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 NGUYỄN, 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**; **Polymorphism**.

Additional Key Words and Phrases: polymorphism, union types, intersection types, type reconstruction.

ACM Reference Format:
Giuseppe Castagna, Mickaël Laurent, and Kim Nguyễn. 2024. Polymorphic Type Inference for Dynamic Languages: Reconstructing Types for Systems Combining Parametric, Ad-Hoc, and Subtyping Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 40 (January 2024), 32 pages. https://doi.org/10.1145/3632882

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) {
    if (x) { return x; } else { return y; }
}
```

A naive type for this function is $(\mathrm{Bool}, \mathrm{Bool}) \to \mathrm{Bool}$, which states that `lOr` 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 `lOR` 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 `lOr` the type $(\mathrm{Any}, \mathrm{Any}) \to \mathrm{Any}$ (where Any is the type of all values), which is a rather useless type since it essentially states that `lOr` is a binary function. To give `lOr` a more informative type, we need

[1]This definition does not capture the short-circuit evaluation of "`||`".

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, Université Paris-Saclay, CNRS, ENS Paris-Saclay, 91190, Gif-sur-Yvette, France.

ACM 2475-1421/2024/1-ART40
https://doi.org/10.1145/3632882

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 `lOr` the following more precise type

$$((\mathsf{Truthy}, \mathsf{Any}) \to \mathsf{Truthy}) \land ((\mathsf{Falsy}, \mathsf{Truthy}) \to \mathsf{Truthy}) \land ((\mathsf{Falsy}, \mathsf{Falsy}) \to \mathsf{Falsy}) \tag{1}$$

In this type, ∧ 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 `lOr` 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 not only it can check that `lOr` has the type in (1), but also it can reconstruct for `lOr` the intersection type $((\mathsf{Truthy}, \mathsf{Any}) \to \mathsf{Truthy}) \land ((\mathsf{Falsy}, \mathsf{Any}) \to \mathsf{Any})$ which, although it is less precise a type than (1), it is inferred from the code of `lOr` 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 step further, and show how to infer (i.e., reconstruct) intersections of *polymorphic* function types. In particular, the system we present here reconstructs for `lOr` the following first order polymorphic type (where α and β are type variables):[2]

$$\forall \alpha, \beta\,.\,((\alpha{\land}\mathsf{Truthy}, \mathsf{Any}) \to \alpha{\land}\mathsf{Truthy}) \land ((\mathsf{Falsy}, \beta) \to \beta) \tag{2}$$

This type completely specifies the semantics of the function `lOR`: it states that if the first argument is a Truthy, then the application of the function returns the first argument,[3] 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 `lOR` 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:

```
function id (x) {
    return lOR(x, x)
}
```

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

[3]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 (value result of the) argument.

then, as we explain later on, our system infers that `id` has type $\forall\alpha.\alpha \to \alpha$, 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 ECMAScript specifies the semantics of JavaScript logical operators, as defined in 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 Section 5). 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 `lOr` for ECMAScript is the following one:

```
7  function lOr (x, y) {
8      if (ToBoolean(x)) { return x; } else { return y; }
9  }
```

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 `lOr` 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 [$\wedge$]) and union elimination (rule [$\vee$]) given in Figure 2. For instance, the intersection introduction rule is used by the system to deduce that since the function `lOr` (either versions) has both type $((\alpha\wedge\mathsf{Truthy}, \mathsf{Any}) \to \alpha\wedge\mathsf{Truthy})$ and type $((\mathsf{Falsy}, \beta) \to \beta)$, then it has their intersection, too; this intersection type is then generalized (when `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 `id` in lines 4–6 has type $\alpha \to \alpha$, the system can assume that `x` has type α and separately infer the type of the body for `x` : $(\alpha\wedge\mathsf{Truthy})$ and for `x` : $(\alpha\wedge\neg\mathsf{Truthy})$; since the first deduction yields $(\alpha\wedge\mathsf{Truthy})$ and the second yields $(\alpha\wedge\neg\mathsf{Truthy})$, then the system deduces that under the hypothesis `x` : α, the body has the union of these two types, that is α. 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[4] 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. MSC-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

[4]A rule is *analytic* (as opposed to *synthetic*) when the input (i.e., Γ 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]).

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.

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 "`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 λ-terms, and the other that performs type narrowing when a typecase is encountered.

The first mechanism is inspired by the algorithm $\mathcal{W}$ 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 σ 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 σ 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 $s \wedge t$ and $s \wedge \neg t$, 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 (ToBoolean(x))...`" at line 8, it splits the type of (the variable bound to) `ToBoolean(x)` in two, by intersecting it with **`true`** and ¬**`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 known systems: first-order parametric polymorphism, intersection types, 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; Kfoury et al. 1993]. 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. We discuss in detail the reasons and implications of incompleteness in Section 4.4.

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 $\forall\alpha,\beta,\gamma.((\alpha\to\beta) \to ((\alpha\to\beta)\wedge\gamma)) \to ((\alpha\to\beta)\wedge\gamma))$ (i.e., in bounded polymorphic notation $\forall(\alpha)(\beta)(\gamma \leq \alpha\to\beta).((\alpha\to\beta) \to \gamma) \to \gamma$, cf. Footnote 2). 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 α, then the type inferred for (a curried version of) `fold_r` is $\forall\alpha,\beta,\gamma.((\alpha\to\beta\to\beta) \to \beta \to [\alpha*] \to \beta) \wedge (\mathsf{Any} \to \gamma \to [\,] \to \gamma)$ 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 $\forall\alpha,\beta,\gamma.((\alpha\wedge\beta\to\mathsf{Bool}) \wedge (\alpha\wedge\neg\beta\to\mathsf{False})) \to [\alpha*] \to [(\alpha\wedge\beta)*]$, stating that if we pass to `filter` a predicate that returns false for the elements of α that are not in β, then filtering a list of α's will return only elements also in β.

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, available on line as supplemental material [Castagna et al. 2024].

1.2 Discussion, Contributions, and Limitations

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 $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$. 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 retype-checking 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 $\forall\alpha.\alpha\to\alpha$ 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 suggests refining 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:

$$\lambda x.(x{\in}\mathsf{Int})\,?\,x : x \qquad\qquad \lambda x.(x{\in}\mathsf{Int})\,?\,x+1 : x$$

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 $\alpha \to \alpha$.[5] For the function on the right it returns the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\alpha{\setminus}\mathsf{Int} \to \alpha{\setminus}\mathsf{Int})$, where $s{\setminus}t$ denotes the set-theoretic difference of the two types, that is, $s{\wedge}\neg t$. These two types suggest how we can refine the intuitive characterization of parametricity. A generic function can inspect the parametric part of its input (as the function on the left-hand side shows) and its output can depend on this inspection (as the function on the right-hand side shows), but the parts of its output that are typed by a type variable—i.e., the "parametric" parts—cannot depend on it. We can speak of "partial" parametricity, and say that a function is parametric "only" for the inputs (or parts thereof) that are either returned unchanged or discarded: the type variables in its type describe such inputs. For instance, the domain of both the functions above is Any: they both can be applied to any argument. But the first function is parametric for all possible inputs, since the result of the inspection is not used to produce any particular output (it has type $\forall\alpha.\alpha \to \alpha$), 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 to generate the result for the integer inputs (by subsumption, the second function has type $\forall\alpha.\,\alpha \to \mathsf{Int}{\vee}(\alpha{\setminus}\mathsf{Int})$: parametricity holds 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 oncombine 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.

[5]Precisely, it deduces for it the type $(\alpha{\wedge}\mathsf{Int} \to \alpha{\wedge}\mathsf{Int}) \wedge (\beta{\setminus}\mathsf{Int} \to \beta{\setminus}\mathsf{Int})$. Instantiating β to α yields a subtype of $\alpha \to \alpha$.

(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 optional type annotations and pattern matching (presented in Appendix A) is available on-line at https://www.cduce.org/dynlang, and whose sources are on Zenodo: [Castagna et al. 2023b].

Limitations. The system we present here has some limitations. Foremost, the reconstruction algorithm of Section 4 uses backtracking, and at each of its passes it may try to type the same piece of code several times. Backtracking is inherent to our algorithm, since it proceeds by successively refining in different passes, the annotation tree of an MSC-form. The checking of a same piece of code several times at each pass is inherent to the use of unions and intersections: the union-elimination rule repeatedly type-checks the same expression, using different type hypotheses for a given sub-expression; the intersection-introduction rule verifies that an expression has all the types of an intersection, by checking each of them separately. Both features are very penalizing in terms of performance, and any naive implementation of the reconstruction described in Section 4 would yield type-inference times that grow exponentially with the size of the program. Clearly, this is an issue that must be addressed if we want to apply our system to real-world dynamic languages, and further work is needed to frame and/or constrain the current system so that its performance becomes acceptable. Fortunately, the room for improvement is significant: our prototype is an unoptimized proof of concept whose implementation was defined to faithfully simulate the reconstruction inference rules, rather than to obtain an efficient execution; but the simple addition of textbook memoization techniques improved its performance by an order of magnitude (cf. Section 5).

A second limitation of our system is that it is not sound in the presence of side-effects. The algorithm transforms an initial expression into its Maximal Sharing Canonical form, which is a list of bindings, one for each sub-expression of the initial expression. As we explain in Section 3.1.2, these forms are called "maximal sharing" since all equivalent sub-expressions (in the sense stated by Definition 3.1) of the initial expression must be bound by the same variable, so that any refinement of the type of one sub-expression (e.g., as a consequence of a type-case) is passed-through to all equivalent sub-expressions. However, this is sound only if all evaluations of equivalent sub-expressions return results that have the same types. While this is true for pure expressions, this can be invalidated by the presence of side-effects. In Section 7 we suggest some research directions on how to modify the equivalence relation of Definition 3.1 to make our system work in the presence of side-effects. Nevertheless, the work presented here is closer to be adapted/adaptable to pure functional languages such as Erlang and Elixir, than to languages such as JavaScript or Python.

Finally, it may be worth pointing out that our approach works only for strict languages, since it uses a semantic subtyping relation that is unsound for call-by-name evaluation strategies [Petrucciani et al. 2018].

2 SOURCE LANGUAGE AND TYPE SYSTEM

2.1 Syntax and Semantics

Our core language is fully defined in Figure 1. Expressions are an untyped λ-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 τ or e_2 otherwise. Type-cases cannot test arbitrary types but just ground types (i.e., types without type variables occurring in them) of the form τ where the only arrow type that can occur in them is $\mathbb{0} \to \mathbb{1}$, 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 $\mathsf{Int}\to\mathsf{Int}$ from those that do not.

Syntax

$$\begin{array}{lll} \textit{Test Type} & \tau ::= b \mid \mathbb{0} \to \mathbb{1} \mid \tau \times \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \\ \textit{Expression} & e ::= c \mid x \mid \lambda x.e \mid ee \\ & \quad \mid\ (e,e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e : e \\ \textit{Value} & v ::= c \mid \lambda x.e \mid (v,v) \\ \textit{Program} & p ::= \mathtt{let}\, x = e\,;\, p \ \mid\ e \end{array}$$

Reduction rules

$$\begin{array}{rcll} (\lambda x.e)v & \rightsquigarrow & e\{v/x\} & \\ \pi_1(v_1,v_2) & \rightsquigarrow & v_1 & \\ \pi_2(v_1,v_2) & \rightsquigarrow & v_2 & \\ (v{\in}\tau)\,?\,e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } v \in \tau \\ (v{\in}\tau)\,?\,e_1 : e_2 & \rightsquigarrow & e_2 & \text{if } v \in \neg\tau \\ \mathtt{let}\, x = v\,;\, p & \rightsquigarrow_{\mathsf{Pr}} & p\{v/x\} & \end{array}$$

Dynamic type test

$$v \in \tau \Leftrightarrow \mathrm{typeof}(v) \leq \tau, \text{ where } \begin{cases} \mathrm{typeof}(c) & = b_c \\ \mathrm{typeof}((v_1,v_2)) & = \mathrm{typeof}(v_1) \times \mathrm{typeof}(v_2) \\ \mathrm{typeof}(\lambda x.e) & = \mathbb{0} \to \mathbb{1} \end{cases}$$

Evaluation Contexts

$$\begin{array}{l} E ::= [\,] \mid Ee \mid vE \mid (E,e) \mid (v,E) \mid \pi_i E \mid (E{\in}\tau)\,?\,e : e \\ P ::= [\,] \mid \mathtt{let}\, x = [\,]\,;\, p \end{array} \qquad \frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']} \qquad \frac{e \rightsquigarrow e'}{P[e] \rightsquigarrow_{\mathsf{Pr}} P[e']}$$

Fig. 1. Syntax and semantics of the source language

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 λ-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 *value* is of a given type or not and holds true if and only if $\mathrm{typeof}(v) \leq \tau$, where $\leq$ is the subtyping relation defined by Castagna and Xu [2011] (we recall its definition in Appendix C). Note that $\mathrm{typeof}(v)$ maps every λ-abstraction to $\mathbb{0} \to \mathbb{1}$ and, thus, dynamic type tests do not depend on static type inference. 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* **Types** *is formed by the terms t coinductively produced by the grammar:*

$$\textbf{Types} \quad t, s \quad ::= \quad b \mid \alpha \mid \boldsymbol{\alpha} \mid t \to t \mid t \times t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

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 $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2)$, and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$. Basic types (e.g., Int, Bool) are ranged over by b, $\mathbb{0}$ and $\mathbb{1}$ 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 $t = t \vee t$ (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 α, 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 $\mathcal{V}_P$. *Monomorphic type variables*, ranged over by $\boldsymbol{\alpha}$ (with bold font), are variables that are not generalized and therefore cannot be instantiated; the set of monomorphic variables is $\mathcal{V}_M$. Types that only contain monomorphic variables are dubbed monomorphic types[6]:

$$\textbf{Monomorphic types}\quad \mathbf{u},\mathbf{v}\quad ::=\quad b \mid \boldsymbol{\alpha} \mid \mathbf{u}\to\mathbf{u} \mid \mathbf{u}\times\mathbf{u} \mid \mathbf{u}\vee\mathbf{u} \mid \neg\mathbf{u} \mid \mathbb{0}$$

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 α-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 π_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 variables) 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, $s \to t$ contains all λ-abstractions that when applied to a value of type s, if their computation terminates, then they return a result of type t (e.g., $\mathbb{0} \to \mathbb{1}$ is the set of all functions and $\mathbb{1} \to \mathbb{0}$ 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 σ. 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 [Const] and [Ax]. The arrow and product constructors have introduction and elimination rules. Notably, in the case of rule [$\to$I] the type of the argument is monomorphic. The rules for intersection ([$\wedge$]) and subtyping ([$\leq$]) are the classical ones, and so is the rule for instantiation ([Inst]) where σ denotes a substitution from polymorphic variables to types. The type-case construction is handled by three rules: [$\mathbb{0}$]; [$\in_1$]; [$\in_2$] . Rule [$\mathbb{0}$] 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

[6]The term *polytypes* and *monotypes* can be found (albeit inconsistently) in the literature: in particular, Milner [1978] uses the latter to denote types with no type variables and the former when he wishes to imply that a type may, or does, contain a variable. We avoided using them to prevent any confusion with our *monomorphic types*. While our *types* are indeed polytypes, our *monomorphic types* are not monotypes: monotypes do not have type variables while monomorphic types may have type variables, though only monomorphic ones. So we used instead *types* (which may have type variables), *ground types* (which cannot have any type variable), and *monomorphic types* (which may have monomorphic type variables, only).

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\text{Ax}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$[\to\text{I}]\ \frac{\Gamma, x : \mathbf{u} \vdash e : t}{\Gamma \vdash \lambda x.e : \mathbf{u} \to t} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

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

$$[\mathbb{0}]\ \frac{\Gamma \vdash e : \mathbb{0}}{\Gamma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : \mathbb{0}} \qquad [\in_1]\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg\tau \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge \mathbf{u} \vdash e : t \qquad \Gamma, x : s \wedge \neg\mathbf{u} \vdash e : t}{\Gamma \vdash e\{e'/x\} : t} \qquad [\wedge]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

$$[\textsc{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma} \qquad [\leq]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t'}\ t \leq t'$$

Fig. 2. Declarative Type System

to have either the type τ or its negation, in which case the corresponding branch is typed. These rules work together with Rule [$\vee$], which we describe in detail next.

At first sight, the formulation of rule [$\vee$] seems odd, since the $\vee$ connector does not appear in it. To understand it, consider the classic union elimination rule by MacQueen et al. [1986]:

$$[\vee\text{E}]\ \frac{\Gamma \vdash e' : s_1{\vee}s_2 \quad \Gamma, x : s_1 \vdash e : t \quad \Gamma, x : s_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

Rule [$\vee$E] types an expression that contains occurrences of an expression e' that has a union type $s_1 \vee s_2$; 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 $(fy{\in}\mathsf{Int})\ ?\ (fy) + 1 : \mathbf{false}$, in the context where f has type $\mathsf{Any}{\to}\mathsf{Any}$ and y is of type Any. This expression can be typed thanks to the rule [$\vee$E], by considering the sub-expression fy. This sub-expression has type Any, which can be seen as the union type $\mathsf{Any} \simeq \mathsf{Int}{\vee}\neg\mathsf{Int}$. We can then replace x for fy and type, using [$\in_1$], the expression $(x{\in}\mathsf{Int})\ ?\ x + 1 : \mathbf{false}$, with $x : \mathsf{Int}$. This yields a type Int (rule [$\in_1$] ignores the second branch) and by subtyping, the expression has type $\mathsf{Int}{\vee}\mathsf{False}$. Likewise for the choice $x : \neg\mathsf{Int}$, using rule [$\in_2$] the second branch has type False and therefore $\mathsf{Int}{\vee}\mathsf{False}$ (again via subtyping). The whole expression has thus the desired type $\mathsf{Int}{\vee}\mathsf{False}$.

A key element is that rule [$\vee$E] guessed how to split the type Any of fy into $\mathsf{Int} \vee \neg\mathsf{Int}$. 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, MacQueen et al.'s [$\vee$E] rule could choose to split, say, $\mathbb{1}$ into a union $\alpha \vee \neg\alpha$, with α a polymorphic type variable. If so, then the rule becomes unsound. As a matter of fact, the premises of the [$\vee$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 [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 between the two occurrences of the same variable α 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 [$\vee$] which requires the type s of e' to be split as $s \equiv (s \wedge \mathbf{u}) \vee (s \wedge \neg\mathbf{u})$ (here is our hidden union).

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

$$[\textsc{TopLevel-Expr}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash_{\mathsf{Pr}} e : t\phi}\ \phi\#\Gamma \qquad [\textsc{TopLevel-Let}]\ \frac{\Gamma \vdash_{\mathsf{Pr}} e : t \qquad \Gamma, x : t \vdash_{\mathsf{Pr}} p : t'}{\Gamma \vdash_{\mathsf{Pr}} \mathtt{let}\, x = e\ ;\ p\ : t'}$$

where ϕ denotes a generalization, that is a substitution transforming monomorphic variables into polymorphic ones and where $\phi\#\Gamma \overset{\text{def}}{\iff} \mathsf{dom}(\phi) \cap \mathsf{vars}(\Gamma) = \varnothing$.

After typing an expression used for a definition, its type is generalized (Rule [Toplevel-Expr]) before being added in the environment (Rule [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 $x : \alpha{\to}\alpha$ (with α polymorphic), then it is possible to obtain a closed program by inhabiting x by a definition like $\mathtt{let}\, x = \lambda y.y$;... ; e . This is the reason why, henceforth, we mainly focus on the typing of expressions.

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

Theorem 2.2 (Soundess). *If $\varnothing \vdash_{Pr} p : \tau$, then either p diverges or $p \rightsquigarrow_{Pr} v$ with $v \in \tau$.*

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 [$\vee$] 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 by defining the *canonical forms*, which are expressions produced by the following grammar:

$$\begin{array}{llll} \textbf{Atomic expressions} & a & ::= & c \mid x \mid \lambda x.\kappa \mid (\mathtt{x},\mathtt{x}) \mid \mathtt{xx} \mid \pi_i\mathtt{x} \mid (\mathtt{x}{\in}\tau)\,?\,\mathtt{x}:\mathtt{x} \\ \textbf{Canonical Forms} & \kappa & ::= & \mathtt{x} \mid \mathtt{bind}\,\mathtt{x} = a\,\mathtt{in}\,\kappa \end{array}$$

Canonical forms, ranged over by κ, 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 λ-abstraction (noted x, y, or z), or a constant, or a λ-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 $\lceil . \rceil$ that basically inlines bindings: $\lceil \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \rceil = \lceil \kappa \rceil \{\lceil a \rceil / \mathsf{x}\}$ (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 there 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 $\equiv_\kappa$ the smallest congruence on canonical forms and atoms that is closed by α-conversion and such that*

$$\mathsf{bind}\,\mathsf{x}_1 = a_1\,\mathsf{in}\,\mathsf{bind}\,\mathsf{x}_2 = a_2\,\mathsf{in}\,\kappa \;\equiv_\kappa\; \mathsf{bind}\,\mathsf{x}_2 = a_2\,\mathsf{in}\,\mathsf{bind}\,\mathsf{x}_1 = a_1\,\mathsf{in}\,\kappa \qquad \mathsf{x}_1 \notin \mathsf{fv}(a_2), \mathsf{x}_2 \notin \mathsf{fv}(a_1)$$

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 α-equivalent to) a canonical form κ such that:*
(1) if $\mathsf{bind}\,\mathsf{x}_1 = a_1\,\mathsf{in}\,\kappa_1$ and $\mathsf{bind}\,\mathsf{x}_2 = a_2\,\mathsf{in}\,\kappa_2$ are distinct sub-expressions of κ, then $a_1 \not\equiv_\kappa a_2$
(2) if $\lambda x.\kappa_1$ is a sub-expression of κ and $\mathsf{bind}\,\mathsf{y} = a\,\mathsf{in}\,\kappa_2$ a sub-expression of κ_1, then $\mathsf{fv}(a) \not\subseteq \mathsf{fv}(\lambda x.\kappa_1)$
(3) if $\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa'$ is a sub-expression of κ, then $\mathsf{x} \in \mathsf{fv}(\kappa')$.

MSC-forms are defined modulo α-conversion.[7] The first condition states that distinct variables denote different definitions, that is, it enforces the *maximal sharing* of common sub-expressions. The second condition requires bindings to extrude λ-abstractions whenever possible. The third condition states that there is no useless bind (bound variables must occur in the body of the binds).

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 κ_1 and κ_2 are two MSC-forms and $\lceil \kappa_1 \rceil \equiv_\alpha \lceil \kappa_2 \rceil$, then $\kappa_1 \equiv_\kappa \kappa_2$.*

We denote the unique MSC-form whose unwinding is e by $\mathsf{MSC}(e)$. 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 $\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa$ means (roughly) that it must be typed by applying the union rule to the expression $\kappa\{a/\mathsf{x}\}$. 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 [$\wedge$], [Inst], and [$\leq$] are still not syntax-directed, and (ii) rules [$\vee$], [Inst], [$\to$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 $\mathbf{u}$ in the premises), the [Inst] rule

[7] For instance, both $\lambda x.\mathsf{bind}\,\mathsf{x} = x\,\mathsf{in}\,\mathsf{bind}\,\mathsf{z} = \mathsf{xy}\,\mathsf{in}\,\mathsf{bind}\,\mathsf{z}' = \mathsf{zy}\,\mathsf{in}\,\mathsf{z}'$ and $\lambda x.\mathsf{bind}\,\mathsf{x} = x\,\mathsf{in}\,\mathsf{bind}\,\mathsf{z} = \mathsf{xy}\,\mathsf{in}\,\mathsf{z}$ are two distinct atoms that can occur in the same MSC-form, even though the atom xy appears in both: an α-renaming of x makes the first MSC-property hold.

requires guessing a substitution, the [→I] rule requires guessing the domain $\mathbf{u}$ of the function, and the [≤] rule requires guessing the type t' to subsume to.

The issue of [Inst] and [≤] 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 [≤] can be made analytic by using some type operators. As for rule [∧], making it syntax-directed, is trickier. Indeed, the usual approach of merging rules [→I] and [∧] does not work here, since terms in MSC-forms may hoist a bind definition outside the function where they are used, causing rule [∧] to be needed on a term that is not, syntactically, a λ-abstraction. Lastly, there is no easy way to guess the substitutions used by [Inst] rules, or the domain used in [→I] rules, or the decompositions performed by [∨] 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 [∨] rules) and which type substitutions (for [Inst] rules) to use. Formally, our algorithmic system uses judgements of the form $\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t$ for a canonical form κ, and $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$ for an atom a where $\mathbb{k}$ and $\mathbb{a}$ are respectively *form annotations* and *atom annotations* defined as follows:

$$\begin{array}{llll} \textbf{Atom annots} & \mathbb{a} & ::= & \varnothing \mid \lambda(\mathbf{u},\mathbb{k}) \mid (\rho,\rho) \mid @(\Sigma,\Sigma) \mid \pi(\Sigma) \mid \mathbb{0}(\Sigma) \mid \in_1(\Sigma) \mid \in_2(\Sigma) \mid \bigwedge(\{\mathbb{a},\dots,\mathbb{a}\}) \\ \textbf{Form annots} & \mathbb{k} & ::= & \rho \mid \mathtt{keep}\ (\mathbb{a},\{(\mathbf{u},\mathbb{k}),\dots,(\mathbf{u},\mathbb{k})\}) \mid \mathtt{skip}\ \mathbb{k} \mid \bigwedge(\{\mathbb{k},\dots,\mathbb{k}\}) \end{array}$$

where ρ ranges over *renamings* of polymorphic variables, that is, injective substitutions from $\mathcal{V}_P$ to $\mathcal{V}_P$, and Σ ranges over instantiations, that is, sets of substitutions from $\mathcal{V}_P$ to **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 $\varnothing$ annotation that is used both in the rule to type constants and in the two rules for variables.

$$[\to\text{I-Alg}]\ \frac{\Gamma, x : \mathbf{u} \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t}{\Gamma \vdash_{\mathcal{A}} [\lambda x.\kappa \mid \lambda(\mathbf{u},\mathbb{k})] : \mathbf{u} \to t}$$

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

$$[\to\text{E-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathtt{x}_1\mathtt{x}_2 \mid @(\Sigma_1,\Sigma_2)] : t_1 \circ t_2}\ \begin{array}{l} t_1 = \Gamma(\mathtt{x}_1)\Sigma_1,\ \ t_2 = \Gamma(\mathtt{x}_2)\Sigma_2 \\ t_1 \leq \mathbb{0} \to \mathbb{1},\ \ t_2 \leq \mathsf{dom}(t_1) \end{array}$$

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., Σ_1 and Σ_2) are sets of type substitutions; their application to a type t is defined as $t\Sigma \stackrel{\text{def}}{=} \bigwedge_{\sigma\in\Sigma} t\sigma$. Since they cannot be directly guessed, they are given by the annotation. Subsumption instead is embedded in two type operators. A first operator, $\mathsf{dom}()$, 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: $\mathsf{dom}(t) \stackrel{\text{def}}{=} \max\{u \mid t \leq u \to \mathbb{1}\}$ and $t \circ s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\}$.

$$[\times\text{E}_1\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_1\mathtt{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_1(t)}\ \begin{array}{l} t = \Gamma(\mathtt{x})\Sigma \\ t \leq (\mathbb{1}\times\mathbb{1}) \end{array} \qquad [\times\text{E}_2\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_2\mathtt{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_2(t)}\ \begin{array}{l} t = \Gamma(\mathtt{x})\Sigma \\ t \leq (\mathbb{1}\times\mathbb{1}) \end{array}$$

The rules for projections [$\times$E$_1$-Alg] and [$\times$E$_2$-Alg] follow the same idea as the rule for application [→E-Alg], with the use of two type operators $\boldsymbol{\pi}_1(t) \stackrel{\text{def}}{=} \min\{u \mid t \leq u\times\mathbb{1}\}$ and $\boldsymbol{\pi}_2(t) \stackrel{\text{def}}{=} \min\{u \mid t \leq \mathbb{1} \times u\}$. All these type operators can be effectively computed (cf. Appendix F).

$$[\times\text{I-Alg}]\ \frac{}{\Gamma \vdash_{\mathscr{A}} [(\mathsf{x}_1, \mathsf{x}_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2}\ t_1 = \Gamma(\mathsf{x}_1)\rho_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\rho_2$$

To type a pair $(\mathsf{x}_1, \mathsf{x}_2)$ it is not necessary to instantiate $\Gamma(\mathsf{x}_1)$ or $\Gamma(\mathsf{x}_2)$. 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 $\mathsf{x} : \alpha \to \alpha$, it is better to type it with $(\alpha \to \alpha, \beta \to \beta)$ rather than $(\alpha \to \alpha, \alpha \to \alpha)$, since the former type has strictly more instances than the latter.

$$[\in_1\text{-Alg}]\ \frac{}{\Gamma \vdash_{\mathscr{A}} [(\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_1(\Sigma)] : \Gamma(\mathsf{x}_1)}\ \Gamma(\mathsf{x})\Sigma \leq \tau$$

To type type-cases, the annotation indicates which of the three rules must be applied (here $[\in_1]$) 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 $[\in_2\text{-Alg}]$ and $[\mathbb{0}\text{-Alg}]$ in Appendix G).

$$[\text{Bind}_1\text{-Alg}]\ \frac{\Gamma \vdash_{\mathscr{A}} [\kappa \mid \Bbbk] : t}{\Gamma \vdash_{\mathscr{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{skip}\,\Bbbk] : t}\ \mathsf{x} \notin \mathsf{dom}(\Gamma)$$

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 κ under the current context Γ. For instance, this is the case when x only appears in a branch of a typecase that cannot be taken under Γ. The side condition $x \notin \Gamma$ prevents a potential unsound name conflict between binding variables: as occurrences of x in κ denote the x binding variable that is being skipped, having the type of a former binding variable x in our environment when typing κ would be unsound.

$$[\text{Bind}_2\text{-Alg}]\ \frac{\Gamma \vdash_{\mathscr{A}} [a \mid \mathbb{a}] : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathscr{A}} [\kappa \mid \Bbbk_i] : t_i}{\Gamma \vdash_{\mathscr{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\,(\mathbb{a}, \{(\mathbf{u}_i, \Bbbk_i)\}_{i \in I})] : \bigvee_{i \in I} t_i}\ \bigvee_{i \in I} \mathbf{u}_i \simeq \mathbb{1}$$

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 $[\vee]$ 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 $[\vee]$) whose union covers $\mathbb{1}$.

Finally, two annotations indicate when and how to apply rule $[\wedge]$ to atoms and canonical forms:

$$[\wedge\text{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathscr{A}} [a \mid \mathbb{a}_i] : t_i}{\Gamma \vdash_{\mathscr{A}} [a \mid \bigwedge(\{\mathbb{a}_i\}_{i \in I})] : \bigwedge_{i \in I} t_i}\ I \neq \varnothing \qquad [\wedge\text{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathscr{A}} [\kappa \mid \Bbbk_i] : t_i}{\Gamma \vdash_{\mathscr{A}} [\kappa \mid \bigwedge(\{\Bbbk_i\}_{i \in I})] : \bigwedge_{i \in I} t_i}\ I \neq \varnothing$$

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 term e of the source language*

$$\begin{array}{llll} \vdash e : t & \Rightarrow & \exists \Bbbk \vdash_{\mathscr{A}} [\mathsf{MSC}(e) \mid \Bbbk] : t' \leq t & \textit{(completeness)} \\ \vdash e : t & \Leftarrow & \vdash_{\mathscr{A}} [\mathsf{MSC}(e) \mid \Bbbk] : t & \textit{(soundness)} \end{array}$$

It is easy to generate the unique MSC-form associated to a 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.[8] Figure 3 gives an example of an MSC-form and two possible annotations for it. The term "$\lambda x.(fx{\in}\mathsf{Int})\,?\,g(fx) : x$" (where $f : \forall\alpha.\alpha{\to}\alpha$ and

[8]As stated by Theorem 3.4 the transformation of an expression into its MSC-form preserves typing. However, intuitively, it does not preserve the reduction semantics, since bindings regroup different occurrences of some sub-expression that in the original expression might be evaluated at different stages of the reduction or not evaluated at all. We said "intuitively" because no operational semantics is defined for canonical forms (this was alredy the case for [Castagna et al. 2022b]).

