know that $\forall E, x. e_2 \not\equiv E[x]$). 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 $e$ can be reduced using the rule 27, otherwise, $e$ can be reduced with the context $e_1[ ]$).

[$\times E_1$] We have $e \equiv \pi_1 e_\circ$, with $\Gamma \vdash e_\circ : t \times s$. By induction on the premise, we know that $e_\circ$ is either a value or it can be reduced. If $e_\circ$ can be reduced, then $e$ can also be reduced with the context $\pi_1[ ]$. Otherwise, as $\Gamma \vdash e_\circ : 1 \times 1$, we know by using the property I.13 that $e_\circ \in 1 \times 1$. Thus, $e_\circ \equiv (v_1, v_2)$ (with $v_1$ and $v_2$ two values) and consequently $e$ can be reduced using the rule 28.

[$\times E_2$] Similar to the previous case.

[$\lor$] We have $e \equiv e_1\{e_2/x\}$, with $\Gamma \vdash e_2 : s$ and $\forall i \in I, \Gamma, x : s \land u_i \vdash e_1 : t$. There are two cases:
• There exists a reduction context $E$ such that $e_1 \equiv E[x]$. In this case, we know that 
$\forall E', y. e_2 \not\equiv E'[y]$, otherwise we would have $e \equiv E'[y]$. Thus, we can apply the
induction hypothesis on the first premise. It gives that either $e_2$ is a value or it can be
reduced.

If $e_2$ can be reduced, then $e$ can also be reduced with the context $E$.
Otherwise, $e_2$ is a value, thus we can apply I.14 on the first premise $\Gamma \vdash e_2 : s$ (as the
derivation is normalised), which gives that there exists $i \in I$ such that $s \wedge u_i = s$. The
corresponding premise, $\Gamma, x : s \wedge u_i \vdash e_1 : t$, can be transformed into a derivation
$\Gamma \vdash e_1\{e_2/x\} : t$ by replacing any occurrence of $x$ by $e_2$ and by replacing occurrences
of an $[\text{Ax}_v]$ rule on $x$ by the derivation of the first premise, $\Gamma \vdash e_2 : s$ (where $\Gamma$ can be
extended as needed to match the current environment). Then, we conclude by applying
the induction hypothesis on this new derivation.

• There is no reduction context $E$ such that $e_1 \equiv E[x]$. Thus, we can apply the induction
hypothesis on the $n$-th premises ($n \geq 2$). It gives that either $e_1$ is a value or it can be
reduced. We can easily conclude in both cases (if $e_1$ is a value, then $e$ is also a value,
and if $e_1$ can be reduced, then $e$ can also be reduced).

[0] We have $e \equiv \langle e_0 \in \tau \rangle ? e_1 : e_2$, with $\Gamma \vdash e_0 : 0$. As values cannot have the type $\emptyset$, we know
that $e_0$ is not a value. Thus, by induction on the premise, we know that $e_0$ can be reduced.
Consequently, $e$ can be reduced with the context $([ \ [ ] \in \tau ] ? e_1 : e_2$.

[ε₁] We have $e \equiv \langle e_0 \in \tau \rangle ? e_1 : e_2$, with $\Gamma \vdash e_0 : \tau$. By induction on the first premise, we know
that $e_0$ is either a value or it can be reduced. If $e_0$ is a value, then $e$ can be reduced using
the rule 30. Otherwise, $e_0$ can be reduced and thus $e$ can also be reduced with the context
$([ \ [ ] \in \tau ] ? e_1 : e_2$.

[ε₂] Similar to the previous case.

\□

**Corollary I.18 (Progress).** If $\emptyset \vdash e : t$, then either $e$ is a value or $\exists e'. e \leadsto e'$.

**Proof.** We can deduce from $\emptyset \vdash e : t$ that there is no evaluation context $E$ and variable $x$ (resp.
$x$) such that $e \equiv E[x]$ (resp. $e \equiv E[x]$). Thus, we can conclude with I.17. \□

**Property I.19.** For any $e$ and $v$, if $e \leadsto^*_* v$ or $e \leadsto^* P v$ (e diverges with $\leadsto_\tau$), then either there exists
$v'$ such that $e \leadsto^* v'$ or $e \leadsto^* (e$ diverges with $\leadsto_\tau$). (see Appendix B for the definition of the semantics
$\leadsto$)

**Proof.** We define the following syntax for expressing a path in an expression:

**Path**  
$\phi ::= [ \ ] | \phi \_ | \_ \phi | (\phi, \_) | (\_ , \phi) | \lambda \phi | (\phi \in \_) \ ? \ _ \ | (\_ \ in \_) \ ? \ _ \ | \phi : _ \ | (\_ \ in \_) \ ? \ _ : \phi$

We first introduce a new semantics $\leadsto_C$ similar to $\leadsto_\tau$ but where the reductions can happen
under any context (not just an evaluation context).

We trivially have that following property: for any $e$ and $v$, if $e \leadsto^*_C v$ then $e \leadsto^* v$. Thus, we
only need to prove the following property: for any $e$, if $e \leadsto^*_C v$ (for any $v$) or $e \leadsto^*_C$, then $e \leadsto^*$ or
there exists $v'$ such that $e \leadsto^* v'$.

We will assume that $e \leadsto^*_C v$ (for some $v$) or $e \leadsto^*_C$, and show that either $e$ is a value or $e \leadsto e'$
for some $e'$ such that $e' \leadsto^*_C v'$ (for some $v'$) or $e' \leadsto^*_C$. The result above can then easily be deduced
by iterating.

Let’s represent each reduction happening in $e \leadsto^*_C v$ or $e \leadsto^*_C$, by the path under which it is
happening (the associated top-level reduction can easily be retrieved as at most one top-level
reduction can apply on a given expression). It gives us a (potentially infinite) list of paths. Now,
let’s consider the first path in this list (if any) that corresponds to a valid reduction context in \( e \) (i.e. such that the _ in the path can be completed to give a reduction context matching \( e \)):

- If there is no such path, and as the final expression is a value, we know that all these reductions happened inside a lambda. Thus, \( e \) is also a value.
- If such a path \( \phi \) exists, then we know that all the others reductions in the list before it are happening on a path that is not a prefix of \( \phi \) (because evaluation contexts are closed by inclusion). Thus, the path \( \phi \) must exist in \( e \). Let’s call \( e_\phi \) the subexpression at the path \( \phi \) in \( e \), and \( e'_\phi \) the actual subexpression that was reduced in the initial reduction sequence \( e \leadsto^{*}_C v \) or \( e \leadsto^{*}_C \).

We know that \( e'_\phi \) is reducible, and we also know that \( e'_\phi \) can be obtained from \( e_\phi \) only by performing reductions not in an evaluation context (as the path we choosed is the first to be a reduction context). We can deduce that \( e_\phi \) is also reducible.

Thus, in our list of paths representing the reduction sequence \( e \leadsto^{*}_C v \) or \( e \leadsto^{*}_C \), we can move the first occurrence of \( \phi \) in first position and update the paths that were before it (and that were suffixes of \( \phi \) in order to obtain the same expression as before (some reductions might need to be removed or duplicated). We obtain a sequence of reductions \( e \leadsto_C e' \leadsto^*_C v \) or \( e \leadsto_C e' \leadsto^*_C \) such that \( e \leadsto e' \), which concludes the proof.

\[ \square \]

**Theorem I.20 (Type safety).** For any expression \( e \), if \( \varnothing \vdash e : t \), then either \( e \leadsto v \) with \( \varnothing \vdash v : t \) or \( e \leadsto^{*} \) (\( e \) diverges).