$\Gamma = \{f : \alpha \to \alpha,\ g : \mathsf{Int} \to \mathsf{Int}\}$

	MSC-form	First annotation	Second annotation	
1	bind z =	keep(	keep(	
2			$\bigwedge(\{$	
3	$\lambda x.$	$\lambda(\boldsymbol{\beta},$	$\lambda(\mathsf{Int},$	$\lambda(\boldsymbol{\beta}\setminus\mathsf{Int},$
4	bind x = x in	keep$(\varnothing, \{(\mathbb{1},$	keep$(\varnothing, \{(\mathbb{1},$	keep$(\varnothing, \{(\mathbb{1},$
5	bind u = f x in	keep$(@(\{\{\alpha \rightsquigarrow \boldsymbol{\beta}\}\}, \{\varnothing\}),$	keep$(@(\{\{\alpha \rightsquigarrow \mathsf{Int}\}\}, \{\varnothing\}),$	keep$(@(\{\{\alpha \rightsquigarrow \boldsymbol{\beta}\setminus\mathsf{Int}\}\}, \{\varnothing\}),$
6		$\{(\mathsf{Int},$	$\{(\mathbb{1},$	$\{(\mathbb{1},$
7	bind v = g u in	keep$(@(\{\varnothing\}, \{\varnothing\}), \{(\mathbb{1},$	keep$(@(\{\varnothing\}, \{\varnothing\}), \{(\mathbb{1},$	skip(
8	bind w =	keep $(\in_1(\{\varnothing\}), \{(\mathbb{1}, \varnothing)\}))\}),$	keep$(\in_1(\{\varnothing\}),$	keep$(\in_2(\{\varnothing\}),$
9	(u∈Int) ? v : x	$(\neg\mathsf{Int},$	$\{(\mathbb{1}, \varnothing)\}))\}))\}))\}$	$\{(\mathbb{1}, \varnothing)\}))\}))\}$
10	in w	skip(		
11		keep $(\in_2(\{\varnothing\}), \{(\mathbb{1}, \varnothing)\}))\}$		
12		$))\})),$	$\}),$	
13	in z	$\{(\mathbb{1}, \varnothing)\})$	$\{(\mathbb{1}, \varnothing)\})$	
	FINAL TYPE:	$\boldsymbol{\beta} \to \boldsymbol{\beta} \vee \mathsf{Int}$	$(\mathsf{Int} \to \mathsf{Int}) \wedge ((\boldsymbol{\beta}\setminus\mathsf{Int}) \to (\boldsymbol{\beta}\setminus\mathsf{Int}))$	

Fig. 3. MSC-form and two annotations for $\lambda x.(fx{\in}\mathsf{Int})\,?\,g(fx) : x$

$g : \mathsf{Int}{\to}\mathsf{Int}$) is put in MSC-form (on the left). In the first annotation, the function is typed with a single λ annotation (line 3). The interesting part is the annotation of the binding for u (line 5): the corresponding keep annotation represents an application of the union elimination rule on the occurrences of the expression fx whose type $\boldsymbol{\beta}$ is split into $\boldsymbol{\beta}{\wedge}\mathsf{Int}$ (line 6) and $\boldsymbol{\beta}{\setminus}\mathsf{Int}$ (line 9). Each subcase is annotated accordingly. Notice in the second subcase that the annotation for v is `skip` (line 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 $\neg\mathsf{Int}$). A different annotation, yielding a better type, is the one on the right. This intersection annotation (line 2) separates the domain of the λ-abstraction into two cases, each typed independently, yielding for the whole function an intersection type.

The example in Figure 3 also shows why the condition of maximal sharing for our forms is necessary, not only for their uniqueness, but also for the completeness of the algorithmic system: if the two occurrences of fx in "$\lambda x.(fx{\in}\mathsf{Int})\,?\,g(fx) : x$" were not bound by the same variable (as in the leftmost column of line 5 in Figure 3), viz., if the sharing were not maximal, then it would not be possible to deduce that $g(fx)$ is well typed: g expects an integer, but without maximal sharing it is not possible to deduce that the occurrence of fx in the first branch is indeed of type Int.

The problem of inferring an annotation for an MSC-form as the above—in particular the rightmost (more precise) annotation in Figure 3—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, it is sound and terminating (see Section 4.4 for the formal statements and a discussion about incompleteness). 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 λ-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 Σ 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 Δ representing the monomorphic type variables, find all substitutions σ whose domain is disjoint from Δ (noted $\sigma\#\Delta$) 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.[9] More precisely, all substitutions that are solutions to a tallying instance are characterized by a *principal* set Σ of substitutions, such that every $\sigma \in \Sigma$ is a solution, and for any solution σ, we have $\exists\sigma_1{\in}\Sigma.\ \exists\sigma_2.\ \sigma_2\#\Delta$ and $\sigma \simeq \sigma_2 \circ \sigma_1$, 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 $\mathsf{tally}(\{t_1 \dot{\leq} t_2\})$ the set of substitutions Σ characterizing all the solutions of the tallying instance $\{(t_1, t_2)\}$, where $\Delta = \mathcal{V}_M$ and thus $\forall\sigma \in \Sigma.\ \mathsf{dom}(\sigma) \subseteq \mathcal{V}_P$ (we use the symbol $\dot{\leq}$, rather than $\leq$ to stress that it denotes a constraint to be solved, rather than a pair in the subtyping relation).

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

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

$$\mathsf{tally\_infer}(\{t_1 \dot{\leq} t_2\}) = \{(\sigma \circ \sigma' \circ \phi)\big|_{\mathcal{V}_M} \mid \sigma' \in \mathsf{tally}(\{\mathsf{fresh}(t_1)\phi \dot{\leq} \mathsf{fresh}(t_2)\phi\})\}$$

where $\mathsf{fresh}(t)$ denotes the type t where polymorphic type variables have been substituted by fresh ones; ϕ is a renaming from $(\mathsf{vars}(t_1) \cup \mathsf{vars}(t_2)) \cap \mathcal{V}_M$ to fresh polymorphic variables; and σ is a substitution mapping polymorphic variables appearing in the image of $\sigma' \circ \phi$ 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 ϕ so that $\mathsf{tally}()$ is allowed to find solutions for them. Each solution σ' is composed with ϕ 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 σ with it. Finally, the substitution is restricted to $\mathcal{V}_M$ (i.e., to the domain of ϕ).

For example, an instance such as $\mathsf{tally\_infer}(\{\mathsf{Int}{\wedge}\alpha \to \mathsf{Int}{\wedge}\alpha \dot{\leq} \boldsymbol{\beta} \to \alpha\})$ can be generated during reconstruction, when a function of type $\mathsf{Int}{\wedge}\alpha \to \mathsf{Int}{\wedge}\alpha$ is applied to an argument of type $\boldsymbol{\beta}$, but the α on the right-hand of $\dot{\leq}$ is unrelated to the one on the left-hand side. Decorrelating them yields a unique solution $\{\boldsymbol{\beta} \rightsquigarrow \boldsymbol{\beta}'{\wedge}\mathsf{Int}\}$, that is, $\boldsymbol{\beta}$ must be substituted by $\boldsymbol{\beta}'{\wedge}\mathsf{Int}$ 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 λ-abstractions and the decompositions of types into disjoint unions to use for bindings. It works by successively

[9] This 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 $\{\alpha \rightsquigarrow s, \beta \rightsquigarrow t\}$, while all substitutions that make the former type become a subtype of the latter are characterized by a set containing three distinct substitutions: $\{\alpha \rightsquigarrow \mathbb{0}\}$, $\{\beta \rightsquigarrow \mathbb{0}\}$, and $\{\alpha \rightsquigarrow s, \beta \rightsquigarrow t\}$.

refining *intermediate annotations* defined below. These intermediate annotations store information about the domains of λ-abstractions and the decompositions of bindings. However, the instantiations Σ 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 Σ of later definitions. Thus, these instantiations Σ 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:

$$\begin{array}{llcl}
\textbf{Split annotations} & \mathcal{S} & ::= & \{(\mathbf{u},\mathcal{K}),\ldots,(\mathbf{u},\mathcal{K})\} \\
\textbf{Atom intermediate annot.} & \mathcal{A} & ::= & \texttt{infer} \mid \texttt{untyp} \mid \texttt{typ} \mid \bigwedge(\{\mathcal{A},\ldots,\mathcal{A}\},\{\mathcal{A},\ldots,\mathcal{A}\}) \\
& & & \mid \in_1 \mid \in_2 \mid \lambda(\mathbf{u},\mathcal{K}) \\
\textbf{Form intermediate annot.} & \mathcal{K} & ::= & \texttt{infer} \mid \texttt{untyp} \mid \texttt{typ} \mid \bigwedge(\{\mathcal{K},\ldots,\mathcal{K}\},\{\mathcal{K},\ldots,\mathcal{K}\}) \\
& & & \mid \texttt{try-skip}\ (\mathcal{K}) \mid \texttt{try-keep}\ (\mathcal{A},\mathcal{K},\mathcal{K}) \\
& & & \mid \texttt{propagate}\ (\mathcal{A},\mathbb{\Gamma},\mathcal{S},\mathcal{S}) \mid \texttt{skip}\ (\mathcal{K}) \mid \texttt{keep}\ (\mathcal{A},\mathcal{S},\mathcal{S})
\end{array}$$

In the following, we use the metavariable η to range over both atoms and expressions (i.e., $\eta ::= a \mid \kappa$). Similarly, the metavariable $\mathbb{h}$ ranges over atom annotations $\mathbb{a}$ and form annotations $\mathbb{k}$ (i.e., $\mathbb{h} ::= \mathbb{a} \mid \mathbb{k}$); while the metavariable $\mathcal{H}$ ranges over atom intermediate annotations $\mathcal{A}$ and form intermediate annotations $\mathcal{K}$ (i.e. $\mathcal{H} ::= \mathcal{A} \mid \mathcal{K}$).

Let ψ range over *monomorphic substitution*, that is, substitutions from $\mathcal{V}_M$ to monomorphic types, and Ψ 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 $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$, where $\mathbb{R}$ is a result defined as follows:

$$\textbf{Result} \quad \mathbb{R} \quad ::= \quad \texttt{Ok}(\mathcal{H}) \mid \texttt{Fail} \mid \texttt{Split}(\Gamma,\mathcal{H},\mathcal{H}) \mid \texttt{Subst}(\Psi,\mathcal{H},\mathcal{H}) \mid \texttt{Var}\ (\mathrm{x},\mathcal{H},\mathcal{H})$$

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

Ok$(\mathcal{H}')$**:** the reconstruction was successful and η can be typed by the algorithmic type system using the annotation $\mathcal{H}'$ (after converting it into an annotation $\mathbb{h}$ 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 η typable with the algorithmic system. This result is terminal.

Subst$(\Psi,\mathcal{H}_1,\mathcal{H}_2)$**:** the reconstruction found a set of substitutions Ψ that if applied to Γ may make η 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 Γ: η 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 Γ 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 $\mathrm{dom}(\Gamma')$ that if applied to Γ may make η typable. In practice, the system generates several new environments: one is obtained by (pointwise) intersecting Γ with Γ' and then it is used to retype η with the annotation $\mathcal{H}_1$; the others are obtained by intersecting Γ with all the possible pointwise negations of Γ' and then they are used to retype η with the annotation $\mathcal{H}_2$.

Var $(\mathbf{x},\mathcal{H}_1,\mathcal{H}_2)$**:** the reconstruction found that in order to type η, 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 η is annotated with $\mathtt{infer}$, and this annotation is then refined until it yields a terminal result (i.e., either $\mathtt{Ok}()$ or $\mathtt{Fail}$). 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: $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$ and $\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$. We first define rules for the judgment $\vdash_{\mathcal{R}}$ 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 $\vdash^*_{\mathcal{R}}$ judgments which call repetitively $\vdash_{\mathcal{R}}$ judgments when relevant, so that in the end we get a terminal result. Let us first focus on $\vdash_{\mathcal{R}}$ judgments.

$$[\textsc{Ok}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathtt{typ} \rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})} \qquad [\textsc{Fail}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathtt{untyp} \rangle \Rightarrow \mathtt{Fail}}$$

If a canonical form or atom η is annotated with $\mathtt{typ}$, then reconstruction is finished for η, and it is typeable in the current context Γ. The annotation $\mathtt{typ}$ is never used on λ-abstractions and bindings because the system needs to store more information for them. Likewise, if a form or atom η is annotated with $\mathtt{untyp}$, then reconstruction is finished for η by failing in the current context.

$$[\textsc{AxOk}]\ \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle x \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})} \qquad [\textsc{AxFail}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle x \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Fail}}$$

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

$$[\textsc{AppVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{infer}, \mathtt{untyp})}$$

To type the application $\mathsf{x}_1\mathsf{x}_2$, we must first ensure that $\{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma)$. If it is not the case, then the two rules [$\textsc{AppVar}_i$] (for $i = 1, 2$) try to remedy it by returning $\mathtt{Var}\ (\mathsf{x}_i, \mathtt{infer}, \mathtt{untyp})$, which is the result that asks the system to try to type the atom bound to x_i for $\mathsf{x}_i \notin \mathsf{dom}(\Gamma)$. If the attempt is successful, then the algorithm will continue the reconstruction for the application with the annotation $\mathtt{infer}$ and $\mathsf{x}_i \in \mathsf{dom}(\Gamma)$, otherwise it will continue with the annotation $\mathtt{untyp}$ making the reconstruction fail on this application.

$$[\textsc{AppInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\{\Gamma(\mathsf{x}_1) \mathrel{\dot{\leq}} \Gamma(\mathsf{x}_2) \to \alpha\})}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}\ \alpha \in \mathcal{V}_P \text{ fresh}$$

If $\{\mathsf{x}_1, \mathsf{x}_2\} \subseteq \mathsf{dom}(\Gamma)$, then the rule [$\textsc{AppInfer}$] tries to find all instances of the current context in which the application $\mathsf{x}_1\mathsf{x}_2$ is typeable, by subsuming $\Gamma(\mathsf{x}_1)$ (the type of the function) to $\Gamma(\mathsf{x}_2){\to}\alpha$ (a function type whose domain is the type of the argument). For that, it calls the tallying algorithm which returns a set of substitutions Ψ. Then, $\mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})$ is returned, meaning that this application should be typeable under every instance $\Gamma\psi$ of the current context Γ (with $\psi \in \Psi$). The default case (i.e., when the current context is unchanged, for example, when $\Psi = \varnothing$) cannot be typed, so it is annotated with $\mathtt{untyp}$ (see rule [$\textsc{Iterate}_2$] later on). The rules for pairs are similar and have been omitted.

$$[\textsc{CaseSplit}]\ \frac{\Gamma(\mathsf{x}) \not\leq \tau \qquad \Gamma(\mathsf{x}) \not\leq \neg\tau}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Split}(\{(\mathsf{x} : \tau)\}, \mathtt{infer}, \mathtt{infer})}$$

The key rule for type-cases is [$\textsc{CaseSplit}$], corresponding to the case where x is in Γ, 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 τ and the other a subtype of $\neg\tau$. This is achieved by returning

$\mathtt{Split}(\{(\mathsf{x}:\tau)\},\mathtt{infer},\mathtt{infer})$: this result is backtracked up to the binding of x, where it will split the associated type, accordingly.

$$[\textsc{CaseThen}]\ \frac{\Gamma(\mathsf{x}) \leq \tau \qquad \Psi = \mathsf{tally\_infer}(\{\Gamma(\mathsf{x}) \dot{\leq} \mathbb{0}\})}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \in_1)}$$

$$[\textsc{CaseElse}]\ \frac{\Gamma(\mathsf{x}) \leq \neg\tau \qquad \Psi = \mathsf{tally\_infer}(\{\Gamma(\mathsf{x}) \dot{\leq} \mathbb{0}\})}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \in_2)}$$

$$[\textsc{CaseVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_i\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{typ}, \mathtt{untyp})}$$

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 $\Gamma(\mathsf{x}) \leq \tau$, then we have to determine whether we will apply the algorithmic rule [$\mathbb{0}$-Alg] or the algorithmic rule [$\in_1$-Alg]. To determine it, the [CaseThen] rule calls $\mathsf{tally\_infer}(\{\Gamma(\mathsf{x}) \dot{\leq} \mathbb{0}\})$ which returns the set of contexts $\Gamma\psi$ (for $\psi \in \Psi$) under which the algorithmic rule [$\mathbb{0}$-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 $\Gamma(\mathsf{x})$ is not guaranteed to be empty and, thus, in which the algorithmic rule [$\in_1$-Alg] must be applied, is annotated with $\in_1$. This annotation is handled by the rule [$\textsc{CaseVar}_1$] which forces the system to type x_1, the binding variable associated to the first branch. The case for [CaseElse] and [$\textsc{CaseVar}_2$] is analogous.

We omitted the remaining rules for type-cases since they are straightforward: the rule for $\mathsf{x} \notin \mathsf{dom}(\Gamma)$, which triggers a $\mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})$ result; two rules similar to [$\textsc{CaseVar}_i$], but where $\mathsf{x}_i \in \mathsf{dom}(\Gamma)$, which simply return $\mathtt{Ok}(\mathtt{typ})$.

$$[\textsc{LambdaInfer}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\boldsymbol{\alpha}, \mathtt{infer})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \mathtt{infer}\rangle \Rightarrow \mathbb{R}}\ \boldsymbol{\alpha} \in \mathcal{V}_M \text{ fresh}$$

$$[\textsc{Lambda}]\ \frac{\Gamma, x : \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K})\rangle \Rightarrow \mathtt{map}(X \mapsto \lambda(\mathbf{u}, X),\ \mathbb{R})}$$

The rules for λ-abstractions mimic algorithm $\mathcal{W}$. Rule [LambdaInfer] transforms the initial `infer` annotation into a $\lambda(\boldsymbol{\alpha}, \mathtt{infer})$ annotation. As in $\mathcal{W}$, λ-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 tally_infer() will transform it into a polymorphic—thus, instantiable—one, just for the reconstruction in the body. Rule [Lambda] adds the λ-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 $\mathtt{map}(X{\mapsto}f(X),\ \mathbb{R})$ denotes the result $\mathbb{R}$ where f has been applied to every annotation X.

$$[\textsc{BindInfer}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \mid \mathtt{try\text{-}skip}\ (\mathtt{infer})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \mid \mathtt{infer}\rangle \Rightarrow \mathbb{R}}$$

The [BindInfer] rule transforms an initial `infer` annotation into a `try-skip (infer)` annotation which skips the binding and annotates the body κ with `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 [BindTrySkip$_1$] rule below:

$$[\textsc{BindTrySkip}_1]\ \frac{\begin{array}{c}\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2) \\ \Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathtt{infer}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathbb{R}}$$

This rule tries to type the body of the binding, starting with the annotation $\mathcal{K}$ (initially, `infer`). If the result is $\mathtt{Var}\ (\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2)$, then it means that the current binding is used in the body κ and, thus, the system should try to type it. Consequently, the annotation for the current binding is changed into a $\mathtt{try\text{-}keep}\ (\mathtt{infer}, \mathcal{K}_1, \mathcal{K}_2)$ so that, at the next iteration, its definition will be reconstructed.

If typing the body of the binding yields a result different from $\mathtt{Var}\ (\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2)$, then this result is just propagated as in [Lambda] (the corresponding rules have been omitted).

$$[\textsc{BindTryKeep}_1]\ \frac{\begin{array}{c}\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathtt{Ok}(\mathcal{A}') \\ \Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing) \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindTryKeep}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathtt{Fail} \qquad \Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{skip}\ (\mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}$$

As expected, if the current annotation for the binding is a $\mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)$, 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 [BindTryKeep$_1$] does by changing the current annotation to $\mathtt{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing)$ (more details below). If the reconstruction of the definition fails (rule [BindTryKeep$_2$]), 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 $\mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$ 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 $\mathtt{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing)$ used in rule [BindTryKeep$_1$] means that the type of the definition does not need to be partitioned: there is only one part, covering $\mathbb{1}$, associated with an annotation $\mathcal{K}_1$ for typing the body.

$$[\textsc{BindOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \varnothing, \mathcal{S}) \rangle \Rightarrow \mathtt{Ok}(\mathtt{keep}\ (\mathcal{A}, \varnothing, \mathcal{S}))}$$

If all the parts of the type decomposition have already been explored (i.e., $\varnothing$ in the annotation in the rule above), then the reconstruction is successful. Otherwise, the following rules are applied:

$$[\textsc{BindKeep}_1]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Ok}(\mathcal{K}') \\ \Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \{(\mathbf{u}, \mathcal{K}')\} \cup \mathcal{S}') \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{BindKeep}_2]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \\ \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Split}(\Gamma', \mathcal{K}_1, \mathcal{K}_2) \\ \mathsf{x} \in \mathrm{dom}(\Gamma') \qquad \Gamma \vdash_{\mathcal{E}} (a : \neg(\mathbf{u} \wedge \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_1 \qquad \Gamma \vdash_{\mathcal{E}} (a : \neg(\mathbf{u} \setminus \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_2\end{array}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathsf{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathtt{Split}(\Gamma' \setminus \mathsf{x}, \mathcal{K}'_1, \mathcal{K}'_2)}$$

with, in the last rule, $\mathcal{K}'_1 = \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}_1 \cup \mathbb{\Gamma}_2, \{(\mathbf{u} \wedge \Gamma'(\mathsf{x}), \mathcal{K}_1),\ (\mathbf{u} \setminus \Gamma'(\mathsf{x}), \mathcal{K}_2)\} \cup \mathcal{S}, \mathcal{S}')$ and $\mathcal{K}'_2 = \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K}_2)\} \cup \mathcal{S}, \mathcal{S}')$

In both rules, the definition of the binding is typed using the annotation $\mathcal{A}$. For that, it is first converted into an annotation $\mathbb{a}$ of the algorithmic type system, using the deduction rules for the judgment $\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$, defined in Section 4.3. Then, the type s obtained for the definition is intersected with one of the parts of the type decomposition, according to the second argument of the keep() annotation (i.e., $\{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}$ in both rules), and the corresponding annotation for the body is reconstructed recursively. Note that, since split annotations are sets, then the order in which the parts are explored is arbitrary.

The rule [BindKeep$_1$] for an annotation 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 Ok(). If instead the reconstruction of the body requires to further split the type of x, then the rule [BindKeep$_2$] 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 `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 True (since x and y are always bound to the same value):

$$\texttt{bind}\,\texttt{x} = x\ \texttt{in}\,\texttt{bind}\,\texttt{y} = \texttt{id}\ \ \texttt{x}\,\texttt{in}\,\texttt{bind}\,\texttt{z} = (\texttt{y}{\in}\text{True})\ ?\ \texttt{x}\,\texttt{:}\,\texttt{true}\ \texttt{in}\,\texttt{z}$$

At some point, the partition associated to y will change from $\{\mathbb{1}\}$ to $\{\text{True}, \neg\text{True}\}$ because of the type-case (rule [CaseSplit]). However, if the case corresponding to $(\texttt{y} : \text{True})$ is immediately explored, it will yield for the body the type Bool, because x still has the type Bool in the environment. In order to obtain the more precise type True, we must deduce, before exploring the case $(\texttt{y} : \text{True})$, that when `id x` (the definition of y) has type True, then x also has type 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 [BindKeep$_2$] rule with the two premises $\Gamma \vdash_{\mathcal{E}} (a : \neg(\mathbf{u} \wedge \Gamma'(\mathrm{x}))) \Rightarrow \mathbb{\Gamma}_1$ and $\Gamma \vdash_{\mathcal{E}} (a : \neg(\mathbf{u} \setminus \Gamma'(\mathrm{x}))) \Rightarrow \mathbb{\Gamma}_2$. This auxiliary judgment $\Gamma \vdash_{\mathcal{E}} (a : \mathbf{u}) \Rightarrow \mathbb{\Gamma}$, defined in Appendix H.3, can be read as follows: *refining the current environment* Γ *with one of the* $\Gamma' \in \mathbb{\Gamma}$ *ensures that the atom* a *will have type* $\mathbf{u}$. The refinements we obtain are stored in the annotation of the binding, using an annotation propagate $(\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S}')$. This annotation is handled by two other rules (omitted here) whose role is to propagate these refinements one after the other using successive $\texttt{Split}(\Gamma', \mathcal{K}_1, \mathcal{K}_2)$ results (with $\Gamma' \in \mathbb{\Gamma}$), before finally restoring a keep $(\mathcal{A}, \mathcal{S}, \mathcal{S}')$ annotation.

$$[\textsc{InterEmpty}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\varnothing, \varnothing) \rangle \Rightarrow \texttt{Fail}} \qquad [\textsc{InterOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\varnothing, S) \rangle \Rightarrow \texttt{Ok}(\bigwedge(\varnothing, S))}$$

$$[\textsc{Inter}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \texttt{Ok}(\mathcal{H}') \qquad \Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(S, \{\mathcal{H}'\} \cup S') \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}\} \cup S, S') \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Inter}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \texttt{Fail} \qquad \Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(S, S') \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}\} \cup S, S') \rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Inter}_3]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}\} \cup S, S') \rangle \Rightarrow \texttt{map}(X \mapsto (\bigwedge(\{X\} \cup S, S')),\ \mathbb{R})}$$

Intersection annotations are introduced by the $\vdash^*_{\mathcal{R}}$ judgments defined below. In an intersection annotation $\bigwedge(S, S')$, the annotations in S' are fully processed (i.e., the associated reconstruction returned Ok()), while the annotations in S are not: they still have to be refined one after the other (rule [Inter$_3$]). If one of them becomes fully processed, it is moved in S' (rule [Inter$_1$]). Conversely, if one of them fails, it is removed (rule [Inter$_2$]). The process stops when S is empty: then, the reconstruction fails if S' is empty (rule [InterEmpty]), and succeed otherwise (rule [InterOk]).

Finally, we formalize the rules for the judgments $\vdash^*_{\mathcal{R}}$. As said earlier, the purpose of $\vdash^*_{\mathcal{R}}$ is to repeatedly call $\vdash_{\mathcal{R}}$ judgments so that, in the end, we obtain a terminal result.

$$[\textsc{Iterate}_1]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \texttt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2) \qquad \Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H}_1 \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}\ \Gamma' = \varnothing$$

$$[\textsc{Iterate}_2]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \texttt{Subst}(\{\psi_i\}_{i\in I}, \mathcal{H}_1, \mathcal{H}_2) \\ \Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \bigwedge(\{\mathcal{H}_1\psi_i\}_{i\in I} \cup \{\mathcal{H}_2\}, \varnothing) \rangle \Rightarrow \mathbb{R}\end{array}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}\ \forall i \in I.\ \psi_i \# \Gamma$$

The iteration continues as long as it yields non-terminal results that are immediately usable, that is, either they return a trivial split (i.e., $\Gamma' = \varnothing$) as in rule [$\textsc{Iterate}_1$], or they return substitutions that do not affect the current environment (i.e., $\psi_i\#\Gamma$) as in rule [$\textsc{Iterate}_2$]. For the latter rule, the iteration may need to introduce an intersection annotation (useless when I is empty) in order to explore all the cases of a $\texttt{Subst}(\{\psi_i\}_{i\in I}, \mathcal{H}_1, \mathcal{H}_2)$ result (where $\mathcal{H}\psi$ is the intermediate annotation $\mathcal{H}$ in which the substitution ψ has been applied recursively to every type in it). An important special case of the [$\textsc{Iterate}_2$] rule is when $I = \varnothing$: in that case the iteration continues by trying to type η with the default annotation $\mathcal{H}_2$ and the current environment Γ. For instance, this special case triggers a [$\textsc{CaseVar}_1$] after a [$\textsc{CaseThen}$] and a [$\textsc{CaseVar}_2$] after a [$\textsc{CaseElse}$].

If the result is already terminal or if it is not immediately usable, then it is directly returned:

$$[\textsc{Stop}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}}$$

In particular, if $\mathbb{R} = \texttt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2)$ where $\Gamma' \neq \varnothing$ (i.e., [$\textsc{Iterate}_1$] does not apply), then [$\textsc{Stop}$] backtracks until Γ' becomes empty; likewise if $\mathbb{R} = \texttt{Subst}(\{\psi_i\}_{i\in I}, \mathcal{H}_1, \mathcal{H}_2)$ and $\Gamma\psi_i \not\simeq \Gamma$ for some i (i.e. [$\textsc{Iterate}_2$] does not apply), then [$\textsc{Stop}$] backtracks until it exits the scope of the binders of the variables that make the side condition of [$\textsc{Iterate}_2$] fail.

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 Σ needed to type the atoms.

Formally, the algorithm takes as input an environment Γ, an atom or canonical form η, and an intermediate annotation $\mathcal{H}$, and produces an annotation $\mathbb{h}$ for the algorithmic type system. It is presented as a deduction rule system for judgments of the form $\Gamma \vdash_{\mathcal{P}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{h}$. 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 $\texttt{typ}$ into an annotation $\varnothing$ for the algorithmic type system; likewise, the rules for λ-abstractions 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:

$$[\textsc{App}]\ \frac{\begin{array}{c} t_1 = \Gamma(\mathsf{x}_1) \qquad t_2 = \Gamma(\mathsf{x}_2) \\ \rho_1 = \texttt{refresh}(t_1) \qquad \rho_2 = \texttt{refresh}(t_2) \qquad \Sigma = \mathsf{tally}(\{t_1\rho_1 \dot{\leq} t_2\rho_2 \to \alpha\})\end{array}}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{x}_1\mathsf{x}_2 \mid \texttt{typ} \rangle \Rightarrow @(\{\sigma \circ \rho_1 \mid \sigma \in \Sigma\}, \{\sigma \circ \rho_2 \mid \sigma \in \Sigma\})}\ \begin{array}{l}\Sigma \neq \varnothing \\ \alpha \in \mathcal{V}_P \text{ fresh}\end{array}$$

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

For applications, an annotation of the form $@(\Sigma_1, \Sigma_2)$ must be produced. In order to find some instantiations Σ_1 and Σ_2 (for x_1 and x_2 respectively) that make the application typable, the [$\textsc{App}$] rule solves the tallying instance $\mathsf{tally}(\{t_1\rho_1 \dot{\leq} t_2\rho_2 \to \alpha\})$. The purpose of ρ_1 and ρ_2 is to decorrelate type variables in $\Gamma(\mathsf{x}_1)$ and in $\Gamma(\mathsf{x}_2)$. For instance, assume we want to reconstruct the instantiations for the atom "x x" with $\Gamma(\mathtt{x}) = \beta \to \beta$. The tallying instance $\mathsf{tally}(\{\beta \to \beta \dot{\leq} (\beta \to \beta) \to \alpha\})$ yields only a very specific, uninteresting solution (i.e., $\alpha = \beta = \mu X.X \to X$)[10] because of the use of the same type variable β on both sides of $\dot{\leq}$. But each occurrence of x has a polymorphic type that can be instantiated independently. Thus, we remove this useless and constraining dependency by refreshing the generic type variables yielding $\mathsf{tally}(\{\beta' \to \beta' \dot{\leq} (\beta \to \beta) \to \alpha\})$ which has interesting solutions, in particular $\{\beta' \rightsquigarrow \beta \to \beta\ ;\ \alpha \rightsquigarrow \beta \to \beta\}$. The side-condition $\Sigma \neq \varnothing$ ensures that the tallying instance has at least one solution (otherwise the annotation produced would be invalid). The rules for projections, pairs, and type-cases are similar and, thus, omitted.

$$[\textsc{BindKeep}]\ \frac{\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A}\rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathscr{A}} [a \mid \mathbb{a}] : s \qquad (\forall i \in I)\ \ \Gamma, \mathtt{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K}_i\rangle \Rightarrow \mathbb{k}_i}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{bind}\,\mathtt{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\ (\mathcal{A}, \varnothing, \{(\mathbf{u}_i, \mathcal{K}_i)\}_{i\in I})\rangle \Rightarrow \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i)\}_{i\in I})}\ (*)$$

(where $(*)$ is $\bigvee_{i\in I} \mathbf{u}_i \simeq \mathbb{1}$). The rule [BindKeep] takes as input an intermediate annotation $\mathsf{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$, with $\mathcal{S} = \varnothing$, since all branches must have been fully explored by the main reconstruction algorithm. The rule recursively transforms the intermediate annotation $\mathcal{A}$ for the definition a into an annotation $\mathbb{a}$ for the algorithmic type system, and uses it to type a. It can then update the environment and proceed recursively on the body κ, for each branch in $\mathcal{S}'$.

4.4 Properties of the Reconstruction Algorithm

As recalled at the beginning of the section, reconstruction is sound, terminating, but incomplete.

Theorem 4.2 (Soundness). *If* $\Gamma \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathbb{k}$, *then* $\exists t.\ \Gamma \vdash_{\mathscr{A}} [\kappa \mid \mathbb{k}] : t$.

Theorem 4.3 (Termination). *The deduction rules* $\vdash^*_{\mathcal{R}}$ *and* $\vdash_{\mathcal{R}}$ *define a terminating algorithm: it can either fail (if no rule applies at some point) or return a result* $\mathbb{R}$.

The incompleteness of the reconstruction algorithm is inherent to our system and derives from the lack of principal typing. A simple example is the curried function `map` defined in the third row of Table 1 in the next section. Our reconstruction deduces for it the type $(\alpha \to \beta) \to [\alpha*] \to [\beta*]$ (actually, a slightly more precise type), where $[\alpha*]$ is the type of the lists of elements of type α. This states that an application of `map` yields a function that maps lists of α's into lists of β's. But for every natural number n, the declarative system can also deduce that the result maps lists of α's of length n into lists of β's of the same length n. Our algorithm can *check* each of these types, but none of them can be deduced from the type reconstructed by the algorithm. And since we do not have dependent types or infinite intersections, then the declarative system cannot have a principal type expressing all these different derivations. In other terms, incompleteness stems from the fact that the declarative system can use all the infinitely many decompositions of unions in the union elimination rule, and the infinitely many decompositions of the domain of a function when reconstructing its type as an intersection of arrows. The algorithmic counterpart of this, is that there are infinitely many annotations that the algorithmic system can use to type these expressions and that these infinite choices cannot be summarized by a notion of principal annotation: the reconstruction chooses one particular annotation, and therefore it will miss some solutions.

There is a second source of incompleteness for reconstruction, which is not inherent to the system, but a design choice, instead: the fact that reconstruction does not perform the so-called "expansion" of intersection types. This is shown by the rule [App] in Section 4.3, where tally is applied without expanding the types in the constraint (e.g., if $\mathsf{tally}(\{t_1\rho_1 \dot{\leq} t_2\rho_2 \to \alpha\})$ fails we can expand the type of the function and try $\mathsf{tally}(\{t_1\rho_1 \wedge t_1\rho_3 \dot{\leq} t_2\rho_2 \to \alpha\})$, and so on and so forth by alternating expansions on the function and on the argument types: see [Castagna et al.

[10] The solution is not interesting since it is the one that allows any simply typed system to type all pure lambda terms. We do not need this recursive type to type, say, the application of the polymorphic identity function to itself.

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

<table>
<tr><th></th><th>Code</th><th>Inferred type</th><th>Time</th></tr>
<tr><td>1</td><td><code>type Falsy = False | "" | 0</code>
<code>type Truthy = ~Falsy</code></td><td></td><td></td></tr>
<tr><td></td><td><code>let toBoolean x =</code>
<code>   if x is Truthy then true else false</code></td><td>$(\text{Falsy} \to \text{False}) \wedge (\text{Truthy} \to \text{True})$</td><td>3.42</td></tr>
<tr><td></td><td><code>let lOr (x,y) =</code>
<code>   if toBoolean x then x else y</code></td><td>$(((\alpha \wedge \text{Truthy}) \times \mathbb{1}) \to \alpha \wedge \text{Truthy}) \wedge$
$((\text{Falsy} \times \beta) \to \beta)$</td><td>13.31</td></tr>
<tr><td></td><td><code>let id x = lOr (x,x)</code></td><td>$\alpha \to \alpha$</td><td>8.46</td></tr>
<tr><td>2</td><td><code>let fixpoint = fun f -></code>
<code>   let delta = fun x -></code>
<code>   f (fun v -> (x x v))</code>
<code>   in delta delta</code></td><td>$((\beta \to \alpha) \to (\beta \to \alpha) \wedge \gamma) \to (\beta \to \alpha) \wedge \gamma$</td><td>15.37</td></tr>
<tr><td>3</td><td><code>let map_stub map f lst =</code>
<code>   if lst is Nil then nil</code>
<code>   else (f (fst lst), map f (snd lst))</code></td><td>...</td><td>33.03</td></tr>
<tr><td></td><td><code>let map = fixpoint map_stub</code></td><td>$((\alpha \to \beta) \to ([\alpha*] \to [\beta*])) \wedge (\mathbb{1} \to [\,] \to [\,])$</td><td>84.75</td></tr>
<tr><td>4</td><td><code>let filter_stub filter</code>
<code>   (f: ('a->Any) & ('b -> ~True))</code>
<code>   (l: [('a|'b)*]) =</code>
<code>   if l is Nil then nil</code>
<code>   else if f(fst(l)) is True</code>
<code>   then (fst(l),filter f (snd(l)))</code>
<code>   else filter f (snd(l))</code></td><td>...</td><td>21.19</td></tr>
<tr><td></td><td><code>let filter = fixpoint filter_stub</code></td><td>$((\alpha \to \mathbb{1}) \wedge (\beta \to \neg\text{True})) \to [(\alpha \vee \beta)*] \to [(\alpha \setminus \beta)*]$</td><td>13.83</td></tr>
<tr><td>5</td><td><code>let rec flatten x = match x with</code>
<code> | [] -> []</code>
<code> | h::t -> concat (flatten h) (flatten t)</code>
<code> | _ -> [x]</code></td><td>$(\text{Tree} \to [(\alpha \setminus [\mathbb{1}*])*]) \wedge (\beta \setminus [\mathbb{1}*] \to [\,\beta \setminus [\mathbb{1}*]\,])$
where $\text{Tree} = [\text{Tree}*] \vee (\alpha \setminus [\mathbb{1}+])$</td><td>374.41</td></tr>
</table>

2015, Section 3.2.3] for more details). The consequence of this is that if you take the definition of the function `filter` given in row 4 of Table 1, and you remove all type annotations, then the type reconstructed by the algorithm for it is less precise than the one specified by the annotations, which could have been reconstructed if the algorithm had instead expanded the type of the parameter `f`.

Despite incompleteness, the declarative rules of Figure 2 form a reliable guide to which programs are accepted, provided we bear in mind that the algorithm approximates data structures according to the tests performed on them. So, typically, the type reconstructed for a function on lists, will probably differentiate the cases for empty and not-empty lists, but not for, say, lists of size 42, unless the function contains an explicit test for it. This (and to a lesser extent, expansion) is essentially the main difference with the declarative system, which has the liberty to deduce the type for the case of lists of size 42, even if this property is not tested in the body of the function. In that case, the programmer can still use an explicit type annotation to *check* that the specific type works.

5 IMPLEMENTATION

We have implemented the reconstruction algorithm presented in Section 4, using the CDuce [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 optimizations, briefly discussed at the end of this section, for instance memoization and a mechanism to avoid typing redundant branches when inferring the domains of λ-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 used 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 tested on the web-based interactive prototype hosted at https://www.cduce.org/dynlang. The web version is compiled to JavaScript using js_of_ocaml [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 $((\beta \to \alpha) \to (\beta \to \alpha)) \to (\beta \to \alpha)$, 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 $[\alpha*]$ denotes a list of elements of type α, and $[]$ denotes an empty list. The branch $\mathbb{1} \to [] \to []$ 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 is applied to a characteristic function for the set $\alpha \vee \beta$ whose type precises that the elements in β do not satisfy the predicate, then the inferred type has these elements removed from the type of the result.

The grammar for expressions in Figure 1 does not include recursive functions, since *from a theoretical viewpoint* they are useless: Milner [1978, page 356] justifies the addition of a "$\mathsf{fix}\, x.e$" expression by the fact that his system cannot type Curry's fixpoint combinator, but, as explained in Section 1.1 and shown by Code 2 above, our system can. However, *from a practical viewpoint*, the use of `let rec` definitions instead of fixed-points combinators may dramatically improve the speed of reconstruction, which is why the previous definitions of `map` and `filter` with a fixed-point combinator must be considered just as stress tests 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. Code 5 shows the use of `let rec` and of pattern matching (cf. Appendix A.3) and is an example of the improvement brought by `let rec` definitions: reconstruction for the same definition but with a fixpoint combinator is four times slower. The code defines the deep flatten function that transforms arbitrary nested lists into the list of their elements (where `concat` is a function of type $[\alpha*] \to [\beta*] \to [(\alpha*)(\beta*)]$, the result being the type of lists starting with α elements and ending with β ones). Greenberg [2019] considers this function to be the ultimate test for any type system: as he explains, 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: CDuce 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., either a list of elements that are trees, or a value different from a list) and returns the list of elements of the tree that are not lists; the other arrow of the intersection states that when flatten is applied to an element different from a list, then it returns the list containing only that element.

Our prototype focuses on proximity with the inference system for reconstruction, rather than on performance: we used it mainly to explore and test our system, which is why it is implemented in a purely functional style with persistent data structures (so as to simulate the reconstruction inference rules). Nonetheless, a few optimizations were implemented in order to mitigate the cost of backtracking and branching. One source of inefficiency comes from the intersection nodes that are generated when a destructor is reconstructed. This generation can lead to an explosion of the number of branches to explore, even though many of these branches are redundant. In the prototype, this is mitigated by recording, for each λ-abstraction, the domains already explored for it, and by trimming branches that do not explore new combinations of domains.

Another source of inefficiency comes from the type decompositions performed after each binding. Although these type decompositions are usually small (e.g., the type of a binding is seldom split in more than two parts), it becomes an issue when typing large expressions with multiple type-cases. For instance, a preliminary and unoptimized implementation of the reconstruction algorithm took about 40 seconds to type the `bal`(ance) function used in the module `Map` of the OCaml standard library, that contains 6 different pattern matches and 4 type-cases [OCaml 2023]. Adding a simple memoization mechanism that prevents the reconstruction from retyping an atom several times for equivalent contexts, decreased the inference time down to 4 seconds.

While these simple optimizations significantly improve performance, they are still far from what would be considered acceptable for real applications. To be used in mainstream languages, the type system will have to be adapted and restricted so as to ensure better and uniform performance. To this purpose, we believe that some more language-oriented optimization techniques could be of help. An example is what the development team of Luau [Luau] did on the occasion of its recent switch to semantic subtyping [Jeffrey 2022]. The developers did this switch by implementing a two-phase approach: first, a sound syntactic system, fast but imprecise, is used to try to prove subtyping, and only if it fails, the computationally expensive semantic subtyping inference is used. We think not only that such a staged approach could be applied in our case, but also that the partial results of the first phase could be used to improve the performance of the later phases, as in the case of the `let rec`, where knowing the arity of the defined function improves the performance of the reconstruction. This could be further coupled with slicing, meaning that our type reconstruction could be applied to very delimited regions that would bound the possibility of backtracking. These techniques are language-dependent, and quite different from the algorithmic aspects developed here, though they will completely rely on it. We plan to explore them in future work.

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 $\mathcal{W}$ 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 λ-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 λ-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 λ-abstraction. Note that none of these works features 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].

Jim [2000] presents a polar type system which features intersections and parametric polymorphism. In Jim's type system, quantifiers may appear only in positive positions in types, while intersections may only appear in negative positions. This yields a system that is more expressive than rank-2 intersection types, and therefore more expressive than ML. Furthermore, the system features principal types, and a decidable type inference. Some aspects of this work are similar to ours, in particular the use of MGS, an algorithm to compute the most general solution of a (syntactic) sub-typing problem, that plays the same role as our tallying algorithm. Despite these similarities, the approaches differ in the kind of programs they handle: in [Jim 2000], intersections are only deduced by applying higher-order function parameters to arguments of distinct types within the body of a function, while in our approach, they can also be caused by a type-case.

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. [2023a] 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 at providing 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). In that sense, our work is more than a simple study on typability: as a matter of fact, monomorphic intersection and union types are sufficient to type a closed program where all function applications are known (cf., Section 1.2), but this would be bad from a language design point of view, and it is the reason why people program using ML-style programming languages rather than intersection based ones. Separate compilation and modular definitions are requirements of any reasonable programming language. The essence of this work is thus to challenge the limits of how much precision one can obtain (through intersection types)—ideally precise enough to type idioms of dynamic languages—while preserving modularity (thanks to let-polymorphism).

While we believe our work to be an important step towards a better static typing of dynamic languages, several key features are still missing. First, the presence of side effects may invalidate our approach: if the [$\vee$] rule in Figure 2 is applied to two different occurrences of an expression e' that is not pure, then the rule may type an expression that yields a run-time type error. This can be seen on the algorithmic system, where the transformation into an MSC-form binds the two occurrences of e' to the same variable, thus wrongly assuming that they both yield the same result. Strictly speaking, our algorithmic approach does not require expressions to be pure; it just needs that when two occurrences of an expression may produce two distinct values that may change the result of a dynamic test, then these two occurrences must be bound by two different binds. Having only pure expressions is a straightforward way to satisfy this property. Having each subexpression bound to a distinct variable (i.e., no sharing, that is, a less precise system, in which the union rule is never used) is a way to retain safety in the presence of side-effects. But between these two extrema, there is a whole palette of less coarse solutions that make it possible to apply our approach in the presence of side-effects, and that we plan to study in future work. This poses two main challenges: (i) how to separate problematic expressions from non-problematic ones (e.g., a `gen_id: Unit→Int` function performs side-effects, but if its result is tested only against `Int`, then it is sound to have all occurrences of `gen_id` bound by the same bind during typing) which, in terms of the type system, corresponds to characterize a class of subexpressions e' that can be safely used in rule [$\vee$]; and (ii) how to do so *before* our type inference, at a point when type information is not available, yet.

Second, while the performance of our prototype is reasonable, it can certainly be improved by using more sophisticated implementations techniques and heuristics on the lines we outlined at the end of Section 5.

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.

DATA AVAILABILITY STATEMENT

- All the auxiliary definitions, proofs and extensions that we omitted from the main text are available in the appendix of the extended version [Castagna et al. 2024].
- The reference implementation of the type-reconstruction system is archived on Zenodo [Castagna et al. 2023b]. An online version of the prototype is available at:

https://www.cduce.org/dynlang

ACKNOWLEDGMENTS

We warmly thank the POPL reviewers: their careful reading and suggestions allowed us to improve the presentation significantly. A special thank the reviewers of the POPL artifact evaluation for their detailed and insightful reviews.

This work was partially supported by the *Chaire Langages Dynamiques pour les Données* of the *Fondation Université Paris-Saclay*, by the *SECUREVAL* ANR project n. ANR-22-PECY-0005 and by a CIFRE PhD. grant with Remote Technology.

REFERENCES

Pedro Ângelo and Mário Florido. 2022. Type Inference For Rank-2 Intersection Types Using Set Unification. In *Theoretical Aspects of Computing – ICTAC 2022: 19th International Colloquium, Tbilisi, Georgia, September 27–29, 2022, Proceedings* (Tbilisi, Georgia). Springer-Verlag, Berlin, Heidelberg, 462–480. https://doi.org/10.1007/978-3-031-17715-6_29

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types. *Inf. Comput.* 119, 2 (June 1995), 202–230. https://doi.org/10.1006/inco.1995.1086

Viviana Bono, Betti Venneri, and Lorenzo Bettini. 2008. A typed lambda calculus with intersection types. *Theor. Comput. Sci.* 398, 1-3 (2008), 95–113. https://doi.org/10.1016/j.tcs.2008.01.046

Giuseppe Castagna. 2023a. Programming with union, intersection, and negation types. In *The French School of Programming*, Bertrand Meyer (Ed.). Springer. ISBN 978-3-031-34517-3. Preprint at arXiv:2111.03354.

Giuseppe Castagna. 2023b. Typing Records, Maps, and Structs. *Proc. ACM Program. Lang.* 7, ICFP, Article 196 (Sept. 2023), 45 pages. https://doi.org/10.1145/3607838

Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023a. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (2023). https://doi.org/10.22152/programming-journal.org/2024/8/4 Preprint in ArXiv: arXiv:2306.06391..

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2022a. Revisiting Occurrence Typing. *Science of Computer Programming* 217 (mar 2022), 102781. https://doi.org/10.1016/j.scico.2022.102781 arXiv:1907.05590

Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2023b. *Prototype: Polymorphic Type Inference for Dynamic Languages.* https://doi.org/10.5281/zenodo.10155221

Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024. Polymorphic Type Inference for Dynamic Languages (Extended version). https://doi.org/10.1145/3632882

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

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

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-Theoretic Types for Polymorphic Variants. In *ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming.* 378–391. https://doi.org/10.1145/2951913.2951928

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming.* 94–106. https://doi.org/10.1145/2034773.2034788

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

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

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

Mariangiola Dezani. 2020. Personal communication.

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

Ecma. 2021. ECMAScript® 2021 Language Specification. https://262.ecma-international.org/12.0/

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

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 137–146. https://doi.org/10.1109/LICS.2002.1029823

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

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

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*. 6:1–6:20. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6

Fritz Henglein. 1993. Type Inference with Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (apr 1993), 253–289. https://doi.org/10.1145/169701.169692

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

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

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

Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. 1993. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (apr 1993), 290–311. https://doi.org/10.1145/169701.169687

Daniel Leivant. 1983. Polymorphic Type Inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Austin, Texas). Association for Computing Machinery, New York, NY, USA, 88–98. https://doi.org/10.1145/567067.567077

Luigi Liquori and Simona Ronchi Della Rocca. 2007. Intersection-types à la Church. *Inf. Comput.* 205, 9 (2007), 1371–1386. https://doi.org/10.1016/j.ic.2007.03.005

Luau. *Luau*. https://luau-lang.org/

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

Per Martin-Löf. 1994. *Analytic and Synthetic Judgements in Type Theory*. Springer Netherlands, Dordrecht, 87–99. https://doi.org/10.1007/978-94-011-0834-8_5

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

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

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

Ocsigen. *JS of OCaml*. Ocsigen https://ocsigen.org/js_of_ocaml/

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. *SIGPLAN Not.* 51, 9 (sep 2016), 364–377. https://doi.org/10.1145/3022670.2951945

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

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

Tommaso Petrucciani, Giuseppe Castagna, Davide Ancona, and Elena Zucca. 2018. Semantic Subtyping for Non-Strict Languages. In *TYPES18: 24th International Conference on Types for Proofs and Programs (LIPIcs, Vol. 130)*, Peter Dybjer, José Espírito Santo, and Luís Pinto (Eds.). 4:1–4:24. https://doi.org/10.4230/LIPIcs.TYPES.2018.4

Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in $F_{\bowtie}$. *Proc. ACM Program. Lang.* 7, POPL, Article 18 (jan 2023), 29 pages. https://doi.org/10.1145/3571211

John Alan Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (jan 1965), 23–41. https://doi.org/10.1145/321250.321253

Simona Ronchi Della Rocca. 2002. Intersection Typed lambda-calculus. *Electr. Notes Theor. Comput. Sci.* 70, 1 (2002), 163–181. https://doi.org/10.1016/S1571-0661(04)80496-1

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style.. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, 288–298. https://doi.org/10.1145/141471.141563

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

Christopher Strachey. 1967. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen.

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

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

Types 2019. What exactly should we call syntax-directed inference rules? Discussion on the Types mailing list. http://lists.seas.upenn.edu/pipermail/types-list/2019/002138.html.

Joseph Brian Wells, Allyn Dimock, Robert J Muller, and Franklyn Albin Turbak. 2002. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.* 12, 3 (2002), 183–227. https://doi.org/10.1017/S0956796801004245

Joe B. Wells and Christian Haack. 2002. Branching Types. In *ESOP '02 (LNCS, Vol. 2305)*. Springer, 115–132. https://doi.org/10.1007/3-540-45927-8_9

Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 9 (sep 2021), 61 pages. https://doi.org/10.1145/3460228

A EXTENSIONS

In this appendix, we present some extensions for the source language, in particular let-bindings (not to be coufounded 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:

$$\textbf{Expressions} \quad e \quad ::= \quad \cdots \mid \texttt{let}\, x = e \,\texttt{in}\, e$$

with the following notion of reduction:

$$\texttt{let}\, x = v \,\texttt{in}\, e \quad \rightsquigarrow \quad e\{v/x\}$$

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

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\, x = e_1 \,\texttt{in}\, e_2\ : t_2}$$

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

$$\lambda x.\ \texttt{let}\, y = x \,\texttt{in}\, (f\ x{\in}\mathsf{Int})\ ?\ f\ y : 42$$

with $f : \mathbb{1} \to \mathbb{1}$.

Though for any argument x this function yields an integer, it is not possible to derive for it the type $\mathbb{1} \to \mathsf{Int}$ using this extension of the declarative type system. Indeed, $f\ x$ and $f\ y$ are not syntactically equivalent and thus the $[\vee]$ rule can only decompose their types independently, loosing 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:

$$\textbf{Expressions} \quad e \quad ::= \quad \cdots \mid \texttt{let}\, e \,\texttt{in}\, e$$

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

$$(\!|\texttt{let}\, x = e_1 \,\texttt{in}\, e_2|\!) = \texttt{let}\, (\!|e_1|\!) \,\texttt{in}\, (\!|e_2|\!)\{(\!|e_1|\!)/x\}$$

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

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\, e_1 \,\texttt{in}\, e_2\ : t_2}$$

A.1.2 Algorithmic Type System. Let-bindings are added to MSC forms as a new atom construction:

$$\textbf{Atomic expr} \quad a \quad ::= \quad \cdots \mid \mathtt{let}\,\mathsf{x}\,\mathtt{in}\,\mathsf{x}$$

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 $\mathtt{let}\,x = e_1\,\mathtt{in}\,e_2$ we must replace each subexpression by a binding variable, which would yield something of the form $\mathtt{let}\,x = \mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2$. 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 $\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2$ and which explains the definition of the atom for let expressions.

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

$$\begin{array}{l}\mathtt{bind}\,\mathsf{x}_1 = (\lambda y.\mathtt{bind}\,\mathsf{y} = y\,\mathtt{in}\,\mathsf{y}\,)\,\mathtt{in}\\ \mathtt{bind}\,\mathsf{x}_2 = (\mathsf{x}_1, \mathsf{x}_1)\,\mathtt{in}\\ \mathtt{bind}\,\mathsf{x}_3 = (\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2\,)\,\mathtt{in}\\ \mathsf{x}_3\end{array}$$

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:

$$[\textsc{Let-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2 \mid \varnothing] : \Gamma(\mathsf{x}_2)}\ \mathsf{x}_1 \in \mathsf{dom}(\Gamma)$$

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 $(e \mathbin{⨟} \tau)$ 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 τ (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:

$$\textbf{Expressions} \quad e \quad ::= \quad \cdots \mid (e \mathbin{⨟} \tau)$$

with the following notion of reduction:

$$(v \mathbin{⨟} \tau) \quad \rightsquigarrow \quad v \qquad\qquad \text{if } v \in \tau$$

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

$$[\textsc{Constr}]\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e : t}{\Gamma \vdash (e \mathbin{⨟} \tau) : t}$$

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

$$\textbf{Atomic expr} \quad a \quad ::= \quad \cdots \mid \mathsf{x} \mathbin{⨟} \tau$$

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

$$\textbf{Atoms annotations} \quad \mathbb{a} \quad ::= \quad \cdots \mid {⨟}(\Sigma)$$

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

$$[\textsc{Constr-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x} \mathbin{⨟} \tau \mid {⨟}(\Sigma)] : \Gamma(\mathsf{x})}\ \Gamma(\mathsf{x})\Sigma \leq \tau$$

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:

$$\textbf{Expressions} \quad e \quad ::= \quad \cdots \mid (\texttt{tcase}\, e\, \texttt{of}\, \tau \to e \mid \ldots \mid \tau \to e)$$

with the following notion of reduction:

$$\texttt{tcase}\, v\, \texttt{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n \quad \rightsquigarrow \quad e_k \quad \begin{array}{l}\text{if } v : \tau_k \setminus (\bigvee_{i\in 1..k-1} \tau_i) \\ \text{for any } k \in 1..n\end{array}$$

In terms 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 langauge presented in section 2, with the let-binding and type constraints extensions (A.1 and A.2). For that, we use the following transformation:

$$\begin{aligned}
(\!|c|\!) &= c \\
(\!|x|\!) &= x \\
(\!|\lambda x.e|\!) &= \lambda x.(\!|e|\!) \\
(\!|\pi_i e|\!) &= \pi_i (\!|e|\!) \\
(\!|e_1 e_2|\!) &= (\!|e_1|\!)(\!|e_2|\!) \\
(\!|(e_1, e_2)|\!) &= ((\!|e_1|\!), (\!|e_2|\!)) \\
(\!|(e\in\tau)\,?\,e_1 : e_2|\!) &= ((\!|e|\!)\in\tau)\,?\,(\!|e_1|\!) : (\!|e_2|\!) \\
(\!|\texttt{let}\, x = e_1\, \texttt{in}\, e_2|\!) &= \texttt{let}\, x = (\!|e_1|\!)\, \texttt{in}\, (\!|e_2|\!) \\
(\!|e \mathbin{⦂} \tau|\!) &= (\!|e|\!) \mathbin{⦂} \tau \\
(\!|\texttt{tcase}\, e\, \texttt{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n|\!) &= \begin{array}{l}\texttt{let}\, x = ((\!|e|\!) \mathbin{⦂} \bigvee_{i\in 1..n} \tau_i)\, \texttt{in} \\ \mathsf{c}_x(\tau_1 \to (\!|e_1|\!)\, ; \ldots ; \tau_n \to (\!|e_n|\!))\end{array} \quad \text{with } x \text{ fresh} \\
\mathsf{c}_x(\tau \to e) &= e \\
\mathsf{c}_x(\tau \to e\, ; C) &= (x\in\tau)\,?\,e : \mathsf{c}_x(C)
\end{aligned}$$

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

$$\begin{array}{llll}
\textbf{Patterns} & p & ::= & \tau \mid x \mid p\&p \mid p|p \mid (p,p) \mid x := c \\
\textbf{Expressions} & e & ::= & \cdots \mid (\texttt{match}\, e\, \texttt{with}\, p \to e \mid \ldots \mid p \to e)
\end{array}$$

The associated reduction rule can be found in Appendix B.

In terms 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:

$$
\begin{aligned}
(\!|c|\!) &= c\\
(\!|x|\!) &= x\\
(\!|\lambda x.e|\!) &= \lambda x.(\!|e|\!)\\
(\!|\pi_i e|\!) &= \pi_i(\!|e|\!)\\
(\!|e_1 e_2|\!) &= (\!|e_1|\!)(\!|e_2|\!)\\
(\!|(e_1, e_2)|\!) &= ((\!|e_1|\!), (\!|e_2|\!))\\
(\!|(e\in\tau)\ ?\ e_1 : e_2|\!) &= ((\!|e|\!)\in\tau)\ ?\ (\!|e_1|\!) : (\!|e_2|\!)\\
(\!|\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2|\!) &= \mathtt{let}\ x = (\!|e_1|\!)\ \mathtt{in}\ (\!|e_2|\!)\\
(\!|e \mathbin{⨾} \tau|\!) &= (\!|e|\!) \mathbin{⨾} \tau\\
(\!|\mathtt{tcase}\ e\ \mathtt{of}\ \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n|\!) &= \mathtt{tcase}\ (\!|e|\!)\ \mathtt{of}\ \tau_1 \to (\!|e_1|\!) \mid \ldots \mid \tau_n \to (\!|e_n|\!)\\
(\!|\mathtt{match}\ e\ \mathtt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n|\!) &= \mathtt{let}\ x = (\!|e|\!)\ \mathtt{in}\ \mathtt{tcase}\ x\ \mathtt{of}\ \begin{array}{l} \wr p_1 \int \to e'_1\\ \mid \ldots \\ \mid \wr p_n \int \to e'_n \end{array}
\end{aligned}
$$

with x fresh,

$$
\begin{aligned}
\wr\tau\int &= \tau\\
\wr x\int &= \mathbb{1}\\
\wr p_1\&p_2\int &= \wr p_1\int \wedge \wr p_2\int\\
\wr p_1|p_2\int &= \wr p_1\int \vee \wr p_2\int\\
\wr(p_1,p_2)\int &= \wr p_1\int \times \wr p_2\int\\
\wr x := c\int &= \mathbb{1}
\end{aligned}
$$

and where for every $i \in 1..m$:
$e'_i = \mathtt{let}\ x_1 = \mathsf{d}_{x_1}(p_i, x)\ \mathtt{in}\ \ldots \mathtt{let}\ x_m = \mathsf{d}_{x_m}(p_i, x)\ \mathtt{in}\ (\!|e_i|\!)$ for $\{x_1, ..., x_m\} = \mathsf{vars}(p_i)$ with

$$
\begin{aligned}
\mathsf{d}_x(x, e) &= e\\
\mathsf{d}_x(x := c, e) &= c\\
\mathsf{d}_x((p_1, p_2), e) &= \mathsf{d}_x(p_i, \pi_i e) && \text{if } x \in \mathsf{vars}(p_i)\\
\mathsf{d}_x(p_1\&p_2, e) &= \mathsf{d}_x(p_i, e) && \text{if } x \in \mathsf{vars}(p_i)\\
\mathsf{d}_x(p_1|p_2, e) &= (e\in \wr p_1 \int)\ ?\ \mathsf{d}_x(p_1, e) : \mathsf{d}_x(p_2, e)\\
\mathsf{d}_x(p, e) &= \text{undefined} && \text{otherwise}
\end{aligned}
$$

B FULL SEMANTICS WITH EXTENSIONS

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

Test Types	τ	::=	$b \mid \mathbb{0} \to \mathbb{1} \mid \tau \times \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0}$
Patterns	p	::=	$\tau \mid x \mid p\&p \mid p\vert p \mid (p, p) \mid x := c$
Expressions	e	::=	$c \mid x \mid \lambda x.e \mid ee \mid (e, e) \mid \pi_i e \mid (e\in\tau)\ ?\ e : e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid (e \mathbin{⨾} \tau)$
			$\mid (\mathtt{tcase}\ e\ \mathtt{of}\ \tau \to e \mid \ldots \mid \tau \to e) \mid (\mathtt{match}\ e\ \mathtt{with}\ p \to e \mid \ldots \mid p \to e)$
Values	v	::=	$c \mid \lambda x.e \mid (v, v)$

The associated reduction rules are:

$$
\begin{array}{rcll}
(\lambda x.e)v & \rightsquigarrow & e\{v/x\} & \\
\pi_1(v_1,v_2) & \rightsquigarrow & v_1 & \\
\pi_2(v_1,v_2) & \rightsquigarrow & v_2 & \\
(v{\in}\tau)\ ?\ e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } v \in \tau \\
(v{\in}\tau)\ ?\ e_1 : e_2 & \rightsquigarrow & e_2 & \text{if } v \in \neg\tau \\
\mathtt{let}\, x = v\, \mathtt{in}\, e & \rightsquigarrow & e\{v/x\} & \\
(v \mathbin{⨾} \tau) & \rightsquigarrow & v & \text{if } v \in \tau \\
\mathtt{tcase}\, v\, \mathtt{of}\, \tau_1 \to e_1 \mid \ldots \mid \tau_n \to e_n & \rightsquigarrow & e_k & \begin{array}{l}\text{if } v : \tau_k \setminus (\bigvee_{i\in 1..k-1} \tau_i) \\ \text{for any } k \in 1..n\end{array} \\
\mathtt{match}\, v\, \mathtt{with}\, p_1 \to e_1 \mid \ldots \mid p_n \to e_n & \rightsquigarrow & e_k(v/p_k) & \begin{array}{l}\text{if } v : \llparenthesis p_k \rrparenthesis \setminus (\bigvee_{i\in 1..k-1} \llparenthesis p_i \rrparenthesis) \\ \text{for any } k \in 1..n\end{array}
\end{array}
$$

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

$$
\begin{array}{lrl}
\textbf{Evaluation Context} & E ::= & [\,] \mid vE \mid Ee \mid (v,E) \mid (E,e) \mid \pi_i E \mid (E{\in}\tau)\ ?\ e : e \\
& \mid & \mathtt{let}\, x = E\, \mathtt{in}\, e \mid (E \mathbin{⨾} \tau) \\
& \mid & (\mathtt{tcase}\, E\, \mathtt{of}\, \tau \to e \mid \ldots \mid \tau \to e) \\
& \mid & (\mathtt{match}\, E\, \mathtt{with}\, p \to e \mid \ldots \mid p \to e)
\end{array}
$$

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

$$
\begin{array}{rcll}
c\{e'/x\} & = & c & \\
x\{e'/x\} & = & e' & \\
y\{e'/x\} & = & y & x \neq y \\
(\lambda x.e)\{e'/x\} & = & \lambda x.e & \\
(\lambda y.e)\{e'/x\} & = & \lambda y.(e\{e'/x\}) & x \neq y, y \notin \mathrm{fv}(e') \\
(\lambda y.e)\{e'/x\} & = & \lambda z.(e\{z/y\}\{e'/x\}) & x \neq y, y \in \mathrm{fv}(e'), z \text{ fresh} \\
(e_1 e_2)\{e'/x\} & = & (e_1\{e'/x\})(e_2\{e'/x\}) & \\
(e_1, e_2)\{e'/x\} & = & (e_1\{e'/x\}, e_2\{e'/x\}) & \\
(\pi_i e)\{e'/x\} & = & \pi_i(e\{e'/x\}) & \\
((e_1{\in}\tau)\ ?\ e_2 : e_3)\{e'/x\} & = & (e_1\{e'/x\}{\in}\tau)\ ?\ e_2\{e'/x\} : e_3\{e'/x\} & \\
(\mathtt{let}\, x = e_1\, \mathtt{in}\, e_2)\{e'/x\} & = & \mathtt{let}\, x = e_1\{e'/x\}\, \mathtt{in}\, e_2 & \\
(\mathtt{let}\, y = e_1\, \mathtt{in}\, e_2)\{e'/x\} & = & \mathtt{let}\, y = e_1\{e'/x\}\, \mathtt{in}\, e_2\{e'/x\} & x \neq y, y \notin \mathrm{fv}(e') \\
(\mathtt{let}\, y = e_1\, \mathtt{in}\, e_2)\{e'/x\} & = & \mathtt{let}\, y = e_1\{e'/x\}\, \mathtt{in}\, e_2\{z/y\}\{e'/x\} & x \neq y, y \in \mathrm{fv}(e'), z \text{ fresh}
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathrm{typeof}(\lambda x.e) & = & \mathbb{0} \to \mathbb{1} \\
\mathrm{typeof}(c) & = & b_c \\
\mathrm{typeof}((v_1, v_2)) & = & \mathrm{typeof}(v_1) \times \mathrm{typeof}(v_2)
\end{array}
$$

Finally, the operators used in the reduction rule for pattern matching are defined as follows:

$$\begin{aligned}
\wr\tau\int &= \tau \\
\wr x\int &= \mathbb{1} \\
\wr p_1\&p_2\int &= \wr p_1\int \wedge \wr p_2\int \\
\wr p_1|p_2\int &= \wr p_1\int \vee \wr p_2\int \\
\wr (p_1,p_2)\int &= \wr p_1\int \times \wr p_2\int \\
\wr x:=c\int &= \mathbb{1}
\end{aligned}$$

and

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

C SUBTYPING RELATION

Subtyping is defined by giving a set-theoretic interpretation of the types of Definition 2.1 into a suitable domain $\mathcal{D}$. 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 succintly 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 *$\mathcal{D}$ is the set of finite terms d produced inductively by the following grammar*

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

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

The elements of $\mathcal{D}$ correspond, intuitively, to (denotations of) the results of the evaluation of expressions, labeled by finite sets of type variables. In particular, in a higher-order language, the results of computations can be functions which, in this model, are represented by sets of finite relations of the form $\{(d_1,\partial_1),\dots,(d_n,\partial_n)\}^L$, where Ω (which is not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This is implemented by using in the second projection the meta-variable ∂ which ranges over $\mathcal{D}_\Omega = \mathcal{D}\cup\{\Omega\}$ (we reserve d to range over $\mathcal{D}$, thus excluding Ω). This constant Ω is used to ensure that $\mathbb{1}\to\mathbb{1}$ is not a supertype of all function types: if we used d instead of ∂, then every well-typed function could be subsumed to $\mathbb{1}\to\mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently 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 α by the set of all elements that are labeled by α, that is $[\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\}$ (where we define $\mathsf{tags}(c^L) = \mathsf{tags}((d, d')^L) = \mathsf{tags}(\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L) = L$).