**Proof.** Straightforward application of I.16, I.18 and I.19. \( \square \)

### I.2 Algorithmic type system

#### I.2.1 Maximal Sharing Canonical Form. As defined in I.1.1, we will consider that expressions of the source language can contain binding variables. Consequently, the unwinding operator \([\_\_\_]\) can freely be used on atoms and on canonical forms containing free binding variables.

**Property I.21.** For any expression \( e \) of the source language, \([\text{term}([e])]\) \( \equiv_\alpha e \).

**Proof.** Straightforward structural induction on \( e \). \( \square \)

**Property I.22.** If \( \kappa \equiv_\kappa \kappa' \), then \([\kappa]\) \( \equiv_\alpha [\kappa']\).

**Proof.** If a reordering as defined in Definition 3.1 applies at top-level on the expression \( \text{bind } x_1 = a_1 \text{ in } \text{bind } x_2 = a_2 \text{ in } \kappa \), the unwinding remains unchanged: as \( x_1 \notin \text{fv}(a_2) \) and \( x_2 \notin \text{fv}(a_1) \), we have \( \kappa\{a_1/x_1\}\{a_2/x_2\} = \kappa\{a_2/x_2\}\{a_1/x_1\} \).

The general case can easily be deduced with the observation that \( \forall C, \kappa_1, \kappa_2. \ [\kappa_1] \equiv_\alpha [\kappa_2] \Rightarrow [\text{C}[\kappa_1]] \equiv_\alpha [\text{C}[\kappa_2]] \) (with \( C \) denoting an arbitrary context). \( \square \)

**Property I.23 (Equivalence of MSC-forms).** If \( \kappa_1 \) and \( \kappa_2 \) are two MSC-forms and \([\kappa_1]\) \( \equiv_\alpha [\kappa_2]\), then \( \kappa_1 \equiv_\kappa \kappa_2 \).

**Proof.** We will show that \( \kappa_2 \) can be transformed into \( \kappa_1 \) just with alpha-renaming and reordering of independent bindings (as specified in the definition of \( \equiv_\kappa \)).

We represent \( \kappa_1 \) as a pair \( (b_1, e_1) \) with a \( b_1 \) being a list of definitions representing its top-level bindings, and \( e_1 \) its final binding variable (noted \( e_1 \) because we allow any expression in order to be more general).

More formally, it gives a representation \( (b, e) \) using the following syntax:
List of definitions  \[ b := x \mapsto a; \ldots; x \mapsto a \]  

(32)

Similarly, we represent \( \kappa \) as a pair \((b_2, e_2)\).

We can define an unwinding operator \([\cdot]\), similar to the one defined in Appendix E, that transforms this representation into an expression of the source language. As \([\kappa_1] \equiv_{\alpha} [\kappa_2]\), we have \([ (b_1, e_1) ] \equiv_{\alpha} [(b_2, e_2)]\). We alpha-rename \((b_2, e_2)\) so that \(e_2 = e_1\) and \([(b_1, e)] = [(b_2, e)]\).

Now, let’s prove the following property (from which the property to prove can be deduced): let \(b_1\) and \(b_2\) be two lists of definitions and \(e\) be an expression such that:

- \([ (b_1, e) ] = [(b_2, e)]\)
- The body of lambdas in \(b_1\) and \(b_2\) are in MSC-form (3.2)
- Both \(b_1\) and \(b_2\) respect the following properties (corresponding to the MSC-form properties applied to the top-level definitions), written here for a list of definitions \(b\):
  1. if \(x_1 \mapsto a_1\) and \(x_2 \mapsto a_2\) are distinct definitions in \(b\), then \(a_1 \not\equiv_{\kappa} a_2\)
  2. for any definition \(x \mapsto \lambda z.\kappa\) in \(b\), any binding \(\text{bind } y = a \in \kappa'\) in \(\kappa\) is such that \(\text{fv}(a) \not\subseteq \text{fv}(\lambda z.\kappa)\)
  3. if \(b\) contains a definition \(x \mapsto a\), then \(x\) is a free variable of one of the next definitions or of \(e\)

Then, we can transform \(b_2\) into \(b_1\) just with alpha-renaming, reordering of independent definitions, and replacement of an atom by a \(\equiv_{\kappa}\)-equivalent one.

We prove this property by induction on the number of definitions in \(b_1\) + the total number of bindings in the atoms.

The base case \((b_1 = \emptyset)\) is trivial: as \([ (b_2, e) ] = [(b_1, e)] = [(\emptyset, e)] = e\), we deduce with property 3 that \(b_2 = \emptyset\).

For the inductive case, let’s say \(b_1 = (b'_1; x \mapsto a_1)\).

With property 3, we know that \(x\) appears in \(e\). As \([ (b_1, e) ] = [(b_2, e)]\), the binding variable \(x\) that appears in \(e\) must be unwound to the same subexpression using the definitions in \(b_1\) than using the definitions in \(b_2\). Thus, \(b_2\) must also feature a definition for \(x\), let’s call \(a_2\) the associated atom. We move in \(b_2\) the definition \(x \mapsto a_2\) at the end (if not already), it gives \(b_2 = (b'_2; x \mapsto a_2)\). We then have \([ (b'_1, a_1) ] = [(b'_2, a_2)]\). As every kind of atom introduces a different syntactic construction, we can deduce that \(a_1\) and \(a_2\) are atoms of the same kind.

- If \(a_1\) and \(a_2\) are atoms that are not lambdas and that do not contain any binding variable (constants, lambda variables), we directly have \(a_1 = a_2\).
- If \(a_1\) and \(a_2\) are atoms that are not lambdas and that contain only one binding variable (projections), we can alpha-rename binding variables in \(b_2\) so that \(a_1 = a_2\).
- If \(a_1\) and \(a_2\) are atoms that are not lambdas and that contain two binding variables (applications, pairs), we consider two cases:
  - If, in \([ (b'_1, a_1) ] = [(b'_2, a_2)]\), these two binding variables are unwound to the same expressions (modulo alpha renaming), we do the following. Let’s call \(x\) and \(y\) the two binding variables in \(a_1\), and let’s show that we necessarily have \(x = y\). We consider the list of definitions \(b_x\), which is a cleaned version of \(b'_1\) where all the definitions that are not related (directly or indirectly) to \(x_1\) have been removed. Similarly, we consider the list of definitions \(b_y\) where the definitions unrelated to \(y\) have been removed. Then, we apply the induction hypothesis to the lists of definition \(b_x\), \(b_y\{x/y\}\) and the expression \(x\). This tells us that \(b_x\) and \(b_y\) are equivalent modulo reordering of the definitions, alpha-renaming and \(\equiv_{\kappa}\)-transformation of atoms. Thus, according to the property 1, \(x\) and \(y\) cannot have two distinct definitions in \(b'_1\), and thus \(x = y\). The same reasoning
can be done for \( a_2 \), and thus we deduce that both \( a_1 \) and \( a_2 \) contain the same binding variable twice. Thus, we can alpha-rename binding variables in \( b_1 \) so that \( a_1 = a_2 \).

- Otherwise, the two binding variables in \( a_1 \) must be different, and the same applies to \( a_2 \). Thus, we can alpha-rename binding variables in \( b_2 \) so that \( a_1 = a_2 \).

• If \( a_1 \) and \( a_2 \) are typecases (containing 3 binding variables), we proceed similarly to the previous case to obtain \( a_1 = a_2 \).

• In the case where \( a_1 \) and \( a_2 \) are lamdas, let’s say \( \lambda x. \kappa_1 \) and \( \lambda x. \kappa_2 \) respectively, we note \((b_{x_1}, x_1)\) and \((b_{x_2}, x_2)\) the representations of \( \kappa_1 \) and \( \kappa_2 \) respectively. We now consider the representation \((b'_{x_1}; b_{x_1}, \lambda x. x_1)\) and remove from it all the unused definitions (i.e. not related to \( x_1 \)), it gives us a new representation \((b''_1, \lambda x. x_1)\). We do the same for \((b'_{x_2}; b_{x_2}, \lambda x. x_2)\), it gives a new representation \((b''_{x_2}, \lambda x. x_2)\). By applying the alpha-renaming \( \{x_1/x_2\} \) to \( b_1 \), we can manage to get \( \lambda x. x_2 = \lambda x. x_1 = e'' \), and we call the induction hypothesis on \((b'_1, b''_{x_1})\) and \((b'_2, b''_{x_2})\) are equivalent modulo reordering of the definitions, alpha-renaming and \( \equiv_{\kappa} \)-transformation of the atoms. By using the property 2, we can deduce that the same applies to \( b_{x_1} \) and \( b_{x_2} \). Thus, we can alpha-rename \( b_2 \) and \( \equiv_{\kappa} \)-transform some of its atoms so that \( b_{x_1} = b_{x_2} \), and thus \( a_1 = a_2 \).

In any case, we get \( a_1 = a_2 = a \), thus the last definition of \( b_1 \) is the same as the last definiton of \( b_2 \). The same can be proven for the previous definitions by using the induction hypothesis on \( b'_1 \), \( b''_{x_1} \) and \( e\{a/x\} \).

**Property I.24.** If \( \kappa \rightarrow \kappa' \), then \( [\kappa] \equiv_{a} [\kappa'] \).

**Proof.** Straightforward: this proof is similar to (and uses) the proof of I.22. □

**Property I.25 (Normalisation).** There is no infinite chain \( \kappa_1 \rightarrow \kappa_2 \rightarrow \cdots \).

**Proof.** Let \( n \) be the maximal number of nested lamdas in \( \kappa_1 \). We call depth of a binding the number of nested lamdas it is into (the depth of a binding of \( \kappa_1 \) is at most \( n \)).

Let \( N_\kappa(i) \) be the number of bindings of depth \( i \) in an expression \( \kappa \). Let \( S(\kappa) \) be the following n-uplet: \((N_\kappa(n), N_\kappa(n-1), \ldots, N_\kappa(0))\).

The chain \( S(\kappa_1), S(\kappa_2), \ldots \) is strictly decreasing with respects to the lexical order, thus it cannot be infinite. □

**Property I.26.** If \( \kappa \not\rightarrow \), then \( \kappa \) is an MSC-form.

**Proof.** Let’s assume we have \( \kappa \not\rightarrow \) and show that all 3 MSC properties are satisfied.

The property 3 (removing of unused bindings) is trivial: any binding that does not satisfy this property can directly be eliminated with the rule 23. As the rule 23 does not apply, this property must be satisfied.

Now, let’s focus on the property 2 (extrusion of bindings). We assume that there exists a subexpression \( \lambda x. \kappa_1 \) of \( \kappa \) and a subexpression \( \text{bind } y = a \in \kappa_2 \) of \( \kappa_1 \) such that \( \text{fv}(a) \subseteq \text{fv}(\lambda x. \kappa_1) \). We know that the definition \( a \) cannot depend on any variable defined inside \( \lambda x. \kappa_1 \) (including \( x \)), otherwise this variable would be in \( \text{fv}(a) \) and not in \( \text{fv}(\lambda x. \kappa_1) \). Thus, we can reorder the binding \( y \) (25) in the first position of its inner-most containing lambda-abstraction, and then apply the rule 24 on it, which contradicts \( \kappa \not\rightarrow \). Thus, the property 2 is satisfied.

Finally, let’s show that the property 1 (sharing of equivalent definitions) is satisfied too. We assume that there are two distinct bindings \( \text{bind } x_1 = a_1 \in \ldots \) and \( \text{bind } x_2 = a_2 \in \ldots \) such that \( a_1 \equiv_{\kappa} a_2 \). As the property 2 is satisfied, and as \( \text{fv}(a_1) = \text{fv}(a_2) \), we know that these two bindings are on the same level (i.e. their contexts cross exactly the same lambda-abstractions). Thus, we can reorder them (25) to be the one next to the other so that the rule 22 is applicable, which contradicts \( \kappa \not\rightarrow \). Thus, the property 1 is satisfied. □
I.2.2 Soundness. See Appendix G for the full algorithmic system, without the rules for extensions.

Lemma I.28. If \( \kappa \) is a canonical form where all the binding variables introduced are distinct, and if \( \Gamma \vdash_{\Pi} [\kappa \mid b] : t \), then the associated derivation does not need to perform any implicit alpha-renaming on binding variables.

Proof. Straightforward induction (note that it relies on the guardian \( x \notin \text{dom}(\Gamma) \) of the \([\text{BIND}_1-\text{Alg}]\) rule).

Thus, in the following, we assume that all the binding variables introduced by a canonical form are distinct, and that derivations of the algorithmic type system never perform implicit alpha-renaming on them.

Theorem I.29 (Soundness).
If \( \Gamma \vdash_{\Pi} [\kappa \mid b] : t \), then \( \Gamma \vdash [\kappa] : t \). If \( \Gamma \vdash [a \mid a] : t \), then \( \Gamma \vdash [a] : t \).

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

We consider the root of the derivation:

- **[CONST-Alg]** Trivial ([CONST] rule).
- **[Ax-Alg]** Trivial ([Ax₃] rule).
- **[→I-Alg]** We have \( a = \lambda x. \kappa \), and thus \( [a] = \lambda x. [\kappa] \).
  By induction on the premise, we get \( \Gamma, x : u \vdash [\kappa] : s \). By applying the rule \([→I]\), we get \( \Gamma \vdash [a] : u \rightarrow s \).
- **[→E-Alg]** We have \( a = x_1 x_2 \). We pose \( t_1 = \Gamma(x_1) \Sigma_1 \) and \( t_2 = \Gamma(x_2) \Sigma_2 \).
  With an [Ax₅] rule, we can derive \( \Gamma \vdash x_1 : \Gamma(x_1) \) and \( \Gamma \vdash x_2 : \Gamma(x_2) \). By using a [Inst∧≤] pattern, we can derive from that \( \Gamma \vdash x_1 : t_1 \) and \( \Gamma \vdash x_2 : t_2 \). We have \( t \equiv t_1 \circ t_2 \). Thus, according to the definition of \( \circ \), we know that \( t_1 \leq t_2 \rightarrow t \). Thus, with an application of \([≤]\) on \( \Gamma \vdash x_1 : t_1 \), we can derive \( \Gamma \vdash x_1 : t_2 \rightarrow t \). We can then conclude with an application of the rule \([→E]\).
- **[×I-Alg]** We have \( a = (x_1, x_2) \).
  With an [Ax₅] rule, we can derive \( \Gamma \vdash x_1 : \Gamma(x_1) \rho_1 \) and \( \Gamma \vdash x_2 : \Gamma(x_2) \rho_2 \) (with \( \rho_1 \) and \( \rho_2 \) as in the \([×I-Alg]\) rule). We can then conclude with an application of the rule \([×I]\).
- **[×E₁-Alg]** We have \( a = \pi_i x \). We pose \( t_0 = \Gamma(x) \Sigma \).
  With an [Ax₅] rule, we can derive \( \Gamma \vdash x : \Gamma(x) \).
  By using a [Inst∧≤] pattern, we can derive from that \( \Gamma \vdash x : t_0 \). We have \( t \equiv \pi_i t_0 \).
  Thus, according to the definition of \( \pi_i \), we can deduce that \( t_0 \leq t \times 1 \).
  Thus, with an application of \([≤]\), we can derive \( \Gamma \vdash x : t \times 1 \).
  We can then conclude with an application of the rule \([×E₁]\).
- **[×E₂-Alg]** Similar to the previous case.
- **[0-Alg]** Similar to the previous case.
- **[ε₁-Alg]** Similar to the previous case.
- **[ε₂-Alg]** Similar to the previous case.
- **[Var-Alg]** Trivial ([Ax₅] rule).