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

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

We cannot take the equations above directly as an inductive definition of $[\![\,]\!]$ because types are not defined inductively but coinductively. Notice however that the contractivity condition of Definition 2.1 ensures that the binary relation $\rhd \subseteq \mathbf{Types} \times \mathbf{Types}$ defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian. This gives an induction principle[11] 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* $t \in \mathbf{Types}$*, by induction on the pair* (d, t) *ordered lexicographically. The predicate is defined as follows:*

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

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

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* $\simeq$ *as* $t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ *and* $t_1 \simeq t_2 \overset{\text{def}}{\iff} (t_1 \leq t_2) \text{ and } (t_2 \leq t_1)$.

D DECLARATIVE TYPE SYSTEM WITH EXTENSIONS

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

$$\textbf{Expressions} \quad e \quad ::= \quad c \mid x \mid \lambda x.e \mid ee \mid (e, e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e : e \mid \texttt{let}\, e\, \texttt{in}\, e \mid (e \mathbin{⨾} \tau)$$

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 $\texttt{let}\, e\, \texttt{in}\, e$ for let-bindings instead of the initial construction $\texttt{let}\, x = e\, \texttt{in}\, e$ in order to avoid aliasing. You should refer to Section A.1 for more details on this transformation.

[11] 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:

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : b_c} \qquad [\text{Ax}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$[\to\text{I}]\ \frac{\Gamma, x : \mathbf{u} \vdash e : t}{\Gamma \vdash \lambda x.e : \mathbf{u} \to t} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

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

$$[\mathbb{0}]\ \frac{\Gamma \vdash e : \mathbb{0}}{\Gamma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : \mathbb{0}} \qquad [\in_1]\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg\tau \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge \mathbf{u} \vdash e : t \qquad \Gamma, x : s \wedge \neg\mathbf{u} \vdash e : t}{\Gamma \vdash e\{e'/x\} : t} \qquad [\wedge]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

$$[\textsc{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma} \qquad [\leq]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t'}\ t \leq t'$$

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

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\, e_1\, \texttt{in}\, e_2\ : t_2} \qquad [\textsc{Constr}]\ \frac{\Gamma \vdash e \mathbin{⨾} \tau \qquad \Gamma \vdash e : t}{\Gamma \vdash (e \mathbin{⨾} \tau) : t}$$

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:

$$\begin{array}{lll} \textbf{Atomic expressions} & a & ::= \ c \mid x \mid \lambda x.\kappa \mid (\mathsf{x}, \mathsf{x}) \mid \mathsf{x}\mathsf{x} \mid \pi_i \mathsf{x} \mid (\mathsf{x}{\in}\tau)\ ?\ \mathsf{x} : \mathsf{x} \mid \texttt{let}\, \mathsf{x}\, \texttt{in}\, \mathsf{x} \mid \mathsf{x} \mathbin{⨾} \tau \\ \textbf{Canonical Forms} & \kappa & ::= \ \mathsf{x} \mid \texttt{bind}\, \mathsf{x} = a\, \texttt{in}\, \kappa \end{array}$$

Any canonical form can be transformed into an expression of the source language using the unwiding operator $\lceil . \rceil$ defined as follows:

$$\begin{array}{rcll} \lceil c \rceil & = & c & \\ \lceil x \rceil & = & x & \\ \lceil \lambda x.\kappa \rceil & = & \lambda x.\lceil \kappa \rceil & \\ \lceil \mathsf{x}_1 \mathsf{x}_2 \rceil & = & \mathsf{x}_1 \mathsf{x}_2 & \\ \lceil (\mathsf{x}_1, \mathsf{x}_2) \rceil & = & (\mathsf{x}_1, \mathsf{x}_2) & \\ \lceil \pi_i \mathsf{x} \rceil & = & \pi_i \mathsf{x} & i = 1, 2 \\ \lceil (\mathsf{x}{\in}\tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2 \rceil & = & (\mathsf{x}{\in}\tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2 & \\ \lceil \texttt{let}\, \mathsf{x}_1\, \texttt{in}\, \mathsf{x}_2 \rceil & = & \texttt{let}\, x = \mathsf{x}_1\, \texttt{in}\, \mathsf{x}_2\{x/\mathsf{x}_1\} \text{ with } x \text{ fresh} & \\ \lceil \mathsf{x} \mathbin{⨾} \tau \rceil & = & \mathsf{x} \mathbin{⨾} \tau & \\ \lceil \texttt{bind}\, \mathsf{x} = a\, \texttt{in}\, \kappa \rceil & = & \lceil \kappa \rceil \{\lceil a \rceil / \mathsf{x}\} & \\ \lceil \mathsf{x} \rceil & = & \mathsf{x} & \end{array}$$

E.2 From Source Language Expressions to Canonical Forms

The inverse direction, that is, producing from a source language expression a canonical form that unwinds to it, is also straightforward.

Let B denote a binding context, that is, an ordered list of mappings from binding variables to atoms. Each mapping is written as a pair (x, e). We note these lists extensionally by separating elements by a semicolon, that is, $(\mathsf{x}_1, a_1);\ldots;(\mathsf{x}_n, a_n)$ and use ε to denote the empty list.

We define an operation $\mathsf{term}(B, \kappa)$ which takes a binding context B and a canonical form κ and constructs the canonical form containing the bindings listed in B and ending with κ, that is:

$$\mathsf{term}(\varepsilon, \kappa) \stackrel{\text{def}}{=} \kappa$$

$$\mathsf{term}(((\mathsf{x}, a); B), \kappa) \stackrel{\text{def}}{=} \mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\mathsf{term}(B, \kappa)$$

We can now define the function $[\![e]\!]$ that transforms an expression e into a pair (B, x) formed by a binding context B and a binding variable x that will be bound to the atom representing e. The definition is as follows, where $\mathsf{x}_\circ$ is a fresh binding variable.

$$\begin{aligned}
[\![c]\!] &= ((\mathsf{x}_\circ, c), \mathsf{x}_\circ) \\
[\![x]\!] &= ((\mathsf{x}_\circ, x), \mathsf{x}_\circ) \\
[\![\lambda x.e]\!] &= ((\mathsf{x}_\circ, \lambda x.\mathsf{term}[\![e]\!]), \mathsf{x}_\circ) \\
[\![\pi_i e]\!] &= ((B; (\mathsf{x}_\circ, \pi_i \mathsf{x})), \mathsf{x}_\circ) && \text{where } (B, \mathsf{x}) = [\![e]\!] \\
[\![e_1 e_2]\!] &= ((B_1; B_2; (\mathsf{x}_\circ, \mathsf{x}_1\mathsf{x}_2)), \mathsf{x}_\circ) && \text{where } (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!] \\
[\![(e_1, e_2)]\!] &= ((B_1; B_2; (\mathsf{x}_\circ, (\mathsf{x}_1, \mathsf{x}_2))), \mathsf{x}_\circ) && \text{where } (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!] \\
[\![(e{\in}\tau)\ ?\ e_1 : e_2]\!] &= ((B; B_1; B_2; (\mathsf{x}_\circ, (\mathsf{x}{\in}\tau)\ ?\ \mathsf{x}_1 : \mathsf{x}_2)), \mathsf{x}_\circ) \\
&\qquad \text{where } (B, \mathsf{x}) = [\![e]\!],\ (B_1, \mathsf{x}_1) = [\![e_1]\!],\ (B_2, \mathsf{x}_2) = [\![e_2]\!] \\
[\![\mathsf{x}]\!] &= (\varepsilon, \mathsf{x}) \\
[\![\mathtt{let}\, x = e_1\,\mathtt{in}\, e_2]\!] &= ((B_1; B_2; (\mathsf{x}_\circ, \mathtt{let}\,\mathsf{x}_1\,\mathtt{in}\,\mathsf{x}_2)), \mathsf{x}_\circ) \\
&\qquad \text{where } (B_1, \mathsf{x}_1) = [\![e_1]\!], (B_2, \mathsf{x}_2) = [\![e_2\{\mathsf{x}_1/x\}]\!] \\
[\![e \mathbin{⨾} \tau]\!] &= ((B; (\mathsf{x}_\circ, \mathsf{x} \mathbin{⨾} \tau)), \mathsf{x}_\circ) && \text{where } (B, \mathsf{x}) = [\![e]\!]
\end{aligned}$$

It is easy to prove that, for any term of the source language e, $\lceil \mathsf{term}([\![e]\!]) \rceil = e$.

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.

$$\begin{array}{l} \mathtt{bind}\,\mathsf{x}_1 = a_1\,\mathtt{in} \\ \mathtt{bind}\,\mathsf{x}_2 = a_2\,\mathtt{in}\,\kappa \end{array} \dashrightarrow \mathtt{bind}\,\mathsf{x}_1 = a_1\,\mathtt{in}\,\kappa\{\mathsf{x}_1/\mathsf{x}_2\} \qquad a_1 \equiv_\kappa a_2 \tag{3}$$

$$\mathtt{bind}\,\mathsf{x} = a\,\mathtt{in}\,\kappa \dashrightarrow \kappa \qquad \mathsf{x} \notin \mathsf{fv}(\kappa) \tag{4}$$

$$\begin{array}{l} \mathtt{bind}\,\mathsf{x} = \lambda y.(\\ \quad \mathtt{bind}\,\mathsf{z} = a\,\mathtt{in}\,\kappa_\circ\,) \\ \mathtt{in}\,\kappa \end{array} \dashrightarrow \begin{array}{l} \mathtt{bind}\,\mathsf{z} = a\,\mathtt{in} \\ \mathtt{bind}\,\mathsf{x} = \lambda y.\kappa_\circ\,\mathtt{in}\,\kappa \end{array} \qquad y \notin \mathsf{fv}(a), \mathsf{z} \notin \mathsf{fv}(\kappa) \tag{5}$$

$$\kappa_1 \dashrightarrow \kappa_2 \qquad \exists \kappa_1'.\ \kappa_1 \equiv_\kappa \kappa_1' \dashrightarrow \kappa_2 \tag{6}$$

Rule (3) implements the maximal sharing: if two variables bind atoms with the same unwinding (modulo α-conversion), then the variables are unified. Rule (4) removes useless bindings. Rule (5) extrudes bindings from abstractions of variables that do not occur in the argument of the binding. Rule (6) 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 $\mathsf{MSC}(e)$ for any expression e of the source language.

F TYPE OPERATORS

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

$$\begin{array}{rcl} \mathsf{dom}(t) & = & \max\{u \mid t \leq u \to \mathbb{1}\} \\ t \circ s & = & \min\{u \mid t \leq s \to u\} \\ \boldsymbol{\pi}_1(t) & = & \min\{u \mid t \leq u \times \mathbb{1}\} \\ \boldsymbol{\pi}_2(t) & = & \min\{u \mid t \leq \mathbb{1} \times u\} \end{array}$$

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).

If any of the types at issue is empty, then the computation is straightforward: $\mathsf{dom}(\mathbb{0}) = \mathbb{1}$ and $\mathbb{0} \circ s = t \circ \mathbb{0} = \boldsymbol{\pi}_1(\mathbb{0}) = \boldsymbol{\pi}_2(\mathbb{0}) = \mathbb{0}$. Otherwise the operators are computed as follows.

For $t \overset{\text{DNF}}{\simeq} \bigvee_{i\in I} \left(\bigwedge_{p'\in P'_i} \alpha_{p'} \wedge \bigwedge_{n'\in N'_i} \neg\alpha'_{n'} \wedge \bigwedge_{p\in P_i} (s_p \to t_p) \wedge \bigwedge_{n\in N_i} \neg(s'_n \to t'_n)\right)$, where each summand of the outer union is not empty, the first two operators are computed by:

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

$$t \circ s = \bigvee_{i\in I} \left(\bigvee_{\{Q \subsetneq P_i \mid s \not\leq \bigvee_{q\in Q} s_q\}} \left(\bigwedge_{p\in P_i \setminus Q} t_p \right)\right) \qquad (\text{for } s \leq \mathsf{dom}(t))$$

For $t \overset{\text{DNF}}{\simeq} \bigvee_{i\in I} \left(\bigwedge_{p'\in P'_i} \alpha_{p'} \wedge \bigwedge_{n'\in N'_i} \neg\alpha'_{n'} \wedge \bigwedge_{p\in P_i} (s_p, t_p) \wedge \bigwedge_{n\in N_i} \neg(s'_n, t'_n)\right)$, where each summand of the outer union is not empty, the last two operators are computed by

$$\boldsymbol{\pi}_1(t) = \bigvee_{i\in I} \bigvee_{N'\subseteq N_i} \left(\bigwedge_{p\in P_i} s_p \wedge \bigwedge_{n\in N'} \neg s'_n \right)$$

$$\boldsymbol{\pi}_2(t) = \bigvee_{i\in I} \bigvee_{N'\subseteq N_i} \left(\bigwedge_{p\in P_i} t_p \wedge \bigwedge_{n\in N'} \neg t'_n \right)$$

G ALGORITHMIC TYPE SYSTEM

Atom annots $\mathbb{a} ::= \varnothing \mid \lambda(\mathbf{u}, \mathbb{k}) \mid (\rho, \rho) \mid @(\Sigma, \Sigma) \mid \pi(\Sigma) \mid \mathbb{0}(\Sigma) \mid \in_1(\Sigma) \mid \in_2(\Sigma) \mid \bigwedge(\{\mathbb{a}, \dots, \mathbb{a}\})$

Form annots $\mathbb{k} ::= \rho \mid \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}, \mathbb{k}), \dots, (\mathbf{u}, \mathbb{k})\}) \mid \mathsf{skip}\ \mathbb{k} \mid \bigwedge(\{\mathbb{k}, \dots, \mathbb{k}\})$

The algorithmic type system is defined by the following deduction rules:

$$[\textsc{Const-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [c \mid \varnothing] : b_c} \qquad [\textsc{Ax-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [x \mid \varnothing] : \Gamma(x)}$$

$$[{\to}\textsc{I-Alg}]\ \frac{\Gamma, x : \mathbf{u} \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t}{\Gamma \vdash_{\mathcal{A}} [\lambda x.\kappa \mid \lambda(\mathbf{u}, \mathbb{k})] : \mathbf{u} \to t}$$

$$[{\to}\textsc{E-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x}_1\mathsf{x}_2 \mid @(\Sigma_1, \Sigma_2)] : t_1 \circ t_2}\ \begin{array}{l} t_1 = \Gamma(\mathsf{x}_1)\Sigma_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\Sigma_2 \\ t_1 \leq \mathbb{0} \to \mathbb{1},\ \ t_2 \leq \mathsf{dom}(t_1) \end{array}$$

$$[{\times}\textsc{I-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}_1, \mathsf{x}_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2}\ t_1 = \Gamma(\mathsf{x}_1)\rho_1,\ \ t_2 = \Gamma(\mathsf{x}_2)\rho_2$$

$$[{\times}\textsc{E}_1\textsc{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_1\mathsf{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_1(t)}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leq (\mathbb{1} \times \mathbb{1}) \end{array}$$

$$[{\times}\textsc{E}_2\textsc{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\pi_2\mathsf{x} \mid \pi(\Sigma)] : \boldsymbol{\pi}_2(t)}\ \begin{array}{l} t = \Gamma(\mathsf{x})\Sigma \\ t \leq (\mathbb{1} \times \mathbb{1}) \end{array}$$

$$[\mathbb{0}\textsc{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}{\in}\tau)\, ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \mathbb{0}(\Sigma)] : \mathbb{0}}\ \Gamma(\mathsf{x})\Sigma \simeq \mathbb{0}$$

$$[\in_1\textsc{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}{\in}\tau)\, ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \in_1(\Sigma)] : \Gamma(\mathsf{x}_1)}\ \Gamma(\mathsf{x})\Sigma \leq \tau$$

$$[\in_2\textsc{-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [(\mathsf{x}{\in}\tau)\, ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \in_2(\Sigma)] : \Gamma(\mathsf{x}_2)}\ \Gamma(\mathsf{x})\Sigma \leq \neg\tau$$

$$[\wedge\textsc{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [a \mid \bigwedge(\{\mathbb{a}_i\}_{i\in I})] : \bigwedge_{i\in I} t_i}\ I \neq \varnothing$$

$$[\textsc{Var-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x} \mid \rho] : \Gamma(\mathsf{x})\rho} \qquad [\textsc{Bind}_1\textsc{-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t}{\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{skip}\,\mathbb{k}] : t}\ \mathsf{x} \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{Bind}_2\textsc{-Alg}]\ \frac{\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [\mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathsf{keep}\,(\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i)\}_{i\in I})] : \bigvee_{i\in I} t_i}\ \bigvee_{i\in I} \mathbf{u}_i \simeq \mathbb{1}$$

$$[\wedge\textsc{-Alg}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}_i] : t_i}{\Gamma \vdash_{\mathcal{A}} [\kappa \mid \bigwedge(\{\mathbb{k}_i\}_{i\in I})] : \bigwedge_{i\in I} t_i}\ I \neq \varnothing$$

To extend the system to type the extensions presented in Appendix A the following rules must be added:

$$[\textsc{Let-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{let}\,\mathsf{x}_1\,\mathsf{in}\,\mathsf{x}_2 \mid \varnothing] : \Gamma(\mathsf{x}_2)}\ \mathsf{x}_1 \in \mathsf{dom}(\Gamma)$$

$$[\textsc{Constr-Alg}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} [\mathsf{x} \mathbin{⨟} \tau \mid ⨟(\Sigma)] : \Gamma(\mathsf{x})}\ \Gamma(\mathsf{x})\Sigma \leq \tau$$

H FULL RECONSTRUCTION SYSTEM

H.1 Main Reconstruction System

$$
\begin{array}{llll}
\textbf{Split annotations} & \mathcal{S} & ::= & \{(\mathbf{u},\mathcal{K}),\ldots,(\mathbf{u},\mathcal{K})\}\\
\textbf{Atoms intermediate annot.} & \mathcal{A} & ::= & \texttt{infer} \mid \texttt{untyp} \mid \texttt{typ} \mid \bigwedge(\{\mathcal{A},\ldots,\mathcal{A}\},\{\mathcal{A},\ldots,\mathcal{A}\})\\
& & & \mid \in_1 \mid \in_2 \mid \lambda(\mathbf{u},\mathcal{K})\\
\textbf{Forms intermediate annot.} & \mathcal{K} & ::= & \texttt{infer} \mid \texttt{untyp} \mid \texttt{typ} \mid \bigwedge(\{\mathcal{K},\ldots,\mathcal{K}\},\{\mathcal{K},\ldots,\mathcal{K}\})\\
& & & \mid \texttt{try-skip}\ (\mathcal{K}) \mid \texttt{try-keep}\ (\mathcal{A},\mathcal{K},\mathcal{K})\\
& & & \mid \texttt{propagate}\ (\mathcal{A},\mathbb{\Gamma},\mathcal{S},\mathcal{S})\\
& & & \mid \texttt{skip}\ (\mathcal{K}) \mid \texttt{keep}\ (\mathcal{A},\mathcal{S},\mathcal{S})
\end{array}
$$

$$
[\textsc{Ok}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \texttt{typ}\rangle \Rightarrow \texttt{Ok}(\texttt{typ})}
\qquad
[\textsc{Fail}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \texttt{untyp}\rangle \Rightarrow \texttt{Fail}}
$$

$$
[\textsc{Const}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle c \mid \texttt{infer}\rangle \Rightarrow \texttt{Ok}(\texttt{typ})}
$$

$$
[\textsc{AxOk}]\ \frac{x \in \operatorname{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle x \mid \texttt{infer}\rangle \Rightarrow \texttt{Ok}(\texttt{typ})}
\qquad
[\textsc{AxFail}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle x \mid \texttt{infer}\rangle \Rightarrow \texttt{Fail}}
$$

$$
[\textsc{PairVar}_i]\ \frac{\mathrm{x}_i \notin \operatorname{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathrm{x}_1,\mathrm{x}_2) \mid \texttt{infer}\rangle \Rightarrow \texttt{Var}\ (\mathrm{x}_i,\texttt{infer},\texttt{untyp})}
$$

$$
[\textsc{PairOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle (\mathrm{x}_1,\mathrm{x}_2) \mid \texttt{infer}\rangle \Rightarrow \texttt{Ok}(\texttt{typ})}
$$

$$
[\textsc{ProjVar}]\ \frac{\mathrm{x} \notin \operatorname{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \pi_i \mathrm{x} \mid \texttt{infer}\rangle \Rightarrow \texttt{Var}\ (\mathrm{x},\texttt{infer},\texttt{untyp})}
$$

$$
[\textsc{ProjInfer}]\ \frac{\Psi = \text{tally\_infer}(\{\Gamma(\mathrm{x}) \mathrel{\dot{\leq}} \alpha \times \beta\})}{\Gamma \vdash_{\mathcal{R}} \langle \pi_i \mathrm{x} \mid \texttt{infer}\rangle \Rightarrow \texttt{Subst}(\Psi,\texttt{typ},\texttt{untyp})}\ \alpha,\beta \in \mathcal{V}_P \text{ fresh}
$$

$$
[\textsc{AppVar}_i]\ \frac{\mathrm{x}_i \notin \operatorname{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathrm{x}_1\mathrm{x}_2 \mid \texttt{infer}\rangle \Rightarrow \texttt{Var}\ (\mathrm{x}_i,\texttt{infer},\texttt{untyp})}
$$

$$
[\textsc{AppInfer}]\ \frac{\Psi = \text{tally\_infer}(\{\Gamma(\mathrm{x}_1) \mathrel{\dot{\leq}} \Gamma(\mathrm{x}_2) \to \alpha\})}{\Gamma \vdash_{\mathcal{R}} \langle \mathrm{x}_1\mathrm{x}_2 \mid \texttt{infer}\rangle \Rightarrow \texttt{Subst}(\Psi,\texttt{typ},\texttt{untyp})}\ \alpha \in \mathcal{V}_P \text{ fresh}
$$

$$
[\textsc{CaseVar}]\ \frac{\mathrm{x} \notin \operatorname{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathrm{x}{\in}\tau)\,?\,\mathrm{x}_1 : \mathrm{x}_2 \mid \texttt{infer}\rangle \Rightarrow \texttt{Var}\ (\mathrm{x},\texttt{infer},\texttt{untyp})}
$$

$$
[\textsc{CaseSplit}]\ \frac{\Gamma(\mathrm{x}) \not\leq \tau \qquad \Gamma(\mathrm{x}) \not\leq \neg\tau}{\Gamma \vdash_{\mathcal{R}} \langle (\mathrm{x}{\in}\tau)\,?\,\mathrm{x}_1 : \mathrm{x}_2 \mid \texttt{infer}\rangle \Rightarrow \texttt{Split}(\{(\mathrm{x}:\tau)\},\texttt{infer},\texttt{infer})}
$$

$$[\textsc{CaseEmpty}]\ \frac{\Gamma(\mathsf{x}) \simeq \mathbb{0}}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Ok}(\mathsf{typ})}$$

$$[\textsc{CaseThen}]\ \frac{\Gamma(\mathsf{x}) \leq \tau \qquad \Psi = \mathsf{tally\_infer}(\{\Gamma(\mathsf{x}) \dot{\leq} \mathbb{0}\})}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathsf{typ}, \in_1)}$$

$$[\textsc{CaseElse}]\ \frac{\Gamma(\mathsf{x}) \leq \neg\tau \qquad \Psi = \mathsf{tally\_infer}(\{\Gamma(\mathsf{x}) \dot{\leq} \mathbb{0}\})}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Subst}(\Psi, \mathsf{typ}, \in_2)}$$

$$[\textsc{CaseVar}_i]\ \frac{\mathsf{x}_i \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_i\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathsf{typ}, \mathsf{untyp})}$$

$$[\textsc{CaseOk}_i]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle (\mathsf{x}{\in}\tau)\,?\,\mathsf{x}_1 : \mathsf{x}_2 \mid \in_i\rangle \Rightarrow \mathtt{Ok}(\mathsf{typ})}$$

$$[\textsc{LambdaInfer}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\boldsymbol{\alpha}, \mathtt{infer})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \mathtt{infer}\rangle \Rightarrow \mathbb{R}}\ \boldsymbol{\alpha} \in \mathcal{V}_M \text{ fresh}$$

$$[\textsc{LambdaEmpty}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\mathbb{0}, \mathcal{K})\rangle \Rightarrow \mathtt{Fail}}$$

$$[\textsc{Lambda}]\ \frac{\Gamma, x : \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K})\rangle \Rightarrow \mathtt{map}(X \mapsto \lambda(\mathbf{u}, X),\ \mathbb{R})}$$

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

$$\begin{aligned}
\mathtt{map}(X \mapsto f(X),\ \mathtt{Ok}(\mathcal{H})) &\overset{\text{def}}{=} \mathtt{Ok}(f(\mathcal{H}))\\
\mathtt{map}(X \mapsto f(X),\ \mathtt{Fail}) &\overset{\text{def}}{=} \mathtt{Fail}\\
\mathtt{map}(X \mapsto f(X),\ \mathtt{Split}(\Gamma, \mathcal{H}_1, \mathcal{H}_2)) &\overset{\text{def}}{=} \mathtt{Split}(\Gamma, f(\mathcal{H}_1), f(\mathcal{H}_2))\\
\mathtt{map}(X \mapsto f(X),\ \mathtt{Subst}(\Psi, \mathcal{H}_1, \mathcal{H}_2)) &\overset{\text{def}}{=} \mathtt{Subst}(\Psi, f(\mathcal{H}_1), f(\mathcal{H}_2))\\
\mathtt{map}(X \mapsto f(X),\ \mathtt{Var}\ (\mathsf{x}, \mathcal{H}_1, \mathcal{H}_2)) &\overset{\text{def}}{=} \mathtt{Var}\ (\mathsf{x}, f(\mathcal{H}_1), f(\mathcal{H}_2))
\end{aligned}$$

$$[\textsc{FormVar}]\ \frac{\mathsf{x} \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathsf{untyp})}$$

$$[\textsc{FormOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \mid \mathtt{infer}\rangle \Rightarrow \mathtt{Ok}(\mathsf{typ})}$$

$$[\textsc{BindInfer}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathtt{try\text{-}skip}\ (\mathtt{infer})\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\,\mathsf{x} = a\,\mathsf{in}\,\kappa \mid \mathtt{infer}\rangle \Rightarrow \mathbb{R}}$$

$$[\text{BindTrySkip}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2) \qquad \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathtt{infer}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathbb{R}}$$

$$[\text{BindTrySkip}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Ok}(\mathcal{K}')}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathtt{Ok}(\mathtt{skip}\ (\mathcal{K}'))}$$

$$[\text{BindTrySkip}_3]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}skip}\ (\mathcal{K}) \rangle \Rightarrow \mathsf{map}(X \mapsto \mathtt{try\text{-}skip}\ (X),\ \mathbb{R})}$$

$$[\text{BindSkip}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathcal{K}_1, \mathcal{K}_2) \qquad \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{skip}\ (\mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{skip}\ (\mathcal{K}) \rangle \Rightarrow \mathbb{R}}$$

$$[\text{BindSkip}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{skip}\ (\mathcal{K}) \rangle \Rightarrow \mathsf{map}(X \mapsto \mathtt{skip}\ (X),\ \mathbb{R})}$$

$$[\text{BindTryKeep}_1]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathtt{Ok}(\mathcal{A}') \qquad \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}', \{(\mathbb{1}, \mathcal{K}_1)\}, \varnothing) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}$$

$$[\text{BindTryKeep}_2]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathtt{Fail} \qquad \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{skip}\ (\mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}}$$

$$[\text{BindTryKeep}_3]\ \frac{\Gamma \vdash^*_{\mathcal{R}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2) \rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \mathsf{map}(X \mapsto \mathtt{try\text{-}keep}\ (X, \mathcal{K}_1, \mathcal{K}_2),\ \mathbb{R})$.