- **[BIND₁-Alg]** We have \( \kappa = \text{bind} x = a \in \kappa \) and thus \( [\kappa] = [\kappa] \{[a]/x\} \).
  By induction on the premise, we get \( \Gamma \vdash [\kappa] : t \).
  As \( x \notin \text{dom}(\Gamma) \), we know that this derivation does not contain any [Ax₅] rule applied on \( x \).
  We can thus easily transform it into a derivation \( \Gamma \vdash [\kappa] \{[a]/x\} : t \) just be replacing every occurrence of \( x \) by \( [a] \).
We have $\kappa = \text{bind} \ x = a \in \kappa_0$ and thus $[\kappa] = [\kappa_0]([a]/x)$.

By induction on the first premise, we get $\Gamma \vdash [a] : s$. For any $i \in I (I \neq \emptyset)$, we apply the induction hypothesis on the corresponding premise. It gives $\Gamma, x : s \land u_i \vdash [\kappa_0] : t_i$. With a $[\leq]$ rule, we can obtain $\Gamma, x : s \land u_i \vdash [\kappa_0] : t_i$ with $t = \bigvee_{i \in I} t_i$. Thus, we can conclude using a $[\lor]$ rule.

$[\land-\text{Alg}]$ Trivial induction on the premises.

I.2.3 Compl eteness. In the following, we fix a total order $\leq$ over the expressions, compatible with the subexpression order (as in I.1.2). For any expression $e$, it determines a unique MSC-form $\text{MSC}(e)$: independent consecutive bindings can be ordered depending on the order $\leq$ of their unwinding.

Note that all MSC-forms of a given expression are equivalent modulo $\equiv_\kappa$ (I.23), so the order $\leq$ is only a way to fix the order of independent bindings for convenience. As the order taken is arbitrary, the proofs below will work for any MSC-form.

**Lemma I.30 (Monotonicity).**

If $\Gamma \vdash_\kappa [k \mid b] : t$ and $\Gamma' \leq_\rho \Gamma$, then $\exists k', t' : \Gamma' \vdash_\kappa [k \mid k'] : t'$ with $t' \leq_\rho t$.

If $\Gamma \vdash_\kappa [a \mid a] : t$ and $\Gamma' \leq_\rho \Gamma$, then $\exists a', t' : \Gamma' \vdash_\kappa [a \mid a'] : t'$ with $t' \leq_\rho t$.

**Proof.** Straightforward induction on the derivation.

**Definition I.31 (Form derivation).** A form derivation is a derivation of the declarative system such that:

- It satisfies the properties of the normalisation lemma I.11,
- Either it does not contain any structural rule, or its root is a $[\lor]$ rule (or a $[\text{Inst} \land \leq]$ pattern with a $[\lor]$ rule as premise)

**Definition I.32 (Atom derivation).** An atom derivation is a derivation of the declarative system such that:

- It satisfies the properties of the normalisation lemma I.11,
- It contains at least one structural rule,
- It has no occurrence of a $[\lor]$ rule except in the subderivation of a $[\rightarrow I]$ rule,
- Its root is not a $[\text{Inst}]$ nor a $[\leq]$

**Definition I.33 (Atomic source expression).** We say that an expression of the source language is an atomic source expression if it has the following shape:

\[
\bar{e} \::=\ c \mid x \mid \lambda x.e \mid (x, x) \mid xx \mid \pi_\tau x \mid (x \in \tau) ? x : x
\]  

and such that, for the case $\lambda x.e$, all subexpressions of $e$ are either a binding variable or they contain a lambda variable that is not in $\text{fv}(\lambda x.e)$.

The variable $\bar{e}$ is used to range over atomic source expressions.

**Definition I.34.** For any atomic source expression $\bar{e}$, we define $\text{MSC}(\bar{e})$ as follows:

\[
\text{MSC}(\lambda x.e) = \lambda x.\text{MSC}(e)
\]

\[
\text{MSC}(\bar{e}) = \bar{e} \quad \text{for any } \bar{e} \text{ that is not a lambda}
\]

**Property I.35.** For any atomic source expression $\bar{e}$, $\text{bind} \ x = \text{MSC}(\bar{e}) \in x$ is a valid MSC-form.

**Proof.** The extrusion property is ensured by the conditions on lambda-expressions in the definition I.33.
Definition I.36 (Binding context). We call binding context (noted $C$) a canonical form ending with a hole:

$$
\text{Binding context } C := [] \mid \text{bind } x = \alpha \text{ in } k
$$

and we use the usual notation $C[\alpha]$ for denoting the canonical form obtained by replacing the hole in $C$ by $\kappa$.

When speaking of the set of (sub)expressions defined by a binding context $C$, it will denote the set of the expressions $[C[\alpha]]$ for any binding variable $x$ in the scope of the hole in $C$.

Definition I.37. We define the operator $[e]_C$ as follows:

$$[e]_C | = e$$

$$[e](\text{bind } x = \alpha \text{ in } C) = ([e]_C)[[\alpha]/x]$$

Definition I.38. We define the operator $e \setminus C$ as follows:

$$e \setminus [] = e$$

$$e \setminus (\text{bind } x = \alpha \text{ in } C) = (e[x/[\alpha]]) \setminus C$$

Property I.39. For any binding context $C$ and expression $e$, we have $[e \setminus C]_C \equiv_{\alpha} [e]_C$ and $([e]_C) \setminus C \equiv_{\alpha} e \setminus C$.

Proof. Straightforward.

Property I.40 (Decomposition of form derivations). If $\Gamma \vdash e : t$ is a form derivation with the root being a $\forall$ rule doing the substitution $e'(x/e_x/x)$, then there exists a binding context $C$ such that $\text{MSC}(e) \equiv_{\alpha} C[\text{bind } x = \text{MSC}(e_x \setminus C) \text{ in } \text{MSC}(e' \setminus C)]$.

Proof. First, we can easily deduce from the fact that our derivation $\Gamma \vdash e_x : s$ is a term derivation (as it is a form derivation) that the premise $\Gamma \vdash e_x : s$ is an atom derivation.

We know that $e_x$ appears in $e$ (as $e'$ contains $x$, see I.7). Thus, we can deduce that $\text{MSC}(e)$ contains a definition for an atom $a$ that unwinds to $e_x$. More formally, we know that there exists a binding context $C$, an atom $a$ and a canonical form $\kappa$ such that $\text{MSC}(e) \equiv_{\alpha} C[\text{bind } x = a \text{ in } k]$ with $[[\alpha]]_C \equiv_{\alpha} e_x$.