$$[\text{BindOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \varnothing, \mathcal{S}) \rangle \Rightarrow \mathtt{Ok}(\mathtt{keep}\ (\mathcal{A}, \varnothing, \mathcal{S}))}$$

$$[\text{BindKeep}_1]\ \frac{\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Ok}(\mathcal{K}') \qquad \Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \{(\mathbf{u}, \mathcal{K}')\} \cup \mathcal{S}') \rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathbb{R}}$$

$$[\text{BindKeep}_2]\ \frac{\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathsf{x} : s \wedge \mathbf{u} \vdash^*_{\mathcal{R}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathtt{Split}(\Gamma', \mathcal{K}_1, \mathcal{K}_2) \qquad \mathsf{x} \in \mathrm{dom}(\Gamma') \qquad \Gamma \vdash_{\mathcal{E}} (a : \neg(\mathbf{u} \wedge \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_1 \qquad \Gamma \vdash_{\mathcal{E}} (a : \neg(\mathbf{u} \setminus \Gamma'(\mathsf{x}))) \Rightarrow \mathbb{\Gamma}_2}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{bind}\, \mathsf{x} = a\, \mathsf{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}') \rangle \Rightarrow \mathtt{Split}(\Gamma' \setminus \mathsf{x}, \mathcal{K}_1', \mathcal{K}_2')}$$

where:

- $\mathcal{K}_1' = \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}_1 \cup \mathbb{\Gamma}_2, \{(\mathbf{u} \wedge \Gamma'(\mathsf{x}), \mathcal{K}_1),\ (\mathbf{u} \setminus \Gamma'(\mathsf{x}), \mathcal{K}_2)\} \cup \mathcal{S}, \mathcal{S}')$
- $\mathcal{K}_2' = \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K}_2)\} \cup \mathcal{S}, \mathcal{S}')$

$$[\textsc{BindKeep}_3]\ \frac{\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A}\rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad \Gamma, \mathrm{x} : s \wedge \mathbf{u} \vdash^{*}_{\mathcal{R}} \langle \kappa \mid \mathcal{K}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathrm{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, \mathcal{K})\} \cup \mathcal{S}, \mathcal{S}')\rangle \Rightarrow \mathbb{R}'}$$

where $\mathbb{R}' = \mathtt{map}(X \mapsto \mathtt{keep}\ (\mathcal{A}, \{(\mathbf{u}, X)\} \cup \mathcal{S}, \mathcal{S}'),\ \mathbb{R})$.

$$[\textsc{BindProp}_1]\ \frac{\Gamma' \in \mathbb{\Gamma} \qquad \mathtt{compatible}(\Gamma, \Gamma')}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathrm{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S}')\rangle \Rightarrow \mathtt{Split}(\Gamma'', \mathcal{K}_1, \mathcal{K}_2)}$$

where:

- $\mathtt{compatible}(\Gamma, \Gamma') \Leftrightarrow (\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma))$ and $(\forall \mathrm{x} \in \mathrm{dom}(\Gamma').\ (\Gamma(\mathrm{x}) \wedge \Gamma'(\mathrm{x}) \not\simeq \mathbb{0})$ or $(\Gamma(\mathrm{x}) \simeq \mathbb{0}))$
- $\Gamma'' = \{(\mathrm{x} : \mathbf{u}) \in \Gamma' \mid \Gamma(\mathrm{x}) \not\leq \mathbf{u}\}$
- $\mathcal{K}_1 = \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')$
- $\mathcal{K}_2 = \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma} \setminus \{\Gamma'\}, \mathcal{S}, \mathcal{S}')$

$$[\textsc{BindProp}_2]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathrm{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{keep}\ (\mathcal{A}, \mathcal{S}, \mathcal{S}')\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{bind}\, \mathrm{x} = a\, \mathtt{in}\, \kappa \mid \mathtt{propagate}\ (\mathcal{A}, \mathbb{\Gamma}, \mathcal{S}, \mathcal{S}')\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{InterEmpty}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(\varnothing, \varnothing)\rangle \Rightarrow \mathtt{Fail}}$$

$$[\textsc{InterOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(\varnothing, S)\rangle \Rightarrow \mathtt{Ok}(\bigwedge(\varnothing, S))}$$

$$[\textsc{Inter}_1]\ \frac{\Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathtt{Ok}(\mathcal{H}') \qquad \Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(S, \{\mathcal{H}'\} \cup S')\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(\{\mathcal{H}\} \cup S, S')\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Inter}_2]\ \frac{\Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathtt{Fail} \qquad \Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(S, S')\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(\{\mathcal{H}\} \cup S, S')\rangle \Rightarrow \mathbb{R}}$$

$$[\textsc{Inter}_3]\ \frac{\Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \ \bigwedge(\{\mathcal{H}\} \cup S, S')\rangle \Rightarrow \mathtt{map}(X \mapsto (\bigwedge(\{X\} \cup S, S')),\ \mathbb{R})}$$

$$[\textsc{Iterate}_1]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathtt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2) \qquad \Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}_1\rangle \Rightarrow \mathbb{R}'}{\Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathbb{R}'}\ \Gamma' = \varnothing$$

$$[\textsc{Iterate}_2]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathtt{Subst}(\{\psi_i\}_{i \in I}, \mathcal{H}_1, \mathcal{H}_2) \\ \Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \ \bigwedge(\{\mathcal{H}_1\psi_i\}_{i \in I} \cup \{\mathcal{H}_2\}, \varnothing)\rangle \Rightarrow \mathbb{R}'\end{array}}{\Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathbb{R}'}\ \forall i \in I.\ \psi_i \# \Gamma$$

$$[\textsc{Stop}]\ \frac{\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathbb{R}}{\Gamma \vdash^{*}_{\mathcal{R}} \langle \eta \mid \mathcal{H}\rangle \Rightarrow \mathbb{R}}$$

with $\mathcal{H}\psi$ denoting the intermediate annotation $\mathcal{H}$ in which the substitution ψ has been applied recursively to every type (in λ-abstraction annotations and binding annotations).

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

$$[\textsc{LetVar}_i]\ \frac{\mathsf{x}_i \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{let}\, \mathsf{x}_1\, \mathtt{in}\, \mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}_i, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{LetOk}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} \langle \mathtt{let}\, \mathsf{x}_1\, \mathtt{in}\, \mathsf{x}_2 \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Ok}(\mathtt{typ})}$$

$$[\textsc{ConstrVar}]\ \frac{\mathsf{x} \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \mathbin{⨾} \tau \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Var}\ (\mathsf{x}, \mathtt{infer}, \mathtt{untyp})}$$

$$[\textsc{ConstrInfer}]\ \frac{\Psi = \mathsf{tally\_infer}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leq}} \tau\})}{\Gamma \vdash_{\mathcal{R}} \langle \mathsf{x} \mathbin{⨾} \tau \mid \mathtt{infer} \rangle \Rightarrow \mathtt{Subst}(\Psi, \mathtt{typ}, \mathtt{untyp})}$$

H.2 Auxiliary Reconstruction System

In the following, $\mathtt{refresh}(t)$ denotes a renaming from $\mathrm{vars}(t) \cap \mathcal{V}_P$ to fresh polymorphic variables.

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash_{\mathcal{P}} \langle c \mid \mathtt{typ} \rangle \Rightarrow \varnothing} \qquad [\textsc{Ax}]\ \frac{}{\Gamma \vdash_{\mathcal{P}} \langle x \mid \mathtt{typ} \rangle \Rightarrow \varnothing}\ x \in \mathrm{dom}(\Gamma)$$

$$[\textsc{Pair}]\ \frac{\rho_1 = \mathtt{refresh}(\Gamma(\mathsf{x}_1)) \qquad \rho_2 = \mathtt{refresh}(\Gamma(\mathsf{x}_2))}{\Gamma \vdash_{\mathcal{P}} \langle (\mathsf{x}_1, \mathsf{x}_2) \mid \mathtt{typ} \rangle \Rightarrow (\rho_1, \rho_2)}$$

$$[\textsc{Proj}]\ \frac{\Sigma = \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leq}} \alpha \times \beta\})}{\Gamma \vdash_{\mathcal{P}} \langle \pi_i \mathsf{x} \mid \mathtt{typ} \rangle \Rightarrow \pi(\Sigma)}\ \begin{array}{l}\Sigma \neq \varnothing \\ \alpha, \beta \in \mathcal{V}_P \text{ fresh}\end{array}$$

$$[\textsc{App}]\ \frac{\begin{array}{c} t_1 = \Gamma(\mathsf{x}_1) \qquad t_2 = \Gamma(\mathsf{x}_2) \\ \rho_1 = \mathtt{refresh}(t_1) \qquad \rho_2 = \mathtt{refresh}(t_2) \qquad \Sigma = \mathsf{tally}(\{t_1\rho_1 \mathrel{\dot{\leq}} t_2\rho_2 \to \alpha\}) \end{array}}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{x}_1 \mathsf{x}_2 \mid \mathtt{typ} \rangle \Rightarrow @(\{\sigma \circ \rho_1 \mid \sigma \in \Sigma\}, \{\sigma \circ \rho_2 \mid \sigma \in \Sigma\})}\ \begin{array}{l}\Sigma \neq \varnothing \\ \alpha \in \mathcal{V}_P \text{ fresh}\end{array}$$

$$[\textsc{Case}_{\mathbb{0}}]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leq}} \mathbb{0}\})}{\Gamma \vdash_{\mathcal{P}} \langle (\mathsf{x} \in \tau)\ ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{typ} \rangle \Rightarrow \mathbb{0}(\{\sigma\})}$$

$$[\textsc{Case}_1]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leq}} \tau\})}{\Gamma \vdash_{\mathcal{P}} \langle (\mathsf{x} \in \tau)\ ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{typ} \rangle \Rightarrow \in_1(\{\sigma\})}\ \mathsf{x}_1 \in \mathrm{dom}(\Gamma)$$

$$[\textsc{Case}_2]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leq}} \neg\tau\})}{\Gamma \vdash_{\mathcal{P}} \langle (\mathsf{x} \in \tau)\ ?\, \mathsf{x}_1 : \mathsf{x}_2 \mid \mathtt{typ} \rangle \Rightarrow \in_2(\{\sigma\})}\ \mathsf{x}_2 \in \mathrm{dom}(\Gamma)$$

$$[\textsc{Lambda}]\ \frac{\Gamma, x : \mathbf{u} \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{k}}{\Gamma \vdash_{\mathcal{P}} \langle \lambda x.\kappa \mid \lambda(\mathbf{u}, \mathcal{K}) \rangle \Rightarrow \lambda(\mathbf{u}, \mathbb{k})}$$

$$[\textsc{Var}]\ \frac{\rho = \mathsf{refresh}(\Gamma(\mathsf{x}))}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{x} \mid \mathsf{typ} \rangle : \rho} \qquad [\textsc{BindSkip}]\ \frac{\Gamma \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{k}}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{bind}\, \mathsf{x} = a \,\mathsf{in}\, \kappa \mid \mathsf{skip}\ (\mathcal{K}) \rangle \Rightarrow \mathsf{skip}\ \mathbb{k}}\ \mathsf{x} \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{BindKeep}]\ \frac{\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a} \qquad \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : s \qquad (\forall i \in I)\ \ \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K}_i \rangle \Rightarrow \mathbb{k}_i}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{bind}\, \mathsf{x} = a \,\mathsf{in}\, \kappa \mid \mathsf{keep}\ (\mathcal{A}, \varnothing, \{(\mathbf{u}_i, \mathcal{K}_i)\}_{i \in I}) \rangle \Rightarrow \mathsf{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i)\}_{i \in I})}\ (*)$$

where $(*)$ is $\bigvee_{i \in I} \mathbf{u}_i \simeq \mathbb{1}$.

$$[\textsc{Inter}]\ \frac{(\forall i \in I) \quad \Gamma \vdash_{\mathcal{P}} \langle \eta \mid \mathcal{H}_i \rangle \Rightarrow \mathbb{h}_i}{\Gamma \vdash_{\mathcal{P}} \langle \eta \mid \ \bigwedge(\varnothing, \{\mathcal{H}_i\}_{i \in I}) \rangle \Rightarrow \bigwedge(\{\mathbb{h}_i\}_{i \in I})}\ I \neq \varnothing$$

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

$$[\textsc{Let}]\ \frac{}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{let}\, \mathsf{x}_1 \,\mathsf{in}\, \mathsf{x}_2 \mid \mathsf{typ} \rangle \Rightarrow \varnothing} \qquad [\textsc{Constr}]\ \frac{\sigma \in \mathsf{tally}(\{\Gamma(\mathsf{x}) \mathrel{\dot{\leq}} \tau\})}{\Gamma \vdash_{\mathcal{P}} \langle \mathsf{x} \mathbin{⨾} \tau \mid \mathsf{typ} \rangle \Rightarrow ⨾(\{\sigma\})}$$

H.3 Split Propagation System

The split propagation system defined in this section tries to deal with the following problem: *given an environment* Γ*, an atom* a *and a type* t*, what additional assumptions can be made on* Γ *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.

$$[\textsc{Const}_1]\ \frac{b_c \leq \mathbf{u}}{\Gamma \vdash_{\mathcal{E}} (c : \mathbf{u}) \Rightarrow \{\varnothing\}} \qquad [\textsc{Const}_2]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (c : \mathbf{u}) \Rightarrow \{\}}$$

$$[\textsc{Ax}_1]\ \frac{\Gamma(x) \leq \mathbf{u}}{\Gamma \vdash_{\mathcal{E}} (x : \mathbf{u}) \Rightarrow \{\varnothing\}} \qquad [\textsc{Ax}_2]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (x : \mathbf{u}) \Rightarrow \{\}}$$

$$[\textsc{Proj}_1]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (\pi_1 \mathsf{x} : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \mathbf{u} \times \mathbb{1}\}\}} \qquad [\textsc{Proj}_2]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (\pi_2 \mathsf{x} : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \mathbb{1} \times \mathbf{u}\}\}}$$

$$[\textsc{Pair}]\ \frac{\mathbf{u} \overset{\text{DNF}}{\simeq} (\bigvee_{i \in I} (\mathbf{u}_i \times \mathbf{v}_i)) \vee \ldots}{\Gamma \vdash_{\mathcal{E}} ((\mathsf{x}_1, \mathsf{x}_2) : \mathbf{u}) \Rightarrow \{\{\mathsf{x}_1 : \mathbf{u}_i\} \wedge \{\mathsf{x}_2 : \mathbf{v}_i\} \mid i \in I\}}$$

$$[\textsc{Case}]\ \frac{}{\Gamma \vdash_{\mathcal{E}} ((\mathsf{x} \in \tau)\, ?\, \mathsf{x}_1 \mathbin{\mathbf{:}} \mathsf{x}_2 : \mathbf{u}) \Rightarrow \{\{\mathsf{x} : \tau,\ \mathsf{x}_1 : \mathbf{u}\}, \{\mathsf{x} : \neg\tau,\ \mathsf{x}_2 : \mathbf{u}\}\}}$$

$$[\textsc{App}]\ \frac{\Gamma(\mathsf{x}_1) \overset{\text{DNF}}{\simeq} \bigvee_{i \in I} t_i \qquad \forall i \in I.\ \{\sigma_j\}_{j \in J_i} = \mathsf{tally}(\{t_i \mathrel{\dot{\leq}} \alpha \to \mathbf{u}\})}{\Gamma \vdash_{\mathcal{E}} (\mathsf{x}_1 \mathsf{x}_2 : \mathbf{u}) \Rightarrow \bigcup_{i \in I} \mathbb{\Gamma}_i}\ \alpha \in \mathcal{V}_P \text{ fresh}$$

where, for every $i \in I$, $\mathbb{\Gamma}_i = \{\{\mathsf{x}_1 : (t_i\sigma_j)\sigma'_j,\ \mathsf{x}_2 : (\alpha\sigma_j)\sigma'_j\} \mid j \in J_i\}$ with σ'_j a type substitution mapping each polymorphic type variable β appearing in $t_i\sigma_j$ or $\alpha\sigma_j$ to either:

- $\mathbb{1}$ if β only appears in covariant positions in $\alpha\sigma_j$,
- $\mathbb{0}$ if β only appears in contravariant positions in $\alpha\sigma_j$,
- a fresh monomorphic type variable otherwise.

$$[\textsc{Lambda}]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (\lambda x.\, \kappa : \mathbf{u}) \Rightarrow \{\}}$$

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

$$[\textsc{Let}]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (\mathtt{let}\, \mathtt{x}_1\, \mathtt{in}\, \mathtt{x}_2\ : \mathbf{u}) \Rightarrow \{\{\mathtt{x}_2 : \mathbf{u}\}\}} \qquad [\textsc{Constr}]\ \frac{}{\Gamma \vdash_{\mathcal{E}} (\mathtt{x} \,\,\mathring{\circ}\,\, \tau : \mathbf{u}) \Rightarrow \{\{\mathtt{x} : \mathbf{u}\}\}}$$

I PROOFS

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

$$\textbf{Expressions} \quad e \quad ::= \quad c \mid x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e : e$$

We fix some notations relative to substitutions:

- ϕ ranges over substitutions from type variables ($\mathcal{V}_P \dot{\cup} \mathcal{V}_M$) to types
- ρ ranges over renamings of polymorphic variables, that is, injective substitutions from $\mathcal{V}_P$ to $\mathcal{V}_P$
- σ ranges over substitutions from polymorphic type variables $\mathcal{V}_P$ to types
- Σ ranges over sets of substitutions from polymorphic type variables $\mathcal{V}_P$ to types
- ψ ranges over substitutions from monomorphic type variables $\mathcal{V}_M$ to monomorphic types
- Ψ ranges over sets of substitutions from monomorphic type variables $\mathcal{V}_M$ to monomorphic types

I.1 A Canonical Form for the Derivations

Derivations for the declarative type system can have many shapes (see Appendix D for the full declarative system, without the rules for extensions). In particular, the union elimination rule [$\vee$E] can be used anywhere in the derivation and changes the expression to type by performing a substitution on it. Other non-structural rules such as [$\wedge$], [$\leq$] and [Inst] can also be applied anywhere in the derivation. In this section, we will define canonical derivations that restrict the use of those rules. This will then be used, in Section I.2, to establish a type safety theorem.

I.1.1 Alternative Form of the Declarative Type System. In order to be able to express our normalisations lemmas, we first need to slightly modify some rules of the declarative type system. In order to avoid confusions, the modified declarative type system will use this turnstile symbol: $\Vdash$.

First, we modify the [Ax] rule so that it can perform a renaming of the polymorphic type variables in $\Gamma(x)$:

$$[\textsc{Ax}]\ \frac{}{\Gamma \Vdash x : \Gamma(x)\rho}$$

This new [Ax] rule is derivable in the initial delcarative type system by composing a [Ax] rule and a [Inst] rule. Still, allowing the [Ax] rule to perform a renaming of polymorphic type variables is useful, as it allows to uncorrelate types without resorting to the [Inst] rule. For instance, consider the pair (x, x) with x having the type $\alpha \to \alpha$. While this pair could be typed $(\alpha \to \alpha)\times(\alpha \to \alpha)$, this type does not allow instantiating the left-hand side and right-hand side of the product independently. A better type would be $(\alpha \to \alpha) \times (\beta \to \beta)$, and with this new [Ax] rule, it can be derived without having to use a [Inst] rule. This way, the [Inst] rule can be reserved to cases that require non trivial instantiations (i.e. not just renamings). Note that the necessity of performing this renaming comes from the fact that we do not use type schemes $\forall\vec{\alpha}.\, t$, where renaming of the type variables in $\vec{\alpha}$ can be performed implicitely anywhere.

Secondly, we use a $[\wedge]$ rule of multiple arity instead of a binary one:

$$[\wedge]\ \frac{(\forall i \in I) \quad \Gamma \Vdash e : t_i}{\Gamma \Vdash e : \bigwedge_{i\in I} t_i}\ I \neq \varnothing$$

This allows to combine successive $[\wedge]$ rule applications into one $[\wedge]$ rule, making the normalisation lemmas easier to express. This new $[\wedge]$ rule is admissible in the $\vdash$ system: it can be replaced by several consecutive $[\wedge]$ nodes.

Similarly, we will use a $[\vee]$ rule of multiple arity:

$$[\vee]\ \frac{\Gamma \Vdash e' : s \qquad (\forall i \in I) \quad \Gamma, x : s \wedge \mathbf{u}_i \Vdash e : t}{\Gamma \Vdash e\{e'/x\} : t}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

with $\mathsf{Part}(t)$ denoting the set of partitions of the type t, that is, the set of all sets $\{t_i\}_{i\in I}$ such that: (i) $\bigvee_{i\in I} t_i \simeq t$, (ii) $\forall i \in I.\ t_i \not\simeq \mathbb{0}$, and (iii) $\forall i, j \in I.\ i \neq j \Rightarrow t_i \wedge t_j \simeq \mathbb{0}$. The guard condition is most of time omitted, for concision.

This allows to combine successive $[\vee]$ rule applications substituting the same sub-expression into one $[\vee]$ rule, making the normalisation lemmas easier to express. Again, this new $[\vee]$ rule is admissible. For instance, the following derivation:

$$[\vee]\ \frac{\dfrac{A}{\Gamma \Vdash e' : s} \quad \dfrac{B}{\Gamma, y : s \wedge \mathbf{u}_1 \Vdash e : t} \quad \dfrac{C}{\Gamma, y : s \wedge \mathbf{u}_2 \Vdash e : t} \quad \dfrac{D}{\Gamma, y : s \wedge \mathbf{u}_3 \Vdash e : t}}{\Gamma \Vdash e\{e'/x\} : t}$$

can be transformed to use only two binary $[\vee]$ rules:

$$[\vee]\ \frac{\dfrac{A}{\Gamma \Vdash e' : s} \quad \dfrac{B}{\Gamma, x : s \wedge \mathbf{u}_1 \Vdash e : t} \quad \dfrac{X}{\Gamma, x : s \wedge \neg\mathbf{u}_1 \Vdash e : t}}{\Gamma \Vdash e\{e'/x\} : t}$$

with X being the following derivation:

$$[\vee]\ \frac{[\mathrm{Ax}] \quad \dfrac{C\{y/x\}}{\Gamma, x : s \wedge \neg\mathbf{u}_1, y : s \wedge \mathbf{u}_2 \Vdash e\{y/x\} : t} \quad \dfrac{D\{y/x\}}{\Gamma, x : s \wedge \neg\mathbf{u}_1, y : s \wedge \mathbf{u}_3 \Vdash e\{y/x\} : t}}{\Gamma, x : s \wedge \neg\mathbf{u}_1 \Vdash (e\{y/x\})\{x/y\} : t}$$

This construction can be generalized for a partition of $\mathbb{1}$ of any cardinality.

Lastly, we distinguish variables that are introduced by a $[\to\mathrm{I}]$ node from variables introduced by a $[\vee]$ node. The formers are called *lambda variables*, the set of all lambda variables is denoted by Vars_λ and ranged over by x, y, and z. The latters are called *binding variables*, the set of all binding variables is denoted by Vars_B and ranged over by x, y, and z. Vars_λ and Vars_B form a partition of the set of variables Vars (formally, $\mathsf{Vars} = \mathsf{Vars}_\lambda \dot\cup \mathsf{Vars}_\mathsf{B}$). The syntax of expressions and the rules of the type system are changed accordingly as follows:

$$\textbf{Expressions} \quad e \quad ::= \quad c \mid x \mid \mathsf{x} \mid \lambda x.e \mid ee \mid (e, e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e : e$$

$$[\mathrm{Ax}_\lambda]\ \frac{}{\Gamma \Vdash x : \Gamma(x)\rho} \qquad [\mathrm{Ax}_\vee]\ \frac{}{\Gamma \Vdash \mathsf{x} : \Gamma(\mathsf{x})\rho}$$

$$[\vee]\ \frac{\Gamma \Vdash e' : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \Vdash e : t}{\Gamma \Vdash e\{e'/\mathsf{x}\} : t}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

When needed, we will use the notation ${}^x\!/\!_\mathsf{x}$ to range over both binding variables and lambda variables. For instance, we could write the proposition $\forall {}^x\!/\!_\mathsf{x} \in \mathsf{dom}(\Gamma).\ \Gamma({}^x\!/\!_\mathsf{x}) \not\simeq \mathbb{0}$. We say that an

$$[\textsc{Const}]\ \frac{}{\Gamma \Vdash c : b_c} \qquad [\mathrm{Ax}_\lambda]\ \frac{}{\Gamma \Vdash x : \Gamma(x)\rho} \qquad [\mathrm{Ax}_\vee]\ \frac{}{\Gamma \Vdash \mathsf{x} : \Gamma(\mathsf{x})\rho}$$

$$[\to\mathrm{I}]\ \frac{\Gamma, x : \mathbf{u} \Vdash e : t}{\Gamma \Vdash \lambda x.e : \mathbf{u} \to t} \qquad [\to\mathrm{E}]\ \frac{\Gamma \Vdash e_1 : t_1 \to t_2 \qquad \Gamma \Vdash e_2 : t_1}{\Gamma \Vdash e_1 e_2 : t_2}$$

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

$$[\mathbb{0}]\ \frac{\Gamma \Vdash e : \mathbb{0}}{\Gamma \Vdash (e{\in}\tau)\ ?\ e_1 : e_2 : \mathbb{0}} \qquad [\in_1]\ \frac{\Gamma \Vdash e : \tau \qquad \Gamma \Vdash e_1 : t_1}{\Gamma \Vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \Vdash e : \neg\tau \qquad \Gamma \Vdash e_2 : t_2}{\Gamma \Vdash (e{\in}\tau)\ ?\ e_1 : e_2 : t_2}$$

$$[\vee]\ \frac{\Gamma \Vdash e' : s \qquad (\forall i \in I) \quad \Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \Vdash e : t}{\Gamma \Vdash e\{e'/\mathsf{x}\} : t}\ \{\mathbf{u}_i\}_{i\in I} \in \mathsf{Part}(\mathbb{1})$$

$$[\wedge]\ \frac{(\forall i \in I) \quad \Gamma \Vdash e : t_i}{\Gamma \Vdash e : \bigwedge_{i\in I} t_i}\ I \neq \varnothing \qquad [\textsc{Inst}]\ \frac{\Gamma \Vdash e : t}{\Gamma \Vdash e : t\sigma} \qquad [\leq]\ \frac{\Gamma \Vdash e : t}{\Gamma \Vdash e : t'}\ t \leq t'$$

Fig. 4. Alternative Declarative Type System

expression e is a ground expression if e does not contain any binding variable, and that a derivation D is a ground derivation if it derives a judgement for a ground expression. For what concerns programs, they use lambda variables for top-level definitions, and are only composed of ground expressions.

This new system is equivalent to the initial type system: the combination of both $[\mathrm{Ax}_\lambda]$ and $[\mathrm{Ax}_\vee]$ gives the previous [Ax] rule.

A full declarative type system with these modifications is presented in Figure 4.

The rules [Const], $[\mathrm{Ax}_\lambda]$, $[\to\mathrm{I}]$, $[\to\mathrm{E}]$, $[\times\mathrm{I}]$, $[\times\mathrm{E}_1]$, $[\times\mathrm{E}_2]$, $[\mathbb{0}]$, $[\in_1]$ and $[\in_2]$ will be called *structural rules* as their use is guided by the structure of the expression to type, each of them allowing to type a specific syntactic construction. In particular, note that the rule $[\mathrm{Ax}_\vee]$ is not considered structural as binding variables x are not supposed to appear in the initial expression (they are only introduced in the derivation when using a $[\vee]$ rule).

Also, the first premise of a $[\vee]$ rule will be called its *definition premise*, and its others premises will be called *body premises*.

All the proofs in the next sections and chapters will use the $\Vdash$ declarative type system, which is equivalent to the $\vdash$ type system.

Proposition I.1. *For any ground expression e, type environment Γ and type t:*

$$\Gamma \vdash e : t \Leftrightarrow \Gamma \Vdash e : t$$

Proof. The $\Rightarrow$ direction is trivial. The $\Leftarrow$ direction is obtained by using [Inst] nodes to rename polymorphic type variables of axioms whenever needed, and by locally transforming n-ary $[\wedge]$ nodes into $n-1$ binary $[\wedge]$ nodes, and n-ary $[\vee]$ nodes into $n-1$ binary $[\vee]$ nodes, as detailled above. □

Now, we introduce a new order $\leq_{\mathrm{P}}$ on types. Intuitively, it adds to the subtyping order $\leq$ the possibility to instantiate polymorphic type variables. We then use it in the statement of a monotonicity lemma that will be used extensively in the next sections.

Definition I.2 (Polymorphic subtyping order). *We define the order relation $\leq_P$ over types as follows:*

$$\forall t_1, t_2.\ t_1 \leq_P t_2 \Leftrightarrow \exists \Sigma.\ t_1\Sigma \leq t_2$$

Note that, while this order will be extensively used in the proofs, it will not be used in algorithms as we have no way to compute it. Indeed, while deciding this order might seem equivalent to solving a tallying problem (defined in Section 4.1), it is actually not the case as deciding this order requires to find a set of substitutions Σ, and not a single substitution σ.

Definition I.3. *For any order relation $\leq$ over types, we define the order relation $\leq$ over environments as follows:*

$$\forall \Gamma_1, \Gamma_2.\ \Gamma_1 \leq \Gamma_2 \Leftrightarrow \forall {}^x\!/\!_x \in dom(\Gamma_2).\ {}^x\!/\!_x \in dom(\Gamma_1) \text{ and } \Gamma_1(x) \leq \Gamma_2(x)$$

We introduce for convenience a new notation that takes the form of a new rule [Inst∧≤], but is actually just a shortand for a specific combination of [Inst], [∧], and [≤] rules:

$$[\text{Inst}\wedge\leq]\ \frac{\dfrac{A}{\Gamma \vdash e : t'} \qquad (\exists\Sigma.\ \bigwedge_{\sigma\in\Sigma} t'\sigma \leq t)}{\Gamma \vdash e : t} \quad \leftrightarrow \quad [\leq]\ \frac{[\wedge]\ \dfrac{[\text{Inst}]\ \dfrac{\dfrac{A}{\Gamma \vdash e : t'}}{\Gamma \vdash e : t'\sigma}\ \forall\sigma\in\Sigma}{\Gamma \vdash e : \bigwedge_{\sigma\in\Sigma} t'\sigma}}{\Gamma \vdash e : t}$$

Lemma I.4 (Monotonicity). *For derivation D of $\Gamma \vdash e : t$ and environment Γ' such that $\Gamma' \leq_P \Gamma$, D can be transformed into a derivation of $\Gamma' \vdash e : t$ just by adding [≤], [Inst] and [∧] nodes.*

Proof. Straightforward induction on the derivation $\Gamma \vdash e : t$, where each [Ax$_\vee$] and [Ax$_\lambda$] node is replaced by a [Inst∧≤] pattern of that node. □

I.1.2 Normalisation Lemmas. Derivations for the declarative type system of Figure 4 can still take many different shapes. In this section, we define several normalisation lemmas, each restricting the use of a non-structural rule. They are then combined into a normalisation theorem.

Normalisation of [∨] nodes

Lemma I.5 (Introduction of an arbitrary [∨] node). *Let Γ a type environment, e and e_x two expressions, and $\{\boldsymbol{u}_i\}_{i\in I}$ a partition of $\mathbb{1}$. Let D be a derivation for the judgement $\Gamma \vdash e\{e_x/x\} : t$ such that D does not contain any [∨] node performing a substitution $\{e_y/y\}$ with e_y a strict sub-expression of e_x. If e_x is typable under the context Γ, then there exists a type s such that D can be transformed into a derivation whose root is a [∨] node of the following form:*

$$[\vee]\ \frac{\dfrac{\cdots}{\Gamma \vdash e_x : s} \qquad \dfrac{\cdots}{\Gamma, x : s \wedge \boldsymbol{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e_x/x\} : t}$$

Proof. Let C a derivation for $\Gamma \vdash e_x : \mathbb{1}$. We collect in D the set $\{C_k\}_{k\in K}$ of all the subderivations for the expression e_x. As substitutions are capture-avoiding, no variable in $\mathrm{fv}(e_x)$ could have been introduced in the environment by a [→I] or [∨] node in D. Thus, we know that the derivations $\{C_k\}_{k\in K}$ are still valid under the initial environment Γ.

Thus, we can build the following derivation:

$$[\vee]\ \frac{[\wedge]\ \dfrac{\dfrac{C}{\Gamma \vdash e_x : \mathbb{1}} \qquad \dfrac{C_k}{\Gamma \vdash e_x : t_k}\ \forall k \in K}{\Gamma \vdash e_x : \bigwedge_{k\in K} t_k} \qquad \dfrac{D'_i}{\Gamma, x : (\bigwedge_{k\in K} t_k) \wedge \mathbf{u}_i \vdash e : t}\ \forall i \in I}{\Gamma \vdash e\{e_x/x\} : t}$$

with each D'_i being a derivation easily derived from D by substituting e_{x} by x when relevant, using a [Ax$_\vee$] rule on x instead of a subderivation for e_{x} when necessary, and by using monotonicity (Lemma I.4). The hypothesis on the derivation D ensures that it does not contain any conflicting [$\vee$] node that would become inapplicable due to the fact that e_{x} has been substituted by x. □

Lemma I.6 (Elimination of aliasing). *Let D be a ground derivation, and N be a [$\vee$] node in D applying a substitution $\{\mathsf{x}/\mathsf{y}\}$. Then, N can be removed from D, without adding any new [$\vee$] node nor structural node in D.*

Proof. The following transformation can be performed to the subderivation introducing x (as D is a ground derivation, there must be a [$\vee$] node that introduces x):

$$[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e_{\mathsf{x}} : s} \quad \dfrac{B}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_k \Vdash e : t} \quad \dfrac{E_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I \setminus \{k\}}{\Gamma \Vdash e\{e_{\mathsf{x}}/\mathsf{x}\} : t}$$

where B is a derivation that contains this subderivation S

(whose root is the node N to eliminate):

$$[\vee]\ \dfrac{\dfrac{C}{\Gamma' \Vdash \mathsf{x} : s'} \quad \dfrac{D_j}{\Gamma', \mathsf{y} : s' \wedge \mathbf{u}'_j \Vdash e' : t'}\ \forall j \in J}{\Gamma' \Vdash e'\{\mathsf{x}/\mathsf{y}\} : t'}$$

with $\Gamma' = (\Gamma, \mathsf{x} : s \wedge \mathbf{u}_k)\dot{\cup}\Gamma''$ for some Γ''

$\downarrow$ (transformed into)

$$[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e_{\mathsf{x}} : s} \quad \dfrac{B'_j}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_k \wedge \mathbf{u}'_j \Vdash e : t}\ \forall j \in J \quad \dfrac{E_i}{\Gamma, \mathsf{x} : s \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I \setminus \{k\}}{\Gamma \Vdash e\{e_{\mathsf{x}}/\mathsf{x}\} : t}$$

where B'_j is constructed from B by monotonicity (Lemma I.4),

and by replacing the subderivation S by this one:

$$\dfrac{D'_j}{(\Gamma, \mathsf{x} : s \wedge \mathbf{u}_k \wedge \mathbf{u}'_j)\dot{\cup}\Gamma'' \Vdash e'\{\mathsf{x}/\mathsf{y}\} : t'}$$

where D'_j is constructed from $D_j\{\mathsf{x}/\mathsf{y}\}$ by monotonicity (Lemma I.4) ($\star$)

($\star$) Note that we have $s \wedge \mathbf{u}_k \leq_{\mathrm{P}} s'$ as the only structural rule that the derivation C can use is [Ax$_\vee$] on x, and thus $s \wedge \mathbf{u}_k \wedge \mathbf{u}'_j \leq s' \wedge \mathbf{u}'_j$. □

An interesting thing to note is that the proof above would not work in the presence of a generalization rule such as the one used at top-level:

$$[\textsc{Gen}]\ \dfrac{\Gamma \Vdash e : t}{\Gamma \Vdash e : t\phi}\ \phi\#\Gamma$$

Indeed, the guard condition $\phi\#\Gamma$ may prevent monotonicity. More precisely, in the transformation above, if the partition $\{\mathbf{u}_i\}_{i\in I}$ introduces a new monomorphic type variable, its introduction earlier in the environment might prevent a potential application of a [Gen] rule in the derivation B. This impossibility to eliminate aliasing would be an issue for the normalisation lemma (Lemma I.13) detailled below, as it states, in particular, that the union elimination rule only needs to be applied once for a given sub-expression. This is the reason why, in our declarative type system, generalisation only occurs at top-level.

Definition I.7 (Acceptable [$\vee$] node). *In any derivation, a [$\vee$] node N doing the substitution $e\{e'/\mathsf{x}\}$ is said acceptable if it satisfies the following constraints:*

- *e contains x (no useless definition), and*
- *e does not contain e' (maximal sharing), and*
- *e' is not a binding variable (no aliasing)*

Definition I.8 (Binding context). *A binding context B is an ordered list of mappings from binding variables to atoms. Each mapping is written as a pair (x, e). We note these lists extensionally by separating elements by a semicolon, that is, $(\mathsf{x}_1, a_1); \ldots; (\mathsf{x}_n, a_n)$ and use ε to denote the empty list.*

We note eB the expression $e\{e_n/\mathsf{x}_n\} \ldots \{e_1/\mathsf{x}_1\}$ where $(\mathsf{x}_1, e_1); \ldots; (\mathsf{x}_n, e_n)$ are the pairs in B.

Given a derivation D and a node N of D, we call *binding context of N in D* the binding context $(\mathsf{x}_1, e_1); \ldots; (\mathsf{x}_n, e_n)$, where $\{e_1/\mathsf{x}_1\}; \ldots; \{e_n/\mathsf{x}_n\}$ are the substitutions made by all the [$\vee$] nodes crossed by one of their body premises when going from the root of D to N.

In the following definition, we use an order over expressions:

Definition I.9 (Expression order). *A (possibly partial) order $\leq$ is called expression order if it is an order over expressions modulo α-renaming, and if it is an extension of the sub-expression order: if e_1 is a sub-expression of e_2 modulo α-renaming, then we should have $e_1 \leq e_2$. We write $e_1 < e_2$ when $e_1 \leq e_2$ and $e_2 \nleq e_1$.*

Definition I.10 (Well-positionned [$\vee$] node). *A [$\vee$] node N of a derivation D, of binding context B in D and doing the substitution $e\{e_1/\mathsf{x}\}$, is said well-positionned in D relatively to the expression order $\leq$ (or more succintly, well-positionned in $(D, \leq)$) if there is no node N' on the path from the root to N such that:*

- *Every variable in $\mathsf{fv}(e_1 B)$ is in the type environment of N', and*
- *N' is not a [$\vee$] rule, or N' is a [$\vee$] rule of binding context B' and performing a substitution $\{e_2/\mathsf{y}\}$ such that $e_1 B < e_2 B'$.*

Definition I.11 ([$\vee$]-canonical derivation). *For a given expression order $\leq$, a derivation D is [$\vee$]-canonical for the order $\leq$ if every [$\vee$] node it contains is acceptable and well-positionned in $(D, \leq)$.*

Definition I.12 (Form derivations, atomic derivations). *A derivation D is a form derivation if every segment of branch containing the root and stopping at the first [$\vee$] node (if any, or stopping at a leaf otherwise):*

- *does not contain any structural node, and*
- *if it ends with a [$\vee$] node N, the definition premise of N is an atomic derivation, and the body premises of N are form derivations.*

A derivation D is an atomic derivation if every segment of branch containing the root and stopping at the first [$\to$I] node (if any, or stopping at a leaf otherwise):

- *does not contain any [$\vee$] node, and*
- *contains exactly one structural node, and*
- *if it ends with a [$\to$I] node N, the premise of N is a form derivation.*

A derivation that is both a [$\vee$]-canonical derivation and a form derivation is called [$\vee$]-canonical form derivation.

Lemma I.13 (Normalisation of [$\vee$]). *Given an expression order $\leq$, any ground derivation D of $\Gamma \Vdash e : t$ can be transformed into a [$\vee$]-canonical form derivation of $\Gamma \Vdash e : t'$ for the order $\leq$ and with $t' \leq_P t$.*

Proof. First, we can remove any aliasing (i.e. [$\vee$] nodes doing a substitution $\{\mathsf{y}/\mathsf{x}\}$) by applying Lemma I.6 as needed. We can also trivially remove useless [$\vee$] nodes (i.e. those doing a substitution $e\{e_\mathsf{x}/\mathsf{x}\}$ where e does not contain x).

Then, let's consider, in the whole derivation, all the nodes that satisfy one of those conditions:

- It is a structural node N such that, when going towards the root, it crosses another structural node before crossing a [$\rightarrow$I] node.
- It is a structural node N such that, when going towards the root, it crosses a [$\vee$] node by one of its body premises before crossing a [$\vee$] node by its definition premise.
- It is a [$\vee$] node that is not acceptable or not well-positionned in $(D, \leq)$.

If there is no such node, then the properties of Lemma I.13 are satisfied. Otherwise, we associate to each of these faulty nodes an expression and a binding context:

- For a structural node applied on an expression e in a binding context B, we associate (e, B),
- For a [$\vee$] node doing the substitution $\{e_\mathsf{x}/\mathsf{x}\}$ in a binding context B, we associate (e_x, B).

Now, we select among those nodes the one whose associated pair is minimal with respects to the following order: (e_1, B_1) is smaller than (e_2, B_2) if and only if $e_1B_1 \leq e_2B_2$. Let's call this node N, and its associated expression and binding context e and B respectively.

Now, let's locate, in the segment from the root to N, the farthest location L from the root such that N would be well-positionned in $(D, \leq)$ at this location.

Let's note $\Gamma_L \Vdash e_L : t_L$ the judgement at this location. We consider the subderivation D_L that derives this judgement in D. Note that, in D_L, there is no [$\vee$] node that makes a substitution $\{e_\mathsf{y}/\mathsf{y}\}$ with e_y a strict sub-expression of e, because $e_\mathsf{y}B_\mathsf{y}$ (with B_y the binding context of the corresponding [$\vee$] node) would be smaller than eB for $\leq$, thus making it not well-positionned and contradicting the minimality of (e, B). Also note that D_L contains the node N (otherwise, the [$\vee$] node N would not be well-positionned at location L).

We apply Lemma I.5 on the root of D_L so that it performs the substitution $e'_L\{e/\mathsf{z}\}$ using the decomposition $\{\mathbf{u}_i\}_{i\in I} = \{\mathbb{1}\}$, with z fresh and e'_L an expression that does not contain e and such that $e'_L\{e/\mathsf{z}\} \equiv e_L\{e/\mathsf{z}\}$ (basically, e'_L is e_L where occurrences of e have been replaced by z). This lemma can be applied as:

- There cannot be in our subderivation any [$\vee$] node substituting a strict sub-expression of e,
- We know that $\Gamma_L \Vdash e : \mathbb{1}$ holds: we can derive it from the definition premise of N if N is a [$\vee$] node, or from N itself if N is a structural node.

This gives us a new derivation D'_L.

If N is a [$\vee$] node, N can be removed in D'_L by using Lemma I.6, as well as other aliasing that would have been introduced in other branches. Not that, if N is a structural node, it has already been eliminated by the application of Lemma I.5. Finally, we replace the subderivation D_L by D'_L in D.

We conclude by repeating this whole process until all the nodes satisfy the conditions. We are guaranteed that it terminates by the fact that eB strictly increases for $\leq$ at each iteration (with (e, B) the pair associated to the choosen node N). □

Normalisation of [Inst] nodes

Lemma I.14. *A derivation $\Gamma \Vdash e : t$ can be transformed into a derivation $\Gamma \Vdash e : t\rho$ (for any renaming ρ) without changing the structure of the derivation.*

Proof. Any polymorphic type variable in t must be introduced either by a [Ax$_\lambda$], [Ax$_\vee$], or [Inst] rule. Thus, we can derive $\Gamma \Vdash e : t\rho$ by induction on $\Gamma \Vdash e : t$, where:

- Every renaming ρ' of a [Ax$_\lambda$] or [Ax$_\vee$] node is replaced by the renaming $\rho \circ \rho'$, and
- Every substitution σ of a [Inst] node is replaced by the substitution $\rho \circ \sigma \circ \rho^{-1}$.

□

Proposition I.15. *If $\forall i \in I.\ t'_i \leq_P t_i$, then $\bigwedge_{i\in I} t'_i \leq_P \bigwedge_{i\in I} t_i$.*

Proof. For each $i \in I$, let Σ_i a set of substitutions such that $t'_i\Sigma_i \leq t_i$. We consider the set of substitutions $\Sigma = \bigcup_{i\in I} \Sigma_i$, and we show that $(\bigwedge_{i\in I} t'_i)\Sigma \leq \bigwedge_{i\in I} t_i$.

We have:

$$\begin{aligned}\textstyle\bigwedge_{\sigma\in\Sigma}(\bigwedge_{i\in I} t'_i)\sigma &\simeq \textstyle\bigwedge_{i\in I}\bigwedge_{\sigma\in\Sigma_i}(\bigwedge_{j\in I} t'_j)\sigma\\ &\leq \textstyle\bigwedge_{i\in I}\bigwedge_{\sigma\in\Sigma_i} t'_i\sigma\\ &\leq \textstyle\bigwedge_{i\in I} t_i\end{aligned}$$

□

Proposition I.16. *If $\forall i \in I.\ t'_i \leq_P t_i$ and all $\{t_i\}_{i\in I}$ have disjoint polymorphic type variables, then $\bigvee_{i\in I} t'_i \leq_P \bigvee_{i\in I} t_i$.*

Proof. For each $i \in I$, let Σ_i a set of substitutions such that $t'_i\Sigma_i \leq t_i$. We consider the set of substitutions $\Sigma = \{\sigma_1 \dot\cup \ldots \dot\cup \sigma_n \mid \sigma_1 \in \Sigma_1, \ldots, \sigma_n \in \Sigma_n\}$ for $I = \{1, \ldots, n\}$, where $\dot\cup$ denotes the composition of disjoint substitutions (their disjointness is guaranteed by the fact that all $\{t'_i\}_{i\in I}$ have disjoint polymorphic type variables), and we show that $(\bigvee_{i\in I} t'_i)\Sigma \leq \bigvee_{i\in I} t_i$.

We have:

$$\begin{aligned}&\textstyle\bigwedge_{\sigma\in\Sigma}(\bigvee_{i\in 1..n} t'_i)\sigma\\ \simeq\ &\textstyle\bigwedge_{(\sigma_1,\ldots,\sigma_n)\in\Sigma_1\times\cdots\times\Sigma_n}(\bigvee_{i\in 1..n} t'_i)(\sigma_1\dot\cup\ldots\dot\cup\sigma_n)\\ \simeq\ &\textstyle\bigwedge_{(\sigma_1,\ldots,\sigma_n)\in\Sigma_1\times\cdots\times\Sigma_n}\bigvee_{i\in 1..n} t'_i(\sigma_1\dot\cup\ldots\dot\cup\sigma_n)\\ \simeq\ &\textstyle\bigwedge_{(\sigma_1,\ldots,\sigma_n)\in\Sigma_1\times\cdots\times\Sigma_n}\bigvee_{i\in 1..n} t'_i\sigma_i\\ \simeq\ &\textstyle\bigvee_{i\in 1..n}\bigwedge_{\sigma_i\in\Sigma_i} t'_i\sigma_i && \text{(distributivity of } \vee \text{ over } \wedge)\\ \leq\ &\textstyle\bigvee_{i\in 1..n} t_i\end{aligned}$$

□

Definition I.17 ([Inst]-canonical derivation). *A derivation D is [Inst]-canonical if every [Inst] node it contains is part of a [Inst$\wedge\leq$] pattern that is either:*

- *The first premise of a [$\mathbb{0}$], [$\in_1$] or [$\in_2$] node, or*
- *The premise of a [$\times E_1$] or [$\times E_2$] node, or*
- *One of the premises of a [$\to$E] node*

Lemma I.18 (Normalisation of [Inst]). *Given an expression order $\leq$, any [$\vee$]-canonical form derivation D of $\Gamma \Vdash e : t$ can be transformed into a [$\vee$][Inst]-canonical form derivation of $\Gamma \Vdash e : t'$ for the order $\leq$ and with $t' \leq_P t$.*

Proof. We proceed by induction on $(n_\vee, n)$ (using the lexicographic order), where $n_\vee$ denotes the number of [$\vee$] nodes in the derivation, and n denotes the total number of nodes in the derivation.

If the root is a [Inst] or [$\leq$], we can remove the root (its premise will be the new root) and proceed inductively on the result.

If the root is a [$\wedge$], we proceed inductively on all its premises and update the intersection type t derived by the root into a new intersection type t' (according to the new premises). We know that the new derived type t' satisfies $t' \leq_{\mathrm{P}} t$ according to Proposition I.15.

If the root is a [$\vee$] of the following form:

$$[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e' : s} \qquad \dfrac{B_i}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}$$

(1) We first proceed inductively on A, which gives a derivation A'. We consider following derivation, where $s' \leq_{\mathrm{P}} s$ (and thus $s' \wedge \mathbf{u}_i \leq_{\mathrm{P}} s \wedge \mathbf{u}_i$):

$$[\vee]\ \dfrac{\dfrac{A'}{\Gamma \Vdash e' : s'} \qquad \dfrac{B'_i}{\Gamma, \mathrm{x} : s' \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}$$

with B'_i a derivation easily derived from B_i by monotonicity (Lemma I.4). Note that the application of the monotonicity lemma might insert unwanted [Inst$\wedge\leq$] patterns after axioms, but they will be eliminated with the next step.

(2) The next step is to proceed inductively on the $\{B'_i\}_{i\in I}$ premises, yielding some derivations $\{B''_i\}_{i\in I}$ that derive some types $\{t_i\}_{i\in I}$ (with $\forall i \in I.\ t_i \leq_{\mathrm{P}} t$). We can suppose that all the $\{t_i\}_{i\in I}$ have disjoint polymorphic type variables: if it is not the case, it can be ensured by applying Lemma I.14 to these premises. Then, we consider the following derivation:

$$[\vee]\ \dfrac{\dfrac{A'}{\Gamma \Vdash e' : s'} \qquad [\leq]\ \dfrac{\dfrac{B''_i}{\Gamma, \mathrm{x} : s' \wedge \mathbf{u}_i \Vdash e : t_i}}{\Gamma, \mathrm{x} : s' \wedge \mathbf{u}_i \Vdash e : \bigvee_{i\in I} t_i}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : \bigvee_{i\in I} t_i}$$

The result $\bigvee_{i\in I} t_i$ satisfies $\bigvee_{i\in I} t_i \leq_{\mathrm{P}} t$ according to Proposition I.16.

(3) The new [$\leq$] nodes that appear as premise of the [$\vee$] root could break the properties of Lemma I.13 if the corresponding B''_i ends with a [$\vee$] node. In this case, we move up the faulty [$\leq$] nodes as needed using this transformation:

$$[\leq]\ \dfrac{[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e' : s} \qquad \dfrac{B_i}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t'}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t'}}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}$$

$$\downarrow$$

$$[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e' : s} \qquad [\leq]\ \dfrac{\dfrac{B_i}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t'}}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}$$

The other cases are straightforward. □

Normalisation of [$\leq$] nodes

Definition I.19 ([$\leq$]-canonical derivation). *A derivation D is [$\leq$]-canonical if every [$\leq$] node it contains is either:*

- *The first premise of a* [$\in_1$] *or* [$\in_2$] *node, or*
- *One of the body premises of a* [$\vee$] *node, or*
- *The premise of a* [$\times E_1$] *or* [$\times E_2$] *node, or*
- *The first premise of a* [$\to$E] *node*

Lemma I.20 (Normalisation of [$\leq$]). *Given an expression order* $\leq$*, any* [$\vee$][Inst]*-canonical form derivation* D *of* $\Gamma \Vdash e : t$ *can be transformed into a* [$\vee$][Inst][$\leq$]*-canonical form derivation of* $\Gamma \Vdash e : t'$ *for the order* $\leq$ *and with* $t' \leq_P t$.

Proof. We proceed by induction on $(n_\vee, n)$ (using the lexicographic order), with $n_\vee$ the number of [$\vee$] nodes in the derivation, and n the total number of nodes in the derivation.

If the root is a [$\leq$] or [Inst], we can remove the root (its premise will be the new root) and proceed by induction on its premise.

If the root is a [$\wedge$], we proceed inductively on all its premises and update the intersection type t derived by the root into a new intersection type t' (according to the new premises). We trivially have $t' \leq_P t$.

If the root is a [$\to$I] pattern, we proceed inductively on its premise and update the arrow type t derived by the root into a new arrow type t' (according to the new premise). We trivially have $t' \leq_P t$.

If the root is a [$\vee$] of the following form:

$$[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e' : s} \qquad \dfrac{B_i}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}$$

(1) We first proceed inductively on A, yielding a derivation A'. We then consider the following derivation, with $s' \leq_P s$:

$$[\vee]\ \dfrac{\dfrac{A'}{\Gamma \Vdash e' : s'} \qquad \dfrac{B_i'}{\Gamma, \mathrm{x} : s' \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}$$

where each B_i' is derived from B_i by applying Lemma I.4 (monotonicity), and then Lemma I.18 in order to normalize [Inst] nodes that might have been introduced by the monotonicity lemma. Note that this might add unwanted [$\leq$] nodes, but they will be eliminated with the next step.

(2) We proceed inductively on the $\{B_i'\}_{i \in I}$ premises, yielding some derivations $\{B_i''\}_{i \in I}$ that derive some types $\{t_i\}_{i \in I}$ (with $\forall i \in I.\ t_i \leq_P t$). Then, we consider the following derivation:

$$[\vee]\ \dfrac{\dfrac{A'}{\Gamma \Vdash e' : s'} \qquad [\leq]\ \dfrac{\dfrac{B_i''}{\Gamma, \mathrm{x} : s' \wedge \mathbf{u}_i \Vdash e : t_i}}{\Gamma, \mathrm{x} : s' \wedge \mathbf{u}_i \Vdash e : \bigvee_{i \in I} t_i}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : \bigvee_{i \in I} t_i}$$

The result $\bigvee_{i \in I} t_i$ trivially satisfies $\bigvee_{i \in I} t_i \leq_P t$.

(3) The new [$\leq$] nodes that appear as premise of the [$\vee$] root could break the properties of Lemma I.13 if the corresponding B_i'' ends with a [$\vee$] node. In this case, we move up the faulty

$[\leq]$ nodes as needed using this transformation:

$$
[\leq]\ \dfrac{[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e' : s} \qquad \dfrac{B_i}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t'}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t'}}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}
$$

$$\downarrow$$

$$
[\vee]\ \dfrac{\dfrac{A}{\Gamma \Vdash e' : s} \qquad [\leq]\ \dfrac{\dfrac{B_i}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t'}}{\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e : t}\ \forall i \in I}{\Gamma \Vdash e\{e'/\mathrm{x}\} : t}
$$

The other cases are straightforward. □

Definition I.21 (Canonical derivation). *Given an expression order $\leq$, a canonical derivation for the expression order $\leq$ is a $[\vee]$[Inst]$[\leq]$-canonical derivation. We say it is a canonical form derivation if it is also a form derivation, and a canonical atomic derivation if it is an atomic derivation.*

When qualifying a derivation D of canonical, the order $\leq$ may be omitted: in this case, we consider that there exists an expression order $\leq$ such that D is canonical for $\leq$.

Theorem I.22 (Normalisation of derivations). *Given an expression order $\leq$, any ground derivation D of $\Gamma \Vdash e : t$ can be transformed into a canonical form derivation of $\Gamma \Vdash e : t'$, for the order $\leq$ and with $t' \leq_{\mathcal{P}} t$.*

Proof. By successive application of Lemma I.13, Lemma I.18, and Lemma I.20. □

I.2 Type Safety

I.2.1 The Parallel Semantics. One technical difficuulty is that the subject reduction does not hold for the semantics presented in Figure 1: performing a reduction step on an expression e might break the use of a $[\vee]$ rule. Indeed, if in the original typing derivation a rule $[\vee]$ substitutes multiple occurrences of the sub-expression e by a variable x, reducing one occurrence of e but not the others can make the application of this $[\vee]$ rule impossible: the correlation between the reduced e and the other occurrences of e is thus 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. With this alternative semantics, the subject reduction becomes true, allowing to prove type safety. The type safety for the initial semantics (Figure 1) is then deduced from this result.

The parallel semantics is formalized in Figure 5. A step of reduction happening at top-level is noted $\leadsto_{\top}$, and a step of reduction of the parallel semantics under any evaluation context is noted $\leadsto_{\mathcal{P}}$. Notice that the rule $[\kappa]$ applies on an expression e'' a substitution from an expression e' to an expression e, noted $e''\{e/e'\}$, which is defined inductively on e'' as follows:

- If $e' \equiv_\alpha e''$, then $e''\{e/e'\} = e$.
- If $e' \not\equiv_\alpha e''$, then $e''\{e/e'\}$ is inductively defined as

Top-level reductions:

$$\begin{array}{rcll}
(\lambda x.e)v & \rightsquigarrow_\top & e\{v/x\} & \quad (7)\\
\pi_1(v_1,v_2) & \rightsquigarrow_\top & v_1 & \quad (8)\\
\pi_2(v_1,v_2) & \rightsquigarrow_\top & v_2 & \quad (9)\\
(v\in\tau)\ ?\ e_1 : e_2 & \rightsquigarrow_\top & e_1 \qquad \text{if } v \in \tau & \quad (10)\\
(v\in\tau)\ ?\ e_1 : e_2 & \rightsquigarrow_\top & e_2 \qquad \text{if } v \in \neg\tau & \quad (11)
\end{array}$$

Parallel reductions:

$$[\kappa]\ \frac{e \rightsquigarrow_\top e'}{E[e] \rightsquigarrow_{\mathcal{P}} (E[e])\{e'/e\}}$$

Evaluation Context $\quad E ::= [\,] \mid vE \mid Ee \mid (v,E) \mid (E,e) \mid (E\in\tau)\ ?\ e : e$

Fig. 5. Parallel Semantics

$$\begin{array}{rll}
c\{e/e'\} &= c & \\
x\{e/e'\} &= x & \\
\mathsf{x}\{e/e'\} &= \mathsf{x} & \\
(e_1e_2)\{e/e'\} &= (e_1\{e/e'\})(e_2\{e/e'\}) & \\
(\lambda x.e_\circ)\{e/e'\} &= \lambda x.e_\circ & x \in \mathsf{fv}(e')\\
(\lambda x.e_\circ)\{e/e'\} &= \lambda x.(e_\circ\{e/e'\}) & x \notin \mathsf{fv}(e) \cup \mathsf{fv}(e')\\
(\lambda x.e_\circ)\{e/e'\} &= \lambda y.((e_\circ\{y/x\})\{e/e'\}) & x \notin \mathsf{fv}(e), x \in \mathsf{fv}(e'), y \text{ fresh}\\
(\pi_i e_\circ)\{e/e'\} &= \pi_i(e_\circ\{e/e'\}) & \\
(e_1,e_2)\{e/e'\} &= (e_1\{e/e'\}, e_2\{e/e'\}) & \\
((e_1\in t)\ ?\ e_2 : e_3)\{e/e'\} &= (e_1\{e/e'\}\in t)\ ?\ e_2\{e/e'\} : e_3\{e/e'\} &
\end{array}$$

In particular, notice that expression substitutions are up to α-renaming and perform only one pass.

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

$$[\kappa]\ \frac{(\lambda x.x)(\lambda x.x) \rightsquigarrow_\top \lambda x.x}{((\lambda x.x)(\lambda x.x), (\lambda x.x)(\lambda x.x)) \rightsquigarrow_{\mathcal{P}} (\lambda x.x, \lambda x.x)}$$

I.2.2 Subject Reduction.

Proposition I.23. *If $\Gamma \Vdash v : \tau$, then $v \in \tau$ (see Figure 1 for the definition of $\in$).*

Proof. Straightforward, by induction on the derivation of the judgement $\Gamma \Vdash v : \tau$. Note that the case of λ-abstractions is trivial as arrows in τ can only be $\mathbb{0} \to \mathbb{1}$. □

Lemma I.24 (Substitution lemma). *If $\Gamma, x : s \Vdash e : t$ and $\Gamma \Vdash e' : s$, then $\Gamma \Vdash e\{e'/x\} : t$.*

Proof. Straightforward induction on the derivation $\Gamma, x : s \Vdash e : t$, where each $[\text{Ax}_\lambda]$ node is replaced by the derivation $\Gamma \Vdash e' : s$. □

Definition I.25 (Atomic type). *A type t is said atomic if it cannot be decomposed into a non-trivial union. Formally, t is atomic if and only if, for any set of types $\{t_i\}_{i\in I}$, $t \leq \bigvee_{i\in I} t_i \Rightarrow \exists i \in I.\ t \leq t_i$. Equivalently, t is atomic if and only if, for any type s, either $t \leq s$ or $t \leq \neg s$.*

Lemma I.26 (Atomicity of value types). *If there exists a canonical atomic derivation D of $\Gamma \Vdash v : s$ (with v a value), then s is atomic.*

Proof. As v is a value, we know that the derivation does not contain any destructor nor axiom node, except possibly in the premise of a [$\rightarrow$I] node. Moreover, as D is a canonical atomic derivation, it does not contain any [$\vee$] nor [$\leq$] node neither. In particular, this implies that s cannot contain any type variable nor union, except under an arrow. More precisely, we can easily prove that s can be constructed with the following syntax:

$$\textbf{Value Type} \quad \bar{t} \quad ::= \quad b \mid t \rightarrow t \mid \bar{t} \times \bar{t} \mid \bar{t} \wedge \bar{t}$$

Any type $t_1 \rightarrow t_2$ is atomic, so is any base type b. Atomicity is preserved by intersection and product, thus we can deduce that a type constructed with the syntax above is atomic. □

Definition I.27 (Unavoidable [$\vee$] node). *In any derivation, a [$\vee$] node N doing the substitution $e\{e'/\mathrm{x}\}$ is said unavoidable if N is acceptable (Definition I.7) and if e' is not a value. A [$\vee$] node that is not unavoidable is said avoidable.*

Definition I.28 (Minimal derivation). *A derivation is said minimal if every [$\vee$] node it contains is unavoidable.*

Lemma I.29 (Elimination of value substitutions). *Any canonical form derivation or canonical atomic derivation D of $\Gamma \Vdash e : t$ can be transformed into a minimal derivation of $\Gamma \Vdash e : t$.*

Proof. We proceed by structural induction on D.

If the root is not a [$\vee$] node, or if it is an unavoidable [$\vee$] node, then we proceed inductively on all its premises.

Otherwise, if the root is an avoidable [$\vee$] node, we know that it is doing a substitution $e\{v/\mathrm{x}\}$ with v a value (as D is canonical, the [$\vee$] node must be acceptable). We can thus apply Lemma I.26 on its definition premise $\Gamma \Vdash v : s$: we deduce that s is atomic. Consequently, among the types $\{\mathbf{u}_i\}_{i\in I}$ composing the partition of $\mathbb{1}$ used by this node, there is a type $\mathbf{u}_i$, for some $i \in I$, such that $s \leq \mathbf{u}_i$.

We consider the corresponding body premise $\Gamma, \mathrm{x} : s \Vdash e : t$, and we proceed by induction on it in order to get a minimal derivation D' for $\Gamma, \mathrm{x} : s \Vdash e : t$. Similarly, we proceed by induction on the definition premise $\Gamma \Vdash v : s$ in order to get a minimal derivation A for $\Gamma \Vdash v : s$.

We can then derive $\Gamma \Vdash e\{v/\mathrm{x}\} : t$ from D' by replacing [$\mathrm{Ax}_\vee$] nodes applying on x by the derivation A (in a similar way as done in Lemma I.24, but for a binding variable). This new derivation is minimal. □

Lemma I.30 (Subject reduction). *If D is a minimal derivation for $\Gamma \Vdash e : t$, and if $e_\circ \rightsquigarrow_\top e'_\circ$, then $\Gamma \Vdash e\{e'_\circ/e_\circ\} : t$.*

Proof. We proceed by structural induction on D.

If e contains no occurrence of $e_\circ$ (modulo α-renaming), this result is trivial. Thus, we will suppose in the following that e contains at least one occurrence of $e_\circ$.

We denote by e' the expression $e\{e'_\circ/e_\circ\}$, and consider the root of the derivation:

[Const] Impossible case (e cannot contain any reducible expression).

[Ax$_\lambda$] Impossible case (e cannot contain any reducible expression).

[Ax$_\vee$] 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 a [$\leq$] node.

[Inst] Similar to the previous case.

[$\wedge$] We proceed inductively on each premise, and intersect the resulting derivations with a [$\wedge$] node.

[→I] We have $e' \equiv \lambda x.\ (e_\lambda\{e'_\circ/e_\circ\})$ for some expression e_λ. We can derive $\Gamma, x : \mathbf{u} \Vdash e_\lambda\{e'_\circ/e_\circ\} : t'$ by induction on the premise $\Gamma, x : \mathbf{u} \Vdash e_\lambda : t'$ (with $t \simeq \mathbf{u} \to t'$) and conclude by using a [→I] node.

[×I] We have $e' \equiv (e_1\{e'_\circ/e_\circ\}, e_2\{e'_\circ/e_\circ\})$ for some e_1 and e_2. We proceed inductively on the premises, as in the previous case.