The expression $e_x$ could contain some subexpressions that are not binding variables and that have no occurrence of a lambda variable defined in $e_x$. Thus, these subexpressions must be defined through atomic bindings in $C$ (the unwindings of the corresponding definitions are necessarily smaller than $e_x$ by $\leq$ as they are subexpressions of $e_x$). The expression $e'$ could also contain some such subexpressions. The ones whose unwinding is smaller than $e_x$ according to $\leq$ must be defined through atomic bindings in $C$ too. No other expression should be defined in $C$ (and each of these subexpressions must be associated to a unique binding variable) or it would contradict the properties of MSC-forms.

Under the context $C$, the expression $e_x \setminus C$ unwinds to $e_x$ (I.39). Moreover, as $\Gamma \vdash e_x : s$ is an atom derivation, $e_x \setminus C$ must be an atomic source expression. Thus, we can deduce from I.35 that $\text{MSC}(e_x \setminus C)$ can be used in place of the atom $a$ without breaking any property of the MSC-form, and thus by unicity of the MSC-form we can conclude that $a \equiv_{\alpha} \text{MSC}(e_x \setminus C)$.

Similarly, the expression $e' \setminus (C[\text{bind } x = \text{MSC}(e_x \setminus C) \text{ in } []])$ unwinds to $e'(e_x/x)$ under the context $C[\text{bind } x = \text{MSC}(e_x \setminus C) \text{ in } []]$ (I.39). As the derivation is normalized, $e_x$ cannot be a subexpression of $e'$ (I.7), thus $e' \setminus C \equiv_{\alpha} e' \setminus (C[\text{bind } x = \text{MSC}(e_x \setminus C) \text{ in } []])$ and thus $e' \setminus C$ also unwinds to $e'(e_x/x)$. Thus, $\text{MSC}(e' \setminus C)$ can be used in place of $\kappa$ without breaking any property of the MSC-form (note that it only contains top-level bindings for expressions greater than $e_x$ by $\leq$, as the smaller ones have been put in $C$). By unicity of the MSC-form, we conclude that $\kappa \equiv_{\alpha} \text{MSC}(e' \setminus C)$, and thus $\text{MSC}(e) \equiv_{\alpha} C[\text{bind } x = \text{MSC}(e_x \setminus C) \text{ in } \text{MSC}(e' \setminus C)]$. □
Theorem I.41 (Completeness). If \( \Gamma \vdash e : t \) is a form derivation, then \( \exists k, t'. \Gamma \vdash_\sigma [\text{MSC}(e) \mid k] : t' \) with \( t' \leq_\rho t \).
If \( \Gamma \vdash \bar{e} : t \) is an atom derivation (with \( \bar{e} \) an atomic source expression), then \( \exists a, t'. \Gamma \vdash_\sigma [\text{MSC}(\bar{e}) \mid a] : t' \) with \( t' \leq_\rho t \).

Proof. We proceed by induction on the depth of \( \Gamma \vdash e : t \).

We consider the root of the derivation (the cases up to \([e_2]\) are for atom derivations, the cases after are for form derivations):

[\text{Const}] Trivial.

[\text{Ax}_1] Trivial.

[\rightarrow I] We have \( \bar{e} \equiv \lambda x. e \) and thus \( \text{MSC}(\bar{e}) \equiv_\alpha \lambda x. \text{MSC}(e) \).

The premise of a normalised \([\rightarrow I]\) rule must be a form derivation (see I.7). Thus, by induction on this premise, we get \( \Gamma, x : \text{MSC}(e) \vdash_\sigma k : t' \) (with \( t' \leq_\rho t \)). We can thus derive \( \Gamma \vdash_\sigma [\lambda x. \text{MSC}(e) \mid \lambda(u, k)] : u \rightarrow t' \) and we have \( u \rightarrow t' \leq_\rho u \rightarrow t \), which concludes this case.

[\rightarrow E] We have \( \bar{e} \equiv x_1 x_2 \) and thus \( \text{MSC}(\bar{e}) \equiv_\alpha x_1 x_2 \).

As our derivation is normalised, we know that the second premise, \( \Gamma \vdash x_2 : t_1 \), is a \([\text{Inst} \land \leq] \) pattern whose premise is a \([\text{Ax}_v] \), and with no \([\leq] \) rule. Thus, we know that there exists \( \Sigma_2 \) such that \( \Gamma(x_2) \Sigma_2 \leq t_1 \). Similarly, the first premise, \( \Gamma \vdash x_1 : t_1 \rightarrow t_2 \), is also a \([\text{Inst} \land \leq] \) pattern whose premise is a \([\text{Ax}_v] \) (with possibly a \([\leq] \) rule). Thus, we know that there exists \( \Sigma_1 \) such that \( \Gamma(x_1) \Sigma_1 \leq t_1 \rightarrow t_2 \).

Consequently, and by definition of \( \circ \), we know that \( (\Gamma(x_1) \Sigma_1) \circ (\Gamma(x_2) \Sigma_2) \leq t_2 \). We can thus derive \( \Gamma \vdash_\sigma [x_1 x_2 \mid \circ(\Sigma_1, \Sigma_2)] : t' \) (with \( t' \leq (\Gamma(x_1) \Sigma_1) \circ (\Gamma(x_2) \Sigma_2) \)) such that \( t' \leq t_2 \), which concludes this case.

[\times I] We have \( \bar{e} \equiv (x_1, x_2) \) and thus \( \text{MSC}(\bar{e}) \equiv_\alpha (x_1, x_2) \).

As our derivation is normalised, both premises can only be a \([\text{Ax}_v] \). Thus, we can deduce that there exists two renamings of polymorphic variables \( p_1 \) and \( p_2 \) such that \( \Gamma(x_1) p_1 \leq t_1 \) and \( \Gamma(x_2) p_2 \leq t_2 \). Thus, we can derive \( \Gamma \vdash_\sigma [(x_1, x_2) \mid (p_1, p_2)] : t_1 \times t_2 \).

[\times E_1] We have \( \bar{e} \equiv \pi_1 x \) and thus \( \text{MSC}(\bar{e}) \equiv_\alpha \pi_1 x \).

As our derivation is normalised, we know that the premise, \( \Gamma \vdash x : t_1 \times t_2 \), is a \([\text{Inst} \land \leq] \) pattern whose premise is a \([\text{Ax}_v] \). Thus, we know that there exists \( \Sigma \) such that \( \Gamma(x) \Sigma \leq t_1 \times t_2 \). Consequently, and by definition of \( \pi_1 \), we know that \( \pi_1(\Gamma(x) \Sigma) \leq t_1 \). We can thus derive \( \Gamma \vdash_\sigma [\pi_1 x \mid \pi_1(\Sigma)] : t' \) (with \( t' \leq \pi_1(\Gamma(x) \Sigma) \)) such that \( t' \leq t_1 \), which concludes this case.

[\times E_2] Similar to the previous case.

[0] We have \( \bar{e} \equiv (x \in \tau) \? x_1 : x_2 \) and thus \( \text{MSC}(\bar{e}) \equiv_\alpha (x \in \tau) \? x_1 : x_2 \).

As our derivation is normalised, we know that the premise, \( \Gamma \vdash x : 0 \), is a \([\text{Inst} \land \leq] \) pattern whose premise is a \([\text{Ax}_v] \), and with no \([\leq] \) rule. Thus, we know that there exists \( \Sigma \) such that \( \Gamma(x) \Sigma \leq 0 \).

We can thus derive \( \Gamma \vdash_\sigma [(x \in \tau) \? x_1 : x_2 \mid 0(\Sigma)] : 0 \).

[\epsilon_1] We have \( \bar{e} \equiv (x \in \tau) \? x_1 : x_2 \) and thus \( \text{MSC}(\bar{e}) \equiv_\alpha (x \in \tau) \? x_1 : x_2 \).

As our derivation is normalised, we know that the first premise, \( \Gamma \vdash x : \tau \), is a \([\text{Inst} \land \leq] \) pattern whose premise is a \([\text{Ax}_v] \). Thus, we know that there exists \( \Sigma \) such that \( \Gamma(x) \Sigma \leq \tau \).

Similarly, the second premise, \( \Gamma \vdash x_1 : t_1 \), can only be a \([\text{Ax}_v] \). Thus, we know that there exists a renaming of polymorphic variables \( \rho \) such that \( \Gamma(x_1) \rho \leq t_1 \).

We can thus derive \( \Gamma \vdash_\sigma [(x \in \tau) \? x_1 : x_2 \mid \epsilon_1(\Sigma)] : \Gamma(x_1) \) with \( \Gamma(x_1) \leq_\rho t_1 \).

[\epsilon_2] Similar to the previous case.

[\text{Ax}_\tau] Trivial.

[\leq] Straightforward induction.
[INST] Straightforward induction.

[∧] By induction on the premises, we get \( \forall i \in I. \Gamma \vdash_{\Pi} [MSC(e) \mid \emptyset_i] : t'_i \) with \( t'_i \leq_{p} t_i \). Thus, we can derive \( \Gamma \vdash_{\Pi} [MSC(e) \mid \bigwedge_{i \in I} \emptyset_i] : \bigwedge_{i \in I} t'_i \) (with \( \bigwedge_{i \in I} t'_i \leq_{p} \bigwedge_{i \in I} t_i \)).

[∨] By using I.40, we know that there exists \( C \) such that \( MSC(e) \equiv_{\alpha} C[b\text{ind}\ x = MSC(e_x \setminus C)] \) (with \( e_x \setminus C \) being an atomic source expression).

The unwinding of the definitions in \( C \) are necessarily smaller than \( e_x \). Thus, none of them can be defined by a \([\lor] \) rule in the current derivation for \( \Gamma \vdash e'\{e_x/x\} : t \). As a structural rule (or a \([\rightarrow\land] \) pattern) can only appear as the first premise of a \([\lor] \) rule (I.11), none of the expressions defined in \( C \) are typed in the current derivation. Thus, we can easily replace them, in the current derivation, by the associated binding variable in \( C \). It gives us a derivation for \( \Gamma \vdash (e'(e_x/x)) \setminus C : t \), or written differently, for \( \Gamma \vdash (e' \setminus C)\{(e_x \setminus C)/x\} : t \).

By induction on the premises of this new derivation, we get \( \Gamma \vdash_{\Pi} [MSC(e_x \setminus C) \mid \emptyset_a] : s' \) (with \( s' \leq_{p} s \)) and \( \forall i \in I. \Gamma, x : s \land u_i \vdash_{\Pi} [MSC(e' \setminus C) \mid \emptyset_i] : t_i \) (with \( t_i \leq_{p} t \)). By monotonicity (I.30), we can derive \( \forall i \in I. \Gamma, x : s' \land u_i \vdash_{\Pi} [MSC(e' \setminus C) \mid \emptyset_i] : t'_i \) (with \( t'_i \leq_{p} t_i \leq_{p} t \)). We can thus derive \( \Gamma \vdash_{\Pi} [b\text{ind}\ x = MSC(e_x \setminus C) \text{ in } MSC(e' \setminus C) \mid \emptyset] : \bigvee_{i \in I} t'_i \) with \( k = \text{keep } (\emptyset_a, \{(u_i, k_i')\}_{i \in I}) \).

From that, we can easily derive \( \Gamma \vdash [MSC(e) \mid \emptyset_k'] : \bigvee_{i \in I} t'_i \) with \( \emptyset_k' \) obtained by adding to the root of \( k \) a skip annotation for each definition in \( C \).

Property I.42. Any derivation \( \Gamma \vdash e : t \) can be transformed into a form derivation.

Proof. We first normalize \( \Gamma \vdash e : t \) using I.11. If the derivation we obtain is not a form derivation, that is, if it contains a structural rule and its root is not a \([\text{INST}\land\leq] \) pattern with a \([\lor] \) rule as premise, then we replace its root \( D \) (or, if its root is a \([\text{INST}\land\leq] \) pattern, the premise of this \([\text{INST}\land\leq] \) pattern) with:

\[
\frac{D}{[\lor]} \quad \frac{[Ax_{x'}]}{\Gamma \vdash e : t} \quad \frac{\Gamma, x : t \vdash x : t}{\Gamma \vdash e/x : t}
\]

It is straightforward to check that the derivation we obtain is a form derivation.

Corollary I.43 (Completeness). If \( \Gamma \vdash e : t \), then \( \exists \emptyset_k t' \). \( \Gamma \vdash_{\Pi} [MSC(e) \mid \emptyset] : t' \) with \( t' \leq_{p} t \).

Proof. Direct application of I.41 after using I.42 on \( \Gamma \vdash e : t \).

I.3 Annotations reconstruction system

In the following proofs, we assume that all the binding variables used in canonical forms are fresh (there is no conflict with another binding in the canonical form, nor with a binding variable already in the environment).

I.3.1 Termination.
Definition I.44. We define the successors of a result $\mathbb{R}$ as the set of pairs $\{(\Gamma_i, \mathcal{H}_i)\}_{i \in I}$ defined as follows:

\[
\text{succ}(\text{Ok}(\mathcal{H})) \overset{\text{def}}{=} \emptyset
\]
\[
\text{succ}(\text{Fail}) \overset{\text{def}}{=} \emptyset
\]
\[
\text{succ}(\text{Split}(\Gamma, \mathcal{H}_1, \mathcal{H}_2)) \overset{\text{def}}{=} \{(\Gamma, \mathcal{H}_1) \cup \{(x: -u), \mathcal{H}_2) \mid (x: u) \in \Gamma\}
\]
\[
\text{succ}(\text{Subst}(\psi_i)_{i \in I}, \mathcal{H}_1, \mathcal{H}_2) \overset{\text{def}}{=} \{(\emptyset, \mathcal{H}_1), (\emptyset, \mathcal{H}_2)\}
\]
\[
\text{succ}(\text{Var}(x, \mathcal{H}_1, \mathcal{H}_2)) \overset{\text{def}}{=} \{\{(x: 1), \mathcal{H}_1\}, (\emptyset, \mathcal{H}_2)\}
\]

Definition I.45. Let $\eta$ a canonical form or atom. Let $\Gamma_1, \Gamma_2$ two environments and $\mathcal{H}_1, \mathcal{H}_2$ two annotations.