[→E] We have $e \equiv e_1 e_2$ for some expressions e_1 and e_2. If $e_\circ$ is a sub-expression of e_1 and/or e_2, we conclude by proceeding inductively on the premises, as in the previous case.

Otherwise, $e_\circ \equiv e_1 e_2$ and thus the reduction $e_\circ \rightsquigarrow_\top e'_\circ$ uses the rule 7. Consequently, we know that $e_\circ \equiv e \equiv (\lambda x.\ e_\lambda)v$ for some expression e_λ and value v, and $e'_\circ \equiv e' \equiv e_\lambda\{v/x\}$.

We have the following premises:

(1) $\Gamma \Vdash \lambda x.\ e_\lambda : t_1 \to t_2$ (with $t_2 \simeq t$)

(2) $\Gamma \Vdash v : t_1$

As D is minimal, we know that the premise (1) has no [∨] node, except possibly in the premise of a [→I] node. Thus, we can extract from (1) a collection of derivations of the judgements $\Gamma, x : \mathbf{u}_i \Vdash e_\lambda : s_i$ for $i \in I$, such that there exists some instantiations Σ_i such that $\bigwedge_{i\in I}(\mathbf{u}_i \to s_i)\Sigma_i \leq t_1 \to t_2$. We have $\bigwedge_{i\in I} \mathbf{u}_i \to s_i\Sigma_i \simeq \bigwedge_{i\in I}(\mathbf{u}_i \to s_i)\Sigma_i \leq t_1 \to t_2$. Moreover, using [Inst∧≤] patterns, we can derive for each $i \in I$ the judgement $\Gamma, x : \mathbf{u}_i \Vdash e_\lambda : s_i\Sigma_i$.

Now, let's consider a partition $\{\mathbf{v}_j\}_{j\in J}$ of $\bigvee_{i\in I} \mathbf{u}_i$ of minimal cardinality such that: $\forall j \in J.\ \forall i \in I.\ \mathbf{v}_j \leq \mathbf{u}_i$ or $\mathbf{v}_j \wedge \mathbf{u}_i \simeq \mathbb{0}$. We know that J is not empty: the case $t_1 \leq \bigvee_{i\in I} \mathbf{u}_i \simeq \mathbb{0}$ is impossible since the value v would have type $\mathbb{0}$, contradicting Proposition I.23. For every $j \in J$, we define $I_j = \{i \in I \mid \mathbf{v}_i \wedge \mathbf{u}_i \not\simeq \mathbb{0}\}$ (I_j cannot be empty as the partition $\{\mathbf{v}_j\}_{j\in J}$ has minimal cardinality). Note that for any $j \in J$ and $i \in I_j$, we have $\mathbf{v}_j \leq \mathbf{u}_i$.

Using the monotonicity lemma (Lemma I.4), we can derive for every $j \in J$ and $i \in I_j$ the judgement $\Gamma, x : \mathbf{v}_j \Vdash e_\lambda : s_i\Sigma_i$. Thus, using a [∧] node, for every $j \in J$ we can derive $\Gamma, x : \mathbf{v}_j \Vdash e_\lambda : \bigwedge_{i\in I_j} s_i\Sigma_i$. Moreover, as $\bigwedge_{i\in I} \mathbf{u}_i \to s_i\Sigma_i \leq t_1 \to t_2$, we have, for every $j \in J$, $\bigwedge_{i\in I_j} s_i\Sigma_i \leq t_2$. Consequently, for every $j \in J$, we can derive the judgement $\Gamma, x : \mathbf{v}_j \Vdash e_\lambda : t_2$ using a [≤] node. Combining these judgements with a [∨] node allows us to derive $\Gamma, x : \bigvee_{j\in J} \mathbf{v}_j \Vdash e_\lambda : t_2$.

As $\bigvee_{j\in J} \mathbf{v}_j$ covers t_1, we can construct from (2) a derivation $\Gamma \Vdash v : \bigvee_{j\in J} \mathbf{v}_j$ using a [≤] node. Finally, applying the substitution lemma (Lemma I.24) gives a derivation for $\Gamma \Vdash e_\lambda\{v/x\} : t_2$.

[×E$_1$] We have $e \equiv \pi_1 e_1$ for some expression e_1. If $e_\circ$ is a sub-expression of e_1, we conclude by proceeding inductively on the premise.

Otherwise, $e_\circ \equiv \pi_1 e_1$ and thus the reduction $e_\circ \rightsquigarrow_\top e'_\circ$ uses the rule 8. Consequently, we know that $e_\circ \equiv e \equiv \pi_1(v_1, v_2)$ for some values v_1 and v_2, and $e'_\circ \equiv e' \equiv v_1$.

Similarly to the previous case, as D is minimal, we can extract from the premise $\Gamma \Vdash (v_1, v_2) : t_1 \times t_2$ a collection of derivations of the judgements $\Gamma \Vdash v_1 : s_i$ and $\Gamma \Vdash v_2 : s'_i$ for $i \in I$, such that there exists some instantiations Σ_i such that $\bigwedge_{i\in I}(s_i \times s'_i)\Sigma_i \leq t_1 \times t_2$. We thus have $\bigwedge_{i\in I} s_i\Sigma_i \leq t_1$, as none of the $\{s'_i\}_{i\in I}$ can be $\mathbb{0}$ (this would contradict Proposition I.23 as v_2 is a value).

Therefore, we can conclude this case by using a [Inst∧≤] pattern with the premises $\{\Gamma \Vdash v_1 : s_i\}_{i\in I}$ in order to derive $\Gamma \Vdash v_1 : t_1$.

[×E$_2$] Similar to the previous case.

[∨] We have $e \equiv e_1\{e_2/\mathrm{x}\}$ for some expressions e_1 and e_2, and thus $e' \equiv (e_1\{e_2/\mathrm{x}\})\{e'_\circ/e_\circ\}$.

We have the following premises:

(1) Definition premise: $\Gamma \Vdash e_2 : s$

(2) Body premises: $\forall i \in I.\ \Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e_1 : t$

As D is minimal, we know that e_2 cannot be a value. Also, we know that $e_\circ$ does not contain x as a free variable (otherwise there would be no occurrence of $e_\circ$ in $e_1\{e_2/\mathrm{x}\}$). Consequently, $e'_\circ$ does not contain x neither, because a reduction step cannot introduce a new free variable. There are several cases:

- $e_\circ$ does not contain e_2 and e_2 does not contain $e_\circ$. In this case, we have: $e' \equiv (e_1\{e_2/\mathrm{x}\})\{e'_\circ/e_\circ\} \equiv (e_1\{e'_\circ/e_\circ\})\{e_2/\mathrm{x}\}$. Thus, we can easily conclude by keeping the definition premise of the $[\vee]$ node and applying the induction hypothesis on the body premises.
- e_2 contains $e_\circ$. In this case, we pose $e'_2 = e_2\{e'_\circ/e_\circ\}$.
  We have $e' \equiv (e_1\{e_2/\mathrm{x}\})\{e'_\circ/e_\circ\} \equiv (e_1\{e'_\circ/e_\circ\})\{e'_2/\mathrm{x}\}$. We can derive $\Gamma \Vdash e'_2 : s$ by induction on the definition premise, and $\Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e_1\{e'_\circ/e_\circ\} : t$ for all $i \in I$ by induction on the body premises. Thus, we can derive $\Gamma \Vdash (e_1\{e'_\circ/e_\circ\})\{e'_2/\mathrm{x}\} : t$ using a $[\vee]$ node.
- $e_\circ$ contains e_2 as a strict sub-expression. In this case, we pose $e_\bullet = e_\circ\{\mathrm{x}/e_2\}$ and $e'_\bullet = e'_\circ\{\mathrm{x}/e_2\}$. We know that e_1 does not contain any occurrence of e_2 (because D is minimal), and thus no occurrence of $e_\circ$ neither. Consequently, we have $e' \equiv (e_1\{e_2/\mathrm{x}\})\{e'_\circ/e_\circ\} \equiv (e_1\{e'_\bullet/e_\bullet\})\{e_2/\mathrm{x}\}$.
  As e_2 is not a value, it can only appear in $e_\circ$ inside of a λ-abstraction, and/or inside of a branch of a typecase: otherwise, $e_\circ$ would not be reducible.
  Thus, we can deduce that $e_\bullet = e_\circ\{\mathrm{x}/e_2\} \rightsquigarrow_\top e'_\circ\{\mathrm{x}/e_2\} = e'_\bullet$. Consequently, we can easily conclude by keeping the definition premise of the $[\vee]$ node and applying the induction hypothesis on the body premises.

[$\mathbb{0}$] We have $e \equiv (e_1 \in \tau)\ ?\ e_2 : e_3$ for some e_1, e_2 and e_3. As values cannot have the type $\mathbb{0}$ (Proposition I.23), we know that e_1 is not a value. Thus, $e' \equiv (e_1\{e'_\circ/e_\circ\} \in \tau)\ ?\ e_2\{e'_\circ/e_\circ\} : e_3\{e'_\circ/e_\circ\}$. We can derive $\Gamma \Vdash e_1\{e'_\circ/e_\circ\} : \mathbb{0}$ by proceeding inductively on the premise, and then we can derive $\Gamma \Vdash e' : \mathbb{0}$ by using [$\mathbb{0}$].

[$\in_1$] We have $e \equiv (e_1 \in \tau)\ ?\ e_2 : e_3$ for some e_1, e_2 and e_3. There are three cases:

$e' \equiv (e_1\{e'_\circ/e_\circ\} \in \tau)\ ?\ e_2\{e'_\circ/e_\circ\} : e_3\{e'_\circ/e_\circ\}$ We can easily conclude by proceeding inductively on the premises.

$e' \equiv e_2$ The second premise, unchanged, allows us to conclude.

$e' \equiv e_3$ 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 Proposition I.23 that $e_1 \in \tau$, which contradicts $e \rightsquigarrow_\top e_3$.

[$\in_2$] Similar to the previous case.

□

Theorem I.31 (Subject reduction). *If $\Gamma \Vdash e : t$ with e a ground expression and if $e \rightsquigarrow_{\mathcal{P}} e'$, then $\Gamma \Vdash e' : t$.*

Proof. Using the normalisation theorem (Theorem I.22) and the Lemma I.29, we can build a minimal derivation for $\Gamma \Vdash e : t$.

Moreover, the root of the reduction step $e \rightsquigarrow_{\mathcal{P}} e'$ is a $[\kappa]$ node, with its premise being of the form $e_\circ \rightsquigarrow_\top e'_\circ$, and with $e' \equiv e\{e'_\circ/e_\circ\}$.

Thus, by using Lemma I.30, we obtain $\Gamma \Vdash e' : t$. □

I.2.3 Progress.

Lemma I.32 (Progress). *If D is a minimal derivation for $\Gamma \Vdash e : t$, and if there is no evaluation context E and variable ${}^{x}\!/\!_{\mathrm{x}}$ such that $e \equiv E[{}^{x}\!/\!_{\mathrm{x}}]$, then either e is a value or $\exists e'.\ e \rightsquigarrow_{\mathcal{P}} e'$.*

Proof. We proceed by structural induction on D.

We consider the root of the derivation:

[Const] Trivial (e is a value).
[Ax$_\lambda$] Impossible case (contradict the hypotheses).
[Ax$_\vee$] Impossible case (contradict the hypotheses).
[$\leq$] Trivial (by induction on the premise).
[Inst] Trivial (by induction on the premise).
[$\wedge$] Trivial (by induction on one of the premises).
[$\to$I] Trivial (e is a value).
[$\times$I] We have $e \equiv (e_1, e_2)$ for some expressions e_1 and e_2.

- If e_1 is not a value, and as we have $\forall E, {}^{x}\!/\!_{\mathrm{x}}.\ e_1 \not\equiv E[{}^{x}\!/\!_{\mathrm{x}}]$, we know by applying the induction hypothesis that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $([\,], e_2)$.
- If e_1 is a value, then we can apply the induction hypothesis on the second premise (as e_1 is a value, we know that $\forall E, {}^{x}\!/\!_{\mathrm{x}}.\ e_2 \not\equiv E[{}^{x}\!/\!_{\mathrm{x}}]$). 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 under the evaluation context $(e_1, [\,])$.

[$\to$E] We have $e \equiv e_1 e_2$ for some expressions e_1 and 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 $\forall E, {}^{x}\!/\!_{\mathrm{x}}.\ e_1 \not\equiv E[{}^{x}\!/\!_{\mathrm{x}}]$, we know by applying the induction hypothesis that e_1 can be reduced. Thus, e can also be reduced under the evaluation context $[\,]e_2$.
- If e_1 is a value, we can apply Proposition I.23 on it: as $\Gamma \Vdash e_1 : \mathbb{0} \to \mathbb{1}$, it implies that $e_1 \in \mathbb{0} \to \mathbb{1}$ and thus $e_1 \equiv \lambda x.\ e_\lambda$ for some e_λ. Moreover, we can apply the induction hypothesis on the second premise (as e_1 is a value, we know that $\forall E, {}^{x}\!/\!_{\mathrm{x}}.\ e_2 \not\equiv E[{}^{x}\!/\!_{\mathrm{x}}]$). 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 7, otherwise, e can be reduced under the evaluation context $e_1[\,]$.

[$\times$E$_1$] We have $e \equiv \pi_1 e_1$ for some e_1, with $\Gamma \Vdash e_1 : t \times s$. By applying the induction hypothesis on the premise, we know that e_1 is either a value or it can be reduced. If e_1 can be reduced, then e can also be reduced under the evaluation context $\pi_1[\,]$. Otherwise, as $\Gamma \Vdash e_1 : \mathbb{1} \times \mathbb{1}$, we can apply Proposition I.23 on it, yielding $e_1 \in \mathbb{1} \times \mathbb{1}$. Thus, $e_1 \equiv (v_1, v_2)$ for some values v_1 and v_2, and consequently e can be reduced using the rule 8.
[$\times$E$_2$] Similar to the previous case.
[$\vee$] We have $e \equiv e_1\{e_2/\mathrm{x}\}$ for some e_1 and e_2, with $\Gamma \Vdash e_2 : s$ and $\forall i \in I.\ \Gamma, \mathrm{x} : s \wedge \mathbf{u}_i \Vdash e_1 : t$. There are two cases:

- There exists an evaluation context E such that $e_1 \equiv E[\mathrm{x}]$. In this case, we know that $\forall E', {}^{y}\!/\!_{\mathrm{y}}.\ e_2 \not\equiv E'[{}^{y}\!/\!_{\mathrm{y}}]$, otherwise we would have $e \equiv E[E'[{}^{y}\!/\!_{\mathrm{y}}]]$. Thus, we can apply the induction hypothesis on the definition premise. It gives that either e_2 is a value or it can be reduced. As D is minimal, e_2 cannot be a value and thus e_2 can be reduced. Consequently, e can also be reduced under the evaluation context E.
- There is no evaluation context E such that $e_1 \equiv E[\mathrm{x}]$. In this case, we apply the induction hypothesis on one of the body premises. 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, otherwise, e can be reduced.

[$\mathbb{0}$] We have $e \equiv (e_1{\in}\tau)\,?\,e_2 : e_3$ for some e_1, e_2 and e_3, with $\Gamma \Vdash e_1 : \mathbb{0}$. As values cannot have the type $\mathbb{0}$ (Proposition I.23), we know that e_1 is not a value. Thus, by applying the induction hypothesis on the premise, we know that e_1 can be reduced. Consequently, e can be reduced under the evaluation context $([\,]{\in}\tau)\,?\,e_2 : e_3$.
[$\in_1$] We have $e \equiv (e_1{\in}\tau)\,?\,e_2 : e_3$ for some e_1, e_2 and e_3, with $\Gamma \Vdash e_1 : \tau$. We thus have $e_1 \in \tau$ (Proposition I.23). By applying the induction hypothesis on the first premise, we know that

e_1 is either a value or it can be reduced. If e_1 is a value, then e can be reduced using the rule 10. Otherwise, e_1 can be reduced and thus e can also be reduced under the evaluation context $([\,]{\in}\tau)\ ?\ e_2 : e_3$.

[$\in_2$] Similar to the previous case.

□

Theorem I.33 (Progress). *If $\varnothing \Vdash e : t$ with e a ground expression, then either e is a value or $\exists e'.\ e \leadsto_{\mathcal{P}} e'$.*

Proof. Using the normalisation theorem (Theorem I.22) and the Lemma I.29, we can build a minimal derivation for $\varnothing \Vdash e : t$.

Moreover, we can deduce from $\varnothing \Vdash e : t$ that there is no evaluation context E and lambda variable $^x\!/_{\mathrm{x}}$ such that $e \equiv E[^x\!/_{\mathrm{x}}]$. Thus, we can conclude using Lemma I.32. □

I.2.4 Type Safety for Programs. Now that we expressed subject reduction and progress for the parallel semantics on expressions, we extend it to programs, still using the parallel semantics. Then, we will deduce from it a type safety theorem on programs for the parallel semantics.

Lemma I.34 (Weak monotonicity). *For derivation D of $\Gamma \Vdash e : t$ and environment Γ' such that $\Gamma' \leq \Gamma$, D can be transformed into a derivation of $\Gamma' \Vdash e : t$ just by adding [$\leq$] nodes.*

Proof. Straightforward induction on the derivation $\Gamma \Vdash e : t$, where each [Ax$_\vee$] and [Ax$_\lambda$] node is replaced by a [$\leq$] node with the [Ax$_\vee$] or [Ax$_\lambda$] node as premise. □

Lemma I.35 (Elimination of [Inst] nodes). *For any canonical form derivation D of $\Gamma \Vdash e : t$ and set of substitutions Σ, D can be transformed into a derivation D' of $\Gamma\Sigma' \Vdash e : t\Sigma$, for some Σ', and such that D' does not contain any [Inst] node.*

Proof. We procced by structural induction on the derivation D (with D being a canonical form derivation).

If the root of D is a [Ax$_\vee$] node, we can directly conclude by choosing $\Sigma' = \Sigma$.

Otherwise, if the root D is a [$\vee$] node, doing a substitution $e\{e'/\mathrm{x}\}$, we first proceed inductively on its body premises (which all are canonical form derivations). It yields some derivations $\Gamma\Sigma'_i, \mathrm{x} : s\Sigma'_i \wedge \mathbf{u}_i \Vdash e : t\Sigma$ for $i \in I$. We consider the set of substitutions $\Sigma' = \bigcup_{i\in I} \Sigma'_i$.

Now, let's try to derive $\Gamma\Sigma'' \Vdash e' : s'$, for some Σ'' and s', with $s' \leq s\Sigma'$ ($\leq s\Sigma'_i$ for every $i \in I$). For that, we consider the definition premise of D, deriving $\Gamma \Vdash e' : s$, and apply the following process on it:

- If its root is a [$\wedge$] node, we apply inductively this process to each premise $\Gamma \Vdash e' : s_j$ (with $\bigvee_{j\in J} s_j \simeq s$) in order to derive some $\Gamma\Sigma''_j \Vdash e' : s'_j$, with $s'_j \leq s\Sigma'$. We then consider $\Sigma'' = \bigcup_{j\in J} \Sigma''_j$, and derive for each $j \in J$ $\Gamma\Sigma'' \Vdash e' : s'_j$ using Lemma I.34. Finally, we intersect the those derivations with a [$\wedge$] node.

- If its root is a [→E] node, we apply the following transformation:

$$\dfrac{[\textsc{Inst}\wedge\leq]\ \dfrac{[\mathrm{Ax}_\vee]\ \overline{\Gamma \Vdash \mathtt{x}_1 : \Gamma(\mathtt{x}_1)}}{\Gamma \Vdash \mathtt{x}_1 : t \to s} \qquad [\textsc{Inst}\wedge\leq]\ \dfrac{[\mathrm{Ax}_\vee]\ \overline{\Gamma \Vdash \mathtt{x}_2 : \Gamma(\mathtt{x}_2)}}{\Gamma \Vdash \mathtt{x}_2 : t}}{\Gamma \Vdash \mathtt{x}_1\mathtt{x}_2 : s}\ [\to\mathrm{E}]$$

$$\text{with } t \to s \leq \Gamma(\mathtt{x}_1)\Sigma''_1 \text{ and } t \leq \Gamma(\mathtt{x}_2)\Sigma''_2$$

$$\downarrow$$

$$\dfrac{[\leq]\ \dfrac{[\mathrm{Ax}_\vee]\ \overline{\Gamma\Sigma'' \Vdash \mathtt{x}_1 : \Gamma(\mathtt{x}_1)\Sigma''}}{\Gamma\Sigma'' \Vdash \mathtt{x}_1 : t\Sigma' \to s\Sigma'} \qquad [\leq]\ \dfrac{[\mathrm{Ax}_\vee]\ \overline{\Gamma\Sigma'' \Vdash \mathtt{x}_2 : \Gamma(\mathtt{x}_2)\Sigma''}}{\Gamma\Sigma'' \Vdash \mathtt{x}_2 : t\Sigma'}}{\Gamma\Sigma'' \Vdash \mathtt{x}_1\mathtt{x}_2 : s\Sigma'}\ [\to\mathrm{E}]$$

$$\text{with } \Sigma'' = \{\sigma' \circ \sigma'' \mid \sigma' \in \Sigma', \sigma'' \in \Sigma''_1 \cup \Sigma''_2\}$$

- If its root is a [→I] node, we apply the following transformation:

$$[\to\mathrm{I}]\ \dfrac{\dfrac{A}{\Gamma, x : \mathbf{u} \Vdash e_\circ : t}}{\Gamma \Vdash \lambda x.e_\circ : \mathbf{u} \to t} \quad \leftrightarrow \quad [\to\mathrm{I}]\ \dfrac{\dfrac{A'}{\Gamma\Sigma'', x : \mathbf{u} \Vdash e_\circ : (t\Sigma')}}{\Gamma\Sigma'' \Vdash \lambda x.e_\circ : \mathbf{u} \to (t\Sigma')}$$

with A' obtained by applying the induction hypothesis on A and Σ'.

- If its root is a [Ax$_\lambda$] node, we apply the following transformation:

$$[\mathrm{Ax}_\lambda]\ \overline{\Gamma \Vdash x : \Gamma(x)} \quad \leftrightarrow \quad [\mathrm{Ax}_\lambda]\ \overline{\Gamma\Sigma' \Vdash x : \Gamma(x)\Sigma'}$$

- The other cases are similar.

We thus have Σ'' and s' such that $\Gamma\Sigma'' \Vdash e' : s'$ and $\forall i \in I.\ s' \leq s\Sigma'_i$. We transform, for each $i \in I$, the derivation $\Gamma\Sigma'_i, \mathtt{x} : s\Sigma'_i \wedge \mathbf{u}_i \Vdash e : t\Sigma$ into a derivation $\Gamma\Sigma'', \mathtt{x} : s' \wedge \mathbf{u}_i \Vdash e : t\Sigma$ using Lemma I.34. By combining all those derivations into a [∨] node, we can derive $\Gamma\Sigma'' \Vdash e\{e'/\mathtt{x}\} : t\Sigma$. □

Lemma I.36. *If $\Gamma \Vdash e : t$, then for any substitution ψ, we can derive $\Gamma\psi \Vdash e : t\psi$.*

Proof. Straightforward induction on the derivation of $\Gamma \Vdash e : t$. The type substitution ψ can be applied to every type (and type environment) in the derivation, and as $\mathrm{dom}(\psi) \subseteq \mathcal{V}_M$ it will not conflict with any [Inst] node in the derivation. Subtyping relations are preserved as $t_1 \leq t_2 \Rightarrow t_1\psi \leq t_2\psi$. □

Lemma I.37 (Generalisation lemma). *If $\Gamma, x : s\phi \Vdash e : t$ with e a ground expression, $\phi\#\Gamma$, and $\Gamma \Vdash e' : s$, then $\Gamma \Vdash e\{e'/x\} : t'$ for some t' such that there exists a substitution ϕ' such that $t'\phi' \simeq t$ and $\phi'\#\Gamma$.*

Proof. We can suppose, without loss of generality, that ϕ is a full generalisation, that is, an injective substitution mapping every type variable in $(\mathrm{vars}(x) \wedge \mathcal{V}_M) \setminus \mathrm{vars}(\Gamma)$ to a fresh polymorphic type variable (and being the identity for any other type variable). Indeed, if for any ϕ the conditions of Lemma I.37 are satisfied, then they are also satisfied for any such generalisation according to the monotonicity lemma (Lemma I.4).

Now, we transform the derivation of $\Gamma, x : s\phi \Vdash e : t$ into a canonical form derivation D of $\Gamma, x : s\phi \Vdash e : t'$ (with $t' \leq_{\mathrm{P}} t$) using Theorem I.22. Let Σ such that $t'\Sigma \leq t$. Then, using Lemma I.35, we transform D into a derivation D' of $\Gamma\Sigma', x : (s\phi)\Sigma' \Vdash e : t'\Sigma$, for some Σ', and such that D' does not contain any [Inst] node. Adding a [≤] at the root gives $\Gamma\Sigma', x : (s\phi)\Sigma' \Vdash e : t$.

Let σ'' be an injective substitution mapping every polymorphic type variable appearing in the image of at least one of the substitutions $\sigma' \in \Sigma'$ and ϕ to a fresh monomorphic type variable (and being the identity for any other type variable). From $\Gamma\Sigma', x : (s\phi)\Sigma' \Vdash e : t$, we can derive a derivation of $(\Gamma\Sigma')\sigma'', x : ((s\phi)\Sigma')\sigma'' \Vdash e : t\sigma''$ simply by renaming the type variables everywhere according to σ''. This is only possible because the derivation does not contain any [Inst] node (substituting a polymorphic type variable by a monomorphic one could invalidate [Inst] nodes). By monotonicity (Lemma I.4), we get $\Gamma, x : ((s\phi)\Sigma')\sigma'' \Vdash e : t\sigma''$.

Now, we consider the set of substitutions $\{\phi_i\}_{i\in I} \overset{\text{def}}{=} \{\sigma'' \circ \sigma' \circ \phi \mid \sigma' \in \Sigma'\}$. For any $i \in I$, we can decompose ϕ_i into two substitutions, $\psi_i \overset{\text{def}}{=} \sigma_i\big|_{\mathcal{V}_M}$ and $\sigma_i \overset{\text{def}}{=} \sigma_i\big|_{\mathcal{V}_P}$. Note that $\forall i \in I.\ \psi_i \# \Gamma$ (it follows from $\phi \# \Gamma$). Rewriting the previous judgement with these new notations gives $\Gamma, x : \bigwedge_{i\in I}(s\psi_i)\sigma_i \Vdash e : t\sigma''$.

Using Lemma I.36 on the derivation of $\Gamma \Vdash e' : s$, we can derive, for any $i \in I$, $\Gamma \Vdash e' : s\psi_i$ (we recall that $\psi_i \# \Gamma$, and thus $\Gamma\psi_i \simeq \Gamma$). Using a [Inst] node, we can then derive $\Gamma \Vdash e' : (s\psi_i)\sigma_i$ for any $i \in I$. Regrouping those derivations with a [$\wedge$] node gives $\Gamma \Vdash e' : \bigwedge_{i\in I}(s\psi_i)\sigma_i$.

Finally, using the substitution lemma (Lemma I.24) on $\Gamma, x : \bigwedge_{i\in I}(s\psi_i)\sigma_i \Vdash e : t\sigma''$ and $\Gamma \Vdash e' : \bigwedge_{i\in I}(s\psi_i)\sigma_i$, we get a derivation for $\Gamma \Vdash e\{e'/x\} : t\sigma''$. We recall that the substitution σ'' is injective and thus inversible, with its inverse ϕ' being such that $\phi' \# \Gamma$ (as the image of σ'' is only composed of fresh monomorphic type variables). Thus, we have $(t\sigma'')\phi' \simeq t$, which concludes the proof. □

We now have all the results necessary to prove the type safety of programs for the parallel semantics. The parallel semantics is extended to programs in a straightforward way:

Reduction rule $\quad \texttt{let}\, x = v \,;\, p \quad \rightsquigarrow_{\mathcal{P},\mathrm{Pr}} \quad p\{v/x\}$

Evaluation Context $\quad P ::= [\,] \mid \texttt{let}\, x = [\,] \,;\, p$

$$\frac{e \rightsquigarrow_{\mathcal{P}} e'}{P[e] \rightsquigarrow_{\mathcal{P},\mathrm{Pr}} P[e']}$$

Theorem I.38 (Subject reduction for programs). *If $\Gamma \vdash_{Pr} p : t$ and $p \rightsquigarrow_{\mathcal{P},Pr} p'$, then $\Gamma \vdash_{Pr} p' : t$.*

Proof. Direct consequence of Theorem I.31 and Lemma I.37. □

Theorem I.39 (Progress for programs). *If $\varnothing \vdash_{Pr} p : t$, then either p is a value or $\exists p'.\ p \rightsquigarrow_{\mathcal{P},Pr} p'$.*

Proof. Direct consequence of Theorem I.33. □

Theorem I.40 (Type safety for the parallel semantics). *For any program p, if $\varnothing \vdash_{Pr} p : t$, then either $p \rightsquigarrow^{*}_{\mathcal{P},Pr} v$ with $\varnothing \vdash_{Pr} v : t$ or $e \rightsquigarrow^{\infty}_{\mathcal{P},Pr}$.*

Proof. Straightforward consequence of Theorem I.38 and Theorem I.39. □

I.2.5 Type Safety for the Source Semantics. The final step is to deduce a type safety theorem for the source semantics (Figure 1) from the type safety theorem for the parallel semantics. In order to help comparing the two reduction semantics, we introduce again another reduction semantics $\rightsquigarrow_C$ on expressions that can perform a reduction $\rightsquigarrow_\top$ under any context C (not just an evaluation context):

$$\begin{aligned} C ::= {} & [\,] \mid Ce \mid eC \mid (C,e) \mid (e,C) \mid \pi_i C \\ & \mid (C{\in}\tau)\ ?\, e : e \mid (e{\in}\tau)\ ?\, C : e \mid (e{\in}\tau)\ ?\, e : C \end{aligned} \qquad \frac{e \rightsquigarrow_\top e'}{C[e] \rightsquigarrow_C C[e']}$$

Definition I.41. *We say that a context C_1 is a subcontext of C_2, noted $C_1 \leq C_2$, if and only if there exists a context C_1' such that $C_2 \equiv C_1[C_1']$.*

Lemma I.42. *For any expressions e_1 and e_3, if we have a chain $e_1 \rightsquigarrow^{*}_{C} e_3$ such that at least one of the reduction steps happen under an evaluation context, then there exists an expression e_2 such that $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow^{*}_{C} e_3$.*

Proof. Let e_1 and e_4 two expressions such that $e_1 \rightsquigarrow_C^* e_4$, where at least one of the reduction steps happen under an evaluation context.

Let's consider the first reduction step $e_2 \rightsquigarrow_C e_3$ happening under an evaluation context. We have $e_1 \rightsquigarrow_C^* e_2$ (with no reduction step happening under an evaluation context) and $e_3 \rightsquigarrow_C^* e_4$. Moreover, we have $e_2 \equiv E[e_2']$ and $e_3 \equiv E[e_3']$ for some evaluation context E and expressions e_2' and e_3' such that $e_2' \rightsquigarrow_\top e_3'$.

No reduction step in $e_1 \rightsquigarrow_C^* e_2$ happen under a context C such that $C \leq E$: otherwise, it would also be an evaluation context, contradicting the fact that $e_2 \rightsquigarrow_C e_3$ is the first reduction step happening under an evaluation context. Consequently, we can "reverse" in E and e_2' the reduction steps happening in $e_1 \rightsquigarrow_C^* e_2$ (one after the other, in reverse order):

- If a reduction step happens under a context C such that $E \leq C$, it only involves e_2', we can thus apply the reverse rewriting to e_2'. After that, the expression we get is still reducible at top-level, as the reduction step that has been reversed cannot happen under an evaluation context (no reduction step in $e_1 \rightsquigarrow_C^* e_2$ can happen under an evaluation context).
- Otherwise, if a reduction step happens under a context C such that $C \nleq E$ and $E \nleq C$, it only involves E, we can thus apply the reverse rewriting to E. After that, the new context we get is still an evaluation context, as the reduction step that has been reversed cannot happen under an evaluation context (no reduction step in $e_1 \rightsquigarrow_C^* e_2$ can happen under an evaluation context).

After this reversing process, we get a new evaluation context E' and expression e_1' such that $e_1 \equiv E'[e_1']$ and $e_1' \rightsquigarrow_\top e_2''$ for some e_2'' such that $E'[e_2''] \rightsquigarrow_C^* E[e_2']$.