We define the notion of derivation step for $\Gamma, \mathcal{H}$ and $\eta$ from the pair $(\Gamma_1, \mathcal{H}_1)$ to the pair $(\Gamma_2, \mathcal{H}_2)$ as follows: $(\Gamma_1, \mathcal{H}_1) \rightsquigarrow (\Gamma_2, \mathcal{H}_2) \Leftrightarrow \exists \psi, \eta. \exists \mathcal{H} \in \text{succ}(\mathbb{R})$ for some $\Gamma'' \geq \Gamma_2$. We also define the notion of derivation step for $\Gamma, \mathcal{H}$ and $\eta$ from the pair $(\Gamma_1, \mathcal{H}_1)$ to the pair $(\Gamma_2, \mathcal{H}_2)$ as follows: $(\Gamma_1, \mathcal{H}_1) \rightsquigarrow (\Gamma_2, \mathcal{H}_2) \Leftrightarrow \exists \psi, \eta. \exists \mathcal{H} \in \text{succ}(\mathbb{R})$ for some $\Gamma'' \geq \Gamma_2$. We use the notation $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$ for denoting a finite sequence of $\rightsquigarrow$ steps starting with $(\Gamma, \mathcal{H})$ and ending with $(\Gamma', \mathcal{H}')$. We also use the notation $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$ to denote the existence of a sequence of $\rightsquigarrow$ steps starting with $(\Gamma, \mathcal{H})$ that can be prolonged infinitely.

Lemma I.46. Let $\Gamma, \mathcal{H}, \Gamma'$ and $\mathcal{H}'$ such that $\exists \psi, \Gamma \leq \Gamma \psi$ and $\exists \psi, \mathcal{H} \equiv \mathcal{H}' \psi$. If $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$, then $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$.

Proof. Straightforward consequence of the definitions of $\rightsquigarrow$ and $\rightsquigarrow$. The first element of the chain can be replaced by $(\Gamma', \mathcal{H}')$ without having to change the next elements nor the derivations justifying each step.

Lemma I.47. If $(\Gamma, (\bigcup (\mathcal{H}_i)_{i \in I}, S)) \rightsquigarrow (\Gamma', \mathcal{H}')$, then $\exists i \in I. (\Gamma, \mathcal{H}_i) \rightsquigarrow (\Gamma', \mathcal{H}_i')$.

Proof. We know that in the chain $(\Gamma, (\bigcup (\mathcal{H}_i)_{i \in I}, S)) \rightsquigarrow (\Gamma', \mathcal{H}')$, all the derivations have as root $\text{Inter}_1$, $\text{Inter}_2$ or $\text{Inter}_3$ (because the result of $\text{InterOk}$ and $\text{InterEmpty}$ cannot have any successor). Thus, we know that there exists $i \in I$ such that we can find in our chain arbitrarily many derivations applying the rule $\text{Inter}_3$ on an annotation issued from the same initial $\mathcal{H}$. Let’s extract from our chain the sub-sequence associated to these derivations, and replace each of its pairs $(\Gamma'_i, \mathcal{H}'_i) \cup \{S_i, S_2\}$ (with $\mathcal{H}'$ the annotation issued from $\mathcal{H}$) with the pair $(\Gamma'_i, \mathcal{H}'_i)$. The sequence we obtain must start with $(\Gamma'_i, \mathcal{H}'_i)$ for some $\Gamma'$ such that $\exists \psi, \Gamma' \leq \Gamma' \psi$ and $\exists \psi, \mathcal{H}'_i \equiv \mathcal{H}' \psi$. Moreover, two consecutive pairs $(\Gamma'_i, \mathcal{H}'_i)$ and $(\Gamma'_{i+1}, \mathcal{H}'_{i+1})$ satisfy $(\Gamma'_i, \mathcal{H}'_i) \rightsquigarrow (\Gamma'_{i+1}, \mathcal{H}'_{i+1})$: it can be deduced from the premise of the associated $\text{Inter}_3$ rule. We thus have $(\Gamma'_i, \mathcal{H}'_i) \rightsquigarrow (\Gamma'_{i+1}, \mathcal{H}'_{i+1})$ and thus by I.46 $(\Gamma'_i, \mathcal{H}'_i) \rightsquigarrow (\Gamma'_{i+1}, \mathcal{H}'_{i+1})$.

Lemma I.48. If $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$, then $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$.

Proof. Let’s assume we have $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$ and let’s build a chain $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$. For that, we define a process that transforms an infinite chain, initially $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$, into some steps $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$ (of length $\geq 1$) and a new chain $(\Gamma', \mathcal{H}') \rightsquigarrow (\Gamma'', \mathcal{H}'')$. This process can be used iteratively to arbitrarily extend the $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$ chain. We consider the derivation associated with the first step $(\Gamma, \mathcal{H}) \rightsquigarrow (\Gamma', \mathcal{H}')$ of the input chain, by supposing without loss of generality that the associated derivation has a judgement
\[ \Gamma \vdash_R (\eta | \mathcal{H}) \Rightarrow \Re \text{ for some } \Re \text{ (we apply I.46 if it is not the case). We proceed by induction on this derivation:} \]

- If the root is [Stop], the premise gives \((\Gamma, \mathcal{H}) \sim_{\Re, \eta} (\Gamma'', \mathcal{H}'')\), and we return this together with the tail \((\Gamma'', \mathcal{H}'') \sim^*_{\Re, \eta}\) of the input chain.
- If the root is [Iterate1], the first premise gives \((\Gamma, \mathcal{H}) \sim_{\Re, \eta} (\Gamma, \mathcal{H}_1)\), and by induction on the second premise we get a chain \((\Gamma, \mathcal{H}_1) \sim^*_{\Re, \eta} (\Gamma', \mathcal{H}')\) and \((\Gamma', \mathcal{H}') \sim^\infty_{\Re, \eta}\) for some \(\Gamma'\) and \(\mathcal{H}'\).
- If the root is [Iterate2], the first premise gives \((\Gamma, \mathcal{H}) \sim_{\Re, \eta} (\Gamma, \mathcal{H}_1)\) and \((\Gamma, \mathcal{H}) \sim_{\Re, \eta} (\Gamma, \mathcal{H}_2)\). Let \(\mathcal{H}' = \land (\mathcal{H}_1\psi_i, \emptyset)\).

By induction on the last premise, we get a chain \((\Gamma, \mathcal{H}') \sim^*_{\Re, \eta} (\Gamma'', \mathcal{H}'')\) for some \(\Gamma''\) and \(\mathcal{H}''\) and a chain \((\Gamma'', \mathcal{H}'') \sim^\infty_{\Re, \eta}\). By concatenating these two chains and applying I.47 on the resulting chain, we know there exists a chain \((\Gamma, \mathcal{H}_i\psi_i) \sim^\infty_{\Re, \eta}\) for some \(i \in I\), or a chain \((\Gamma, \mathcal{H}_2) \sim^\infty_{\Re, \eta}\).

In the first case, we apply I.46 to get a chain \((\Gamma, \mathcal{H}_i) \sim^\infty_{\Re, \eta}\) (we have \(\psi_i\#\Gamma\)), and we return it with the chain \((\Gamma, \mathcal{H}) \sim^\infty_{\Re, \eta} (\Gamma, \mathcal{H}_i)\). In the other case, we directly return \((\Gamma, \mathcal{H}_2) \sim^\infty_{\Re, \eta}\) with \((\Gamma, \mathcal{H}) \sim^\infty_{\Re, \eta} (\Gamma, \mathcal{H}_2)\).

\[
\text{Lemma I.49. If } (\Gamma, \land \langle \{\mathcal{H}_i\}_{i \in I}, S \rangle) \sim^\infty_{\Re, \eta}, \text{ then } \exists i \in I. \ (\Gamma, \mathcal{H}_i) \sim^\infty_{\Re, \eta}.
\]

\[
\text{Proof. Combination of I.47 and I.48.}
\]

\[
\text{Lemma I.50. For any } \eta, \Gamma \text{ and } \mathcal{H}, \text{ there is no chain } (\Gamma, \mathcal{H}) \sim^\infty_{\Re, \eta}.
\]

\[
\text{Proof. We proceed by structural induction on } \eta.
\]

Let \((\Gamma, \mathcal{H}) \sim_{\Re, \eta} (\Gamma', \mathcal{H}')\) be the first step of a chain. We show, with another induction on the depth of the derivation associated with this first step, that the chain cannot be infinitely extended.

Let’s first consider the case of an atom \((\eta \equiv a \text{ and } \mathcal{H} \equiv \mathcal{A})\).

If \(\mathcal{A}\) is typ or untyp, only the rules [Ok] or [Fail] can apply, whose result has no successor.

If \(\mathcal{A}\) is an intersection annotation \(\land \langle \{\mathcal{A}_i\}_{i \in I}, S \rangle\), we get by induction that for any \(i \in I\) there is no chain \((\Gamma, \mathcal{A}_i) \sim^\infty_{\Re, a}\). By applying (the contrapositive of) I.49, we deduce that there is no chain \((\Gamma, \land \langle \{\mathcal{A}_i\}_{i \in I}, S \rangle) \sim^\infty_{\Re, a}\).

If \(\mathcal{A}\) is an annotation \(\lambda(u, \mathcal{K})\), then only [LambdaEmpty] or [Lambda] can apply for the first step. By the absurd, let’s suppose we have a chain \((\Gamma, \lambda(u, \mathcal{K})) \sim^\infty_{\Re, a}\). As the result of [LambdaEmpty] has no possible successor, the derivation of the first step of this chain must use a [Lambda] rule as root, and so must the derivations of the next steps. From the premises of these derivations, we can construct a chain \((\Gamma, \mathcal{K}) \sim^\infty_{\Re, a}\) and by applying I.48 we obtain a chain for \((\Gamma, \mathcal{K}) \sim^\infty_{\Re, a}\), which contradicts the induction hypotheses.

Otherwise, \(\mathcal{A}\) is an annotation \(\text{infer}\). Depending on \(a\):

- \(c\) Trivial (only [Const] applies)
- \(x\) Trivial (only [AxOk] and [AxFail] apply)
- \((x_1, x_2)\) There can be at most two derivations using [PairVar] in the chain, because for any of them, either its successor uses the annotation \(\text{untyp}\) or it is still using the annotation \(\text{infer}\) but with \(x_1 \in \text{dom}(\Gamma)\). The only remaining rule that can be applied is [PairOk], and it has no successor.
- \(\pi_{1}x\) There can be at most one derivation using [ProjVar] in the chain, because either its successor uses the annotation \(\text{untyp}\) or it is still using the annotation \(\text{infer}\) but with \(x \in \text{dom}(\Gamma)\). Moreover, all the possible successors for [ProjInfer] use the annotation \(\text{typ}\).
Similar to the previous case.

(\times_{e \in \tau} \ ? x_1 \times x_2) Similar to the previous case, using for the [CASESPLIT] rule the guarantee that its successors will either have $\Gamma(x) \trianglelefteq \tau$ or $\Gamma(x) \trianglelefteq \neg \tau$.

$\lambda x . \kappa$ The only rule applicable in this case is [LAMBDAINF], and we can easily conclude by induction on its premise.

Now, let’s consider the case of a canonical form ($\eta \equiv \kappa$ and $\mathcal{H} \equiv \mathcal{K}$).

If $\mathcal{K}$ is an intersection annotation, then we conclude in a similar way as for the intersection case of the atoms.

Otherwise, and if $\kappa$ is a variable $x$, then the annotation can only be typ, untyp or infer. Only the rules [OK], [FAIL], [FORMVAR] or [FORMOK] can apply, and we conclude trivially in each case.

Lastly, if $\kappa$ is a binding, let’s say $\text{bind } x = a \in \kappa'$, the annotation $\mathcal{K}$ can be one of those (the case for an intersection annotation has already been treated):

- **infer** We can conclude by using the induction hypotheses.
- **try-skip** ($\mathcal{K}_1$) We cannot have arbitrarily many derivations applying [BINDTRYSKIP] on $\kappa$ in the chain, because it would give, for some $\Gamma'$ and $\mathcal{K'}$, a chain $(\Gamma', \mathcal{K'}) \sim_{\tau, \mathcal{K'}}^\infty$ (I.48), which would contradict the induction hypotheses.

Thus, extending the chain infinitely would necessarily require to apply [BINDTRYSKIP] or [BINDTRYSKIP'] at some point, allowing us to conclude with the next cases.

- **try-keep** ($\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2$) This case is similar to the previous one.

- **skip** ($\mathcal{K}_1$) We cannot have arbitrarily many derivations applying [BINDSKIP] on $\kappa$ in the chain, because it would give, for some $\Gamma'$ and $\mathcal{K}'$, a chain $(\Gamma', \mathcal{K'}) \sim_{\tau, \mathcal{K'}}^\infty$ (I.48), which would contradict the induction hypotheses.

- **propagate** ($\mathcal{A}, \Gamma, S, S'$) There cannot be arbitrarily many consecutive derivations in the chain that can use [BINDPROP] on $\kappa$. Thus, extending the chain infinitely would necessarily require to apply [BINDPROP] at some point, allowing us to conclude with the next case.

- **keep** ($\mathcal{A}, S, S'$) Let’s suppose we have $(\Gamma, \text{keep } (\mathcal{A}, S, S')) \sim_{\tau, \mathcal{K}}^\infty$.

As there cannot be arbitrarily many consecutive derivations in the chain that can use [BINDPROP] or [BINDPROP'], we know that the chain features arbitrarily many [BINDSPLIT], [BINDSPLIT2] or [BINDSPLIT3] applied on $\kappa$. Also, arbitrarily many of them must apply to an annotation that is issued from the same initial annotation $\mathcal{K}' ((\_, \mathcal{K}') \in S)$ (by considering that when a split occurs, both new annotations are issued from the annotation that was splitted). We can thus extract from the chain the sub-sequence corresponding to the derivations that apply one of the [BINDSPLIT] rules on an annotation issued from $\mathcal{K}'$. From the premises of the derivations of this sub-sequence, we can extract a chain $(\Gamma', \mathcal{K'}) \sim_{\tau, \mathcal{K'}}^\infty$ for some $\Gamma'$, and thus using I.48 a chain $(\Gamma', \mathcal{K'}) \sim_{\tau, \mathcal{K'}}^\infty$, which contradicts the induction hypotheses.

\[\square\]

**Theorem 1.51 (Termination).** For any $\Gamma$, $\kappa$ and $\mathcal{K}$, applying the reconstruction deduction rules on the input $\Gamma \vdash_{\mathcal{R}} (\kappa \mid \mathcal{K})$ always terminate: either it gets stuck or it derives a judgement $\Gamma \vdash_{\mathcal{R}} (\kappa \mid \mathcal{K}) \Rightarrow \mathcal{R}$ for some $\mathcal{R}$.

**Proof.** All the recursive premises in the $\vdash_{\mathcal{R}}$ rules are either applied on a strict subexpression, or on the same expression but with a strictly decreasing annotation for some straightforward partial order. For the $\vdash^*_{\mathcal{R}}$ rules, we justify termination using lemma 1.50. \[\square\]