Consequently, we have $e_1 \equiv E'[e_1'] \rightsquigarrow E'[e_2'']$, and $E'[e_2''] \rightsquigarrow_C^* E[e_2'] \equiv e_3 \rightsquigarrow_C^* e_4$, which concludes the proof. □

Lemma I.43. *For any expressions e_1, e_2 and e_3, if $e_1 \rightsquigarrow_C^* e_2 \rightsquigarrow_\mathcal{P} e_3$, then there exists an expression e_1' such that $e_1 \rightsquigarrow e_1' \rightsquigarrow_C^* e_3$.*

Proof. The step $e_2 \rightsquigarrow_\mathcal{P} e_3$ can be decomposed into several $\rightsquigarrow_C$ steps with at least one happening under an evaluation context. Thus, this lemma is an immediate consequence of Lemma I.42 applied on the chain $e_1 \rightsquigarrow_C^* e_2 \rightsquigarrow_C^* e_3$. □

Lemma I.44. *For any expression e and value v, if $e \rightsquigarrow_C^* v$ then either $e \rightsquigarrow^\infty$ or there exists a value v' such that $e \rightsquigarrow^* v' \rightsquigarrow_C^* v$.*

Proof. If e is not already a value, then there must be at least one step in $e \rightsquigarrow_C^* v$ that happen under an evaluation context (it is not possible for v to be a value otherwise). We can thus apply Lemma I.42 successively, starting on $e \rightsquigarrow_C^* v$ and continuing until the remaining $e' \rightsquigarrow_C^* v$ chain is such that e' is a value. If this process terminates, it builds a chain $e \rightsquigarrow^* v'$ with $v' \rightsquigarrow_C^* v$, otherwise it builds $e \rightsquigarrow^\infty$. □

Lemma I.45. *For any expression e_1, e_2, and value v, if $e_1 \rightsquigarrow_C^* e_2 \rightsquigarrow_\mathcal{P}^* v$ then there exists v' such that $e_1 \rightsquigarrow^* v' \rightsquigarrow_C^* v$ or $e_1 \rightsquigarrow^\infty$.*

Proof. By induction on the number of steps in $e_2 \rightsquigarrow_\mathcal{P}^* v$, we prove using Lemma I.43 that $e_1 \rightsquigarrow^* e' \rightsquigarrow_C^* v$ for some e'. Then, we conclude by applying Lemma I.44 on $e' \rightsquigarrow_C^* v$. □

Lemma I.46. *For any expression e_1 and e_2, if $e_1 \rightsquigarrow_C^* e_2 \rightsquigarrow_\mathcal{P}^\infty$ then $e_1 \rightsquigarrow^\infty$.*

Proof. We can arbitrarily extend a chain $e_1 \rightsquigarrow \ldots$ using Lemma I.43. □

The semantics $\rightsquigarrow_C$ is extended into a semantics $\rightsquigarrow_{C,\mathsf{Pr}}$ for programs, with an extended context allowing to perform a reduction under any top-level definition of the program:

$$C_{\mathsf{Pr}} ::= [\,] \mid \texttt{let}\, x = [\,] \,;\, p \mid \texttt{let}\, x = e \,;\, C_{\mathsf{Pr}} \qquad\qquad \frac{e \rightsquigarrow_{\top} e'}{C_{\mathsf{Pr}}[C[e]] \rightsquigarrow_{C,\mathsf{Pr}} C_{\mathsf{Pr}}[C[e']]}$$

Lemma I.47. *For any program p_1, p_2, and value v, if $p_1 \rightsquigarrow^*_{C,Pr} p_2 \rightsquigarrow^*_{\mathcal{P},Pr} v$ then there exists v' such that $p_1 \rightsquigarrow^*_{Pr} v' \rightsquigarrow^*_{C,Pr} v$ or $p_1 \rightsquigarrow^{\infty}_{Pr}$.*

Proof. Straightforward induction on the number of top-level definitions in p_2, using Lemma I.45 (note that p_1 and p_2 must have the same number of top-level definitions as $p_1 \rightsquigarrow^*_{C,\mathsf{Pr}} p_2$). □

Lemma I.48. *For any program p_1 and p_2, if $p_1 \rightsquigarrow^*_{C,Pr} p_2 \rightsquigarrow^{\infty}_{\mathcal{P},Pr}$ then $p_1 \rightsquigarrow^{\infty}_{Pr}$.*

Proof. Straightforward induction on the number of top-level definitions in p_2, using Lemma I.46 (note that p_1 and p_2 must have the same number of top-level definitions as $p_1 \rightsquigarrow^*_{C,\mathsf{Pr}} p_2$). □

Lemma I.49. *For any values v_1 and v_2 such that $v_1 \rightsquigarrow^*_{C} v_2$, if $v_2 \in \tau$ then $v_1 \in \tau$.*

Proof. As v_1 is a value, every reduction step in $v_1 \rightsquigarrow^*_{C} v_2$ can only happen under a λ-abstraction. Given that the $\in$ relation ignores the body of λ-abstractions ($(\lambda x.e) \in \mathbb{0} \to \mathbb{1}$ for any e), none of the reduction steps in $v_1 \rightsquigarrow^*_{C} v_2$ can change the relation $\cdot \in \tau$. □

Theorem I.50 (Type safety for the source semantics). *For any program p, if $\varnothing \vdash_{Pr} p : \tau$, then either $p \rightsquigarrow^*_{Pr} v$ with $v \in \tau$ or $p \rightsquigarrow^{\infty}_{Pr}$.*

Proof. Straightforward combination of the type safety for the parallel semantics (Theorem I.40) with the Lemmas I.47 and I.48. In the case where we get $p \rightsquigarrow^*_{\mathsf{Pr}} v$, for some value v such that there exists v' such that $v \rightsquigarrow^*_{C} v'$ and $\varnothing \vdash_{\mathsf{Pr}} v' : \tau$, we can deduce $\varnothing \Vdash v' : \tau$, then $v' \in \tau$ using Proposition I.23, and finally $v \in \tau$ using Lemma I.49. □

I.3 Algorithmic Type System

I.3.1 Maximal Sharing Canonical Form. This section applies to definitions of Appendix E.

As defined in Section I.1, we consider that expressions of the source language can contain binding variables. In particular, the unwinding operator $\lceil . \rceil$ can be used on atoms and canonical forms containing free binding variables. An expression without binding variables is called ground expression.

Proposition I.51. *For any ground expression of the source language e, $\lceil \mathsf{term}([\![e]\!]) \rceil \equiv e$.*

Proof. Straightorward structural induction on e. □

Proposition I.52. *If $\kappa \equiv_\kappa \kappa'$, then $\lceil \kappa \rceil \equiv_\alpha \lceil \kappa' \rceil$.*

Proof. If a reordering, as defined in Definition 3.1, applies at top-level on the expression $\texttt{bind}\, \mathsf{x}_1 = a_1 \,\texttt{in}\, \texttt{bind}\, \mathsf{x}_2 = a_2 \,\texttt{in}\, \kappa$, the unwinding remains unchanged: as $\mathsf{x}_1 \notin \mathsf{fv}(a_2)$ and $x_2 \notin \mathsf{fv}(a_1)$, we have $\kappa\{a_1/\mathsf{x}_1\}\{a_2/\mathsf{x}_2\} \equiv \kappa\{a_2/\mathsf{x}_2\}\{a_1/\mathsf{x}_1\}$.

The general case can easily be deduced with the observation that $\forall C, \kappa_1, \kappa_2.\ \lceil \kappa_1 \rceil \equiv_\alpha \lceil \kappa_2 \rceil \Rightarrow \lceil C[\kappa_1] \rceil \equiv_\alpha \lceil C[\kappa_2] \rceil$ (with C denoting an arbitrary context). □

Proposition I.53 (Equivalence of MSC-forms). *If κ_1 and κ_2 are two MSC-forms and $\lceil \kappa_1 \rceil \equiv_\alpha \lceil \kappa_2 \rceil$, then $\kappa_1 \equiv_\kappa \kappa_2$.*

Proof. We will show that κ_2 can be transformed into κ_1 just with α-renaming and reordering of independent bindings (as specified in the definition of $\equiv_\kappa$).

In this proof, we represent *partially unwinded canonical forms* by a pair (B, e), where B is a binding context and e an expression. With this representation, the unwinding of (B, e) is eB, but for clarity we can also use the notation $\lceil (B, e) \rceil$.

Let (B_1, x_1) be the representation of κ_1, with B_1 representing its top-level definitions and x_1 its final binding variable, and (B_2, x_2) be the representation of κ_2. Formally, we have $\mathsf{term}(B_1, \mathsf{x}_1) \equiv \kappa_1$ and $\mathsf{term}(B_2, \mathsf{x}_2) \equiv \kappa_2$.

As $\lceil\kappa_1\rceil \equiv_\alpha \lceil\kappa_2\rceil$, we have $\lceil(B_1, \mathsf{x}_1)\rceil \equiv_\alpha \lceil(B_2, \mathsf{x}_2)\rceil$. By α-renaming, we can assume that $\mathsf{x}_1 = \mathsf{x}_2 = \mathsf{x}$ and $\lceil(B_1, \mathsf{x})\rceil \equiv \lceil(B_2, \mathsf{x})\rceil$.

Now, let's prove the property below, from which Proposition I.53 can be deduced. *Let B_1 and B_2 two binding contexts, and e an expression such that:*

- $\lceil(B_1, e)\rceil \equiv \lceil(B_2, e)\rceil$
- *The body of λ-abstractions in B_1 and B_2 are in MSC-form (Definition 3.2)*
- *Both B_1 and B_2 satisfy the following properties (corresponding to the MSC-form properties applied to the top-level definitions), written here for a binding context B:*

  (1) if (x_1, a_1) and (x_2, a_2) are distinct definitions in B, then $a_1 \not\equiv_\kappa a_2$
  (2) for any definition $(\mathsf{x}, \lambda z.\kappa)$ in B, any binding $\mathtt{bind}\ \mathsf{y} = a\ \mathtt{in}\ \kappa'$ in κ is such that $\mathsf{fv}(a) \not\subseteq \mathsf{fv}(\lambda z.\kappa)$
  (3) if B contains a definition (x, a), then x is a free variable of one of the next definitions in B or of e

Then, we can transform B_2 into B_1 just with α-renaming, reordering of independent definitions, and replacement of an atom by a $\equiv_\kappa$-equivalent one.

We prove this property by induction on the total number of atoms appearing in B_1 (by counting top-level atoms as well as the sub-atoms they contain).

The base case ($B_1 = \varepsilon$) is trivial: as $\lceil(B_2, e)\rceil \equiv \lceil(B_1, e)\rceil \equiv \lceil(\varepsilon, e)\rceil \equiv e$, we deduce with Property 3 of MSC-forms that $B_2 = \varepsilon$.

For the inductive case, let's consider $B_1 = B_1'\ ;\ (\mathsf{x}, a_1)$.

With property 3 of MSC-forms, we know that x appears in e. As $\lceil(B_1, e)\rceil \equiv \lceil(B_2, e)\rceil$, we can deduce that $\lceil(B_1, \mathsf{x})\rceil \equiv \lceil(B_2, \mathsf{x})\rceil$. 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, a_2) at the end (if not already), it gives $B_2 = B_2'\ ;\ (\mathsf{x}, a_2)$. We then have $\lceil(B_1', a_1)\rceil \equiv \lceil(B_2', a_2)\rceil$. 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 λ-abstractions and that do not contain any binding variable (constants, lambda variables), we directly have $a_1 \equiv a_2$.
- If a_1 and a_2 are atoms that are not λ-abstractions and that contain only one binding variable (projections), we can α-rename binding variables in B_2 so that $a_1 \equiv a_2$.
- If a_1 and a_2 are atoms that are not λ-abstractions and that contain two binding variables x and y (applications, pairs), we consider two cases:
  - If $\lceil(B_1', \mathsf{x})\rceil \equiv_\alpha \lceil(B_1', \mathsf{y})\rceil$, let's show that we necessarily have $\mathsf{x} = \mathsf{y}$. We consider the binding context 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 binding context B_y where the definitions unrelated to y have been removed. Then, we apply the induction hypothesis to the binding contexts B_x, $B_\mathsf{y}\{\mathsf{x}/\mathsf{y}\}$ and the expression x. This tells us that B_x and B_y are equivalent modulo reordering of the definitions, α-renaming and $\equiv_\kappa$-transformation of atoms. Thus, according to the property 1 of MSC-forms, x and y cannot have two distinct definitions in B_1', and thus $\mathsf{x} = \mathsf{y}$. 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 α-rename binding variables in B_2 so that $a_1 \equiv a_2$.
  - Otherwise, x and y must be different, and the same applies to a_2. Thus, we can α-rename binding variables in B_2 so that $a_1 \equiv a_2$.
- If a_1 and a_2 are typecases (containing 3 binding variables), we proceed similarly to the previous case to obtain $a_1 \equiv a_2$.

- In the case where a_1 and a_2 are λ-abstractions, let's say $\lambda x.\ \kappa_1$ and $\lambda x.\ \kappa_2$, we note $(B_{\mathtt{x}_1}, \mathtt{x}_1)$ and $(B_{\mathtt{x}_2}, \mathtt{x}_2)$ the representations of κ_1 and κ_2 respectively. We now consider the representation $(B'_1; B_{\mathtt{x}_1}, \lambda x.\ \mathtt{x}_1)$ and remove from it all the unused definitions (i.e. not related to $\mathtt{x}_1$), it gives us a new representation $(B''_1, \lambda x.\ \mathtt{x}_1)$. We do the same for $(B'_2; B_{\mathtt{x}_2}, \lambda x.\ \mathtt{x}_2)$, it gives a new representation $(B''_2, \lambda x.\ \mathtt{x}_2)$. Then, we use the induction hypothesis on B''_1, $B''_2\{\mathtt{x}_1/\mathtt{x}_2\}$ and $\lambda x.\ \mathtt{x}_1$. It gives us that B''_1 and B''_2 are equivalent modulo reordering of the definitions, α-renaming and $\equiv_\kappa$-transformation of the atoms. By using the property 2 of MSC-forms, we can deduce that $B_{\mathtt{x}_1}$ and $B_{\mathtt{x}_2}$ are equivalent modulo reordering of the definitions, α-renaming and $\equiv_\kappa$-transformation of the atoms. Thus, we can α-rename B_2 and $\equiv_\kappa$-transform some of its atoms so that $B_{\mathtt{x}_1} \equiv B_{\mathtt{x}_2}$, and thus $a_1 \equiv a_2$.

In any case, we get $a_1 \equiv a_2 \equiv a$, 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'_2 and $e\{a/\mathtt{x}\}$. □

Proposition I.54. *If $\kappa \dashrightarrow \kappa'$, then $\lceil\kappa\rceil \equiv_\alpha \lceil\kappa'\rceil$.*

Proof. Similar to Proposition I.52. □

Proposition I.55 (Normalisation). *There is no infinite chain $\kappa_1 \dashrightarrow \kappa_2 \dashrightarrow \cdots$*

Proof. Let n be the maximal number of nested λ-abstractions in κ_1. We call depth of a binding the number of nested λ-abstractions it is into (the depth of a binding of κ_1 is at most n).

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

For any chain $\kappa_1 \dashrightarrow \kappa_2 \dashrightarrow \cdots$, the sequence $S(\kappa_1), S(\kappa_2), \ldots$ is strictly decreasing with respects to the lexicographic order. Thus $\kappa_1 \dashrightarrow \kappa_2 \dashrightarrow \cdots$ cannot be infinite. □

Proposition I.56. *If $\kappa \not\dashrightarrow$ (i.e., no $\dashrightarrow$ rule apply on κ), then κ is an MSC-form.*

Proof. We assume $\kappa \not\dashrightarrow$ and show that all 3 MSC properties are satisfied.

The property 3 (no unused bindings) is trivially verified: any binding that does not satisfy this property can directly be eliminated with the rule 4. As the rule 4 does not apply, this property must be satisfied.

Now, we focus on the property 2 (extrusion of bindings). We assume that there exists a sub-expression $\lambda x.\ \kappa_1$ of κ and a sub-expression $\mathtt{bind}\, \mathtt{y} = a \,\mathtt{in}\, \kappa_2$ of κ_1 such that $\mathsf{fv}(a) \subseteq \mathsf{fv}(\lambda x.\ \kappa_1)$. We know that a does not depend on x, otherwise x would be in $\mathsf{fv}(a)$ and not in $\mathsf{fv}(\lambda x.\ \kappa_1)$. Thus, we can reorder the binding y (6) in the first position of its inner-most containing lambda-abstraction, and then apply the rule 5 on it, which contradicts $\kappa \not\dashrightarrow$. Consequently, the property 2 is satisfied.

Finally, we show that the property 1 (sharing) is also satisfied. We assume that there are two distinct bindings $\mathtt{bind}\, \mathtt{x}_1 = a_1 \,\mathtt{in}\, \ldots$ and $\mathtt{bind}\, \mathtt{x}_2 = a_2 \,\mathtt{in}\, \ldots$ such that $a_1 \equiv_\kappa a_2$. As the property 2 is satisfied, and as $\mathsf{fv}(a_1) = \mathsf{fv}(a_2)$, we know that these two bindings are in the same λ-abstraction. Thus, we can reorder them (6) to be the one next to the other so that the rule 3 is applicable, which contradicts $\kappa \not\dashrightarrow$. Thus, the property 1 is satisfied. □

Proposition I.57 (Confluence). *Let κ_1, κ_2 and κ'_2 such that $\kappa_1 \dashrightarrow^* \kappa_2$ and $\kappa_1 \dashrightarrow^* \kappa'_2$. Then, there exists κ_3 and κ'_3 such that $\kappa_2 \dashrightarrow^* \kappa_3$, $\kappa'_2 \dashrightarrow^* \kappa'_3$, and $\kappa_3 \equiv_\kappa \kappa'_3$.*

Proof. Immediate consequence of Proposition I.55 (normalisation), Proposition I.54 (preservation of $\lceil.\rceil$), Proposition I.56 ($\not\dashrightarrow$ implies MSC-form), and Proposition I.53 (equivalence of MSC-forms). □

I.3.2 Soundness. See Appendix G for the full algorithmic system, without the rules for extensions.

Lemma I.58 (Monotonicity).
If $\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t$ *and* $\Gamma' \leq_P \Gamma$*, then* $\exists \mathbb{k}', t'.\ \Gamma' \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}'] : t'$ *with* $t' \leq_P t$*.*
If $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$ *and* $\Gamma' \leq_P \Gamma$*, then* $\exists \mathbb{a}', t'.\ \Gamma' \vdash_{\mathcal{A}} [a \mid \mathbb{a}'] : t'$ *with* $t' \leq_P t$*.*

Proof. Straightforward structural induction on the derivation. □

Theorem I.59 (Soundness). *If* $\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t$*, then* $\Gamma \Vdash \lceil \kappa \rceil : t$*. If* $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$*, then* $\Gamma \Vdash \lceil a \rceil : t$*.*

Proof. We proceed by structural induction on the typing derivation of $\Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t$ (resp. $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$) in order to build a derivation $\Gamma \Vdash \lceil \kappa \rceil : t$ (resp. $\Gamma \Vdash \lceil a \rceil : t$). We consider the root of the derivation:

[Const-Alg] Trivial (we use a [Const] node).

[Ax-Alg] Trivial (we use a [Ax_λ] node).

[→I-Alg] We have $a \equiv \lambda x.\ \kappa$, and thus $\lceil a \rceil \equiv \lambda x.\ \lceil \kappa \rceil$.
By induction on the premise, we get $\Gamma, x : \mathbf{u} \Vdash \lceil \kappa \rceil : s$. By applying the rule [→I], we get $\Gamma \Vdash \lceil a \rceil : \mathbf{u} \to s$ (with $t \simeq \mathbf{u} \to s$).

[→E-Alg] We have $a \equiv \mathtt{x}_1 \mathtt{x}_2$. We pose $t_1 \overset{\text{def}}{=} \Gamma(\mathtt{x}_1)\Sigma_1$ and $t_2 \overset{\text{def}}{=} \Gamma(\mathtt{x}_2)\Sigma_2$.
With an [$\text{Ax}_\vee$] node, we can derive $\Gamma \Vdash \mathtt{x}_1 : \Gamma(\mathtt{x}_1)$ and $\Gamma \Vdash \mathtt{x}_2 : \Gamma(\mathtt{x}_2)$. Using a [Inst∧≤] pattern, we can derive from that $\Gamma \Vdash \mathtt{x}_1 : t_1$ and $\Gamma \Vdash \mathtt{x}_2 : t_2$. We have $t \simeq t_1 \circ t_2$. Thus, according to the definition of $\circ$, we know that $t_1 \leq t_2 \to t$. Thus, with an application of the [≤] rule on $\Gamma \Vdash \mathtt{x}_1 : t_1$, we can derive $\Gamma \Vdash \mathtt{x}_1 : t_2 \to t$. We can then conclude with an application of the [→E] rule.

[×I-Alg] We have $a \equiv (\mathtt{x}_1, \mathtt{x}_2)$.
With an [$\text{Ax}_\vee$] node, we can derive $\Gamma \Vdash \mathtt{x}_1 : \Gamma(\mathtt{x}_1)\rho_1$ and $\Gamma \Vdash \mathtt{x}_2 : \Gamma(\mathtt{x}_2)\rho_2$ (with ρ_1 and ρ_2 as in the [×I-Alg] node). We can then conclude with an application of the [×I] rule.

[$\times\text{E}_1$-Alg] We have $a \equiv \pi_1 \mathtt{x}$. We pose $t_\circ = \Gamma(\mathtt{x})\Sigma$.
With an [$\text{Ax}_\vee$] node, we can derive $\Gamma \Vdash \mathtt{x} : \Gamma(\mathtt{x})$. Using a [Inst∧≤] pattern, we can derive from that $\Gamma \Vdash \mathtt{x} : t_\circ$. We have $t \simeq \boldsymbol{\pi}_1(t_\circ)$. Thus, according to the definition of $\boldsymbol{\pi}_1$, we know that $t_\circ \leq t \times \mathbb{1}$. Thus, with an application of the [≤] rule, we can derive $\Gamma \Vdash \mathtt{x} : t \times \mathbb{1}$. We can then conclude with an application of the [$\times\text{E}_1$] rule.

[$\times\text{E}_2$-Alg] Similar to the previous case.

[$\mathbb{0}$-Alg] Similar to the previous case.

[$\in_1$-Alg] Similar to the previous case.

[$\in_2$-Alg] Similar to the previous case.

[Var-Alg] Trivial (we use a [$\text{Ax}_\vee$] node).

[Bind_1-Alg] We have $\kappa \equiv \mathtt{bind}\,\mathtt{x} = a\,\mathtt{in}\,\kappa_\circ$ and thus $\lceil \kappa \rceil \equiv \lceil \kappa_\circ \rceil \{\lceil a \rceil / \mathtt{x}\}$.
By induction on the premise, we get $\Gamma \Vdash \lceil \kappa_\circ \rceil : t$. As $\mathtt{x} \notin \mathrm{dom}(\Gamma)$, we know that this derivation does not contain any [$\text{Ax}_\vee$] node applied on x. We can thus easily transform it into a derivation of $\Gamma \Vdash \lceil \kappa_\circ \rceil \{\lceil a \rceil / \mathtt{x}\} : t$ by substituting every occurrence of x by $\lceil a \rceil$.

[Bind_2-Alg] We have $\kappa \equiv \mathtt{bind}\,\mathtt{x} = a\,\mathtt{in}\,\kappa_\circ$ and thus $\lceil \kappa \rceil \equiv \lceil \kappa_\circ \rceil \{\lceil a \rceil / \mathtt{x}\}$.
We can suppose without loss of generality that the decomposition $\{\mathbf{u}_i\}_{i \in I}$ is a partition of $\mathbb{1}$: if for any two distinct $i, j \in I$, $\mathbf{u}_i$ and $\mathbf{u}_j$ are not disjoint, then we can replace the $\mathbf{u}_j$ part by $\mathbf{u}_j \setminus \mathbf{u}_i$ and conclude by monotonicity (Lemma I.58).
By induction on the first premise, we get $\Gamma \Vdash \lceil a \rceil : s$. For any $i \in I$, we apply the induction hypothesis on the corresponding premise. It gives $\Gamma, \mathtt{x} : s \wedge \mathbf{u}_i \Vdash \lceil \kappa_\circ \rceil : t_i$. With a [≤] node, we can obtain $\Gamma, \mathtt{x} : s \wedge \mathbf{u}_i \Vdash \lceil \kappa_\circ \rceil : t$ (with $t \simeq \bigvee_{i \in I} t_i$). We conclude with a [∨] node.

[∧-Alg] Trivial induction on the premises.

□

I.3.3 Completeness.

Definition I.60 (Atomic source expression). *We say that an expression of the source language is an atomic source expression if it can be constructed with the following syntax:*

$$\textbf{\textit{Atomic source expressions}} \quad \bar{a} \quad ::= \quad c \mid x \mid \lambda x.e \mid (\mathsf{x},\mathsf{x}) \mid \mathsf{x}\mathsf{x} \mid \pi_i\mathsf{x} \mid (\mathsf{x}\in\tau)\,?\,\mathsf{x}:\mathsf{x}$$

and if, for the case $\lambda x.e$*, all sub-expressions of* e *are either a binding variable or they contain a lambda variable that is not in* $\mathsf{fv}(\lambda x.e)$.

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

Definition I.61. *For any atomic source expression* $\bar{a}$*, we define* $\mathsf{MSCA}(\bar{a})$ *as follows:*

$$\mathsf{MSCA}(\lambda x.e) = \lambda x.\mathsf{MSC}(e)$$
$$\mathsf{MSCA}(\bar{a}) = \bar{a} \qquad \text{for any } \bar{a} \text{ that is not a } \lambda\text{-abstraction}$$

Proposition I.62. *For any atomic source expression* $\bar{a}$*,* $\mathtt{bind}\,\mathsf{x} = \mathsf{MSCA}(\bar{a})\ \mathtt{in}\,\mathsf{x}$ *is a valid MSC-form.*

Proof. The extrusion property is ensured by the condition on λ-abstractions in Definition I.60. The two other properties are trivially satisfied. □

Definition I.63. *For a binding context* B *and an expression* e*, we define the operator* $e \setminus B$ *as follows:*

$$e \setminus \varepsilon = e$$
$$e \setminus ((\mathsf{x}, a); B) = (e\{\mathsf{x}/\lceil a\rceil\}) \setminus B$$

Proposition I.64. *For any binding context* B *and expression* e*, we have* $(e \setminus B)B \equiv_\alpha eB$ *and* $(eB) \setminus B \equiv_\alpha e \setminus B$.

Proof. Straightforward induction on B. □

In the following, we fix an expression order $\leq$ over expressions (c.f. Desfinition I.9). This order should be total, so that for any expression e, it determines a unique MSC-form $\mathsf{MSC}(e)$ modulo α-renaming (and not only modulo $\equiv_\kappa$): independent consecutive bindings must follow the increasing order $\leq$ of their unwinding. As the order $\leq$ is arbitrarily choosen, the proofs below will work for any MSC-form.

Lemma I.65 (Decomposition of canonical form derivations). *If* D *is a canonical form derivation of* $\Gamma \vdash e : t$*, with the root being a* $[\vee]$ *node doing the substitution* $e'\{e_{\mathsf{x}}/\mathsf{x}\}$*, then there exists a binding context* B *such that* $\mathsf{MSC}(e) \equiv_\alpha \mathsf{term}(B, \mathtt{bind}\,\mathsf{x} = \mathsf{MSCA}(e_{\mathsf{x}} \setminus B)\ \mathtt{in}\,\mathsf{MSC}(e' \setminus B)\,)$.

Proof. First, we deduce from the fact that D is a canonical form derivation that the definition premise, $\Gamma \vdash e_{\mathsf{x}} : s$, is an atomic derivation.

We know that e_{x} appears in e (as D is canonical, see Definition I.7). Thus, we can deduce that $\mathsf{MSC}(e)$ contains a definition for an atom a that unwinds to e_{x}. Formally, we know that there exists a binding context B, an atom a and a canonical form κ such that $\mathsf{MSC}(e) \equiv_\alpha \mathsf{term}(B, \mathtt{bind}\,\mathsf{x} = a\ \mathtt{in}\,\kappa\,)$ with $\lceil a\rceil B \equiv_\alpha e_{\mathsf{x}}$.

First, we determine what B is. The expression e_{x} could contain some sub-expressions that are not binding variables and that have no occurrence of a lambda variable defined in e_{x}. In this case, these sub-expressions should be defined by some bindings in B (the unwinding of each such binding is necessarily smaller than e_{x} by $\leq$, as $\leq$ is an extension of the sub-expressions order). The expression e' could also contain some such sub-expressions. The ones whose unwnding is smaller than e_{x} according to $\leq$ must be defined by some bindings in B too. No other expression should be defined in B or it would contradict the properties of MSC-forms.

Now, we determine what a is. Under the context B, the expression $e_{\mathsf{x}} \setminus B$ unwinds to e_{x} (Proposition I.64). Moreover, as $\Gamma \Vdash e_{\mathsf{x}} : s$ is an atomic derivation, and given how B is constructed, $e_{\mathsf{x}} \setminus B$ must be an atomic source expression. Thus, we can deduce from Proposition I.62 that $\mathsf{MSCA}(e_{\mathsf{x}} \setminus B)$ 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 (Proposition I.53) we can conclude that $a \equiv_\alpha \mathsf{MSCA}(e_{\mathsf{x}} \setminus B)$.

Finally, we determine what κ is. We note B' the binding context $B; (\mathsf{x}, \mathsf{MSCA}(e_{\mathsf{x}} \setminus B))$. The expression $e' \setminus B'$ unwinds to $e'\{e_{\mathsf{x}}/\mathsf{x}\}$ under the context B' (Proposition I.64). As D is a canonical form, e_{x} cannot be a sub-expression of e' (c.f. Definition I.7), thus $e' \setminus B \equiv_\alpha e' \setminus B'$ and thus $e' \setminus B$ also unwinds to $e'\{e_{\mathsf{x}}/\mathsf{x}\}$. Thus, $\mathsf{MSC}(e' \setminus B)$ can be used in place of κ without breaking any property of the MSC-form (note that it only contains top-level bindings for expressions whose unwinding is greater than e_{x} by $\leq$, as the smaller ones have been put in B). By unicity of the MSC-form (Proposition I.53), we conclude that $\kappa \equiv_\alpha \mathsf{MSC}(e' \setminus B)$, and thus $\mathsf{MSC}(e) \equiv_\alpha \mathsf{term}(B, \mathsf{bind}\,\mathsf{x} = \mathsf{MSCA}(e_{\mathsf{x}} \setminus B) \,\mathsf{in}\, \mathsf{MSC}(e' \setminus B)\,)$. □

Lemma I.66 (Completeness). *If D is a canonical form derivation of $\Gamma \Vdash e : t$, then $\exists \Bbbk, t'.\ \Gamma \vdash_{\mathscr{A}} [\mathsf{MSC}(e) \mid \Bbbk] : t'$ with $t' \leq_P t$.*
If D is a canonical atomic derivation of $\Gamma \Vdash \bar{a} : t$ (with $\bar{a}$ an atomic source expression), then $\exists \mathbb{a}, t'.\ \Gamma \vdash_{\mathscr{A}} [\mathsf{MSCA}(\bar{a}) \mid \mathbb{a}] : t'$ with $t' \leq_P t$.

Proof. We proceed by induction on the depth of D.

We consider the root of the derivation (the cases up to $[\in_2]$ are for canonical atomic derivations, the cases after are for canonical form derivations):

[Const] Trivial.

[Ax$_\lambda$] Trivial.

[→I] We have $\bar{a} \equiv \lambda\mathsf{x}.\ e$ and thus $\mathsf{MSCA}(\bar{a}) \equiv_\alpha \lambda\mathsf{x}.\ \mathsf{MSC}(e)$.
The premise of this [→I] node is a canonical form derivation. Thus, by induction on this premise, we get $\Gamma, \mathsf{x} : \mathbf{u} \vdash_{\mathscr{A}} [\mathsf{MSC}(e) \mid \Bbbk] : t'$ (with $t' \leq_P t$). We can thus derive $\Gamma \vdash_{\mathscr{A}} [\lambda\mathsf{x}.\ \mathbb{MSC}(e) \mid \lambda(\mathbf{u}, \Bbbk)] : \mathbf{u} \to t'$ and we have $\mathbf{u} \to t' \leq_P \mathbf{u} \to t$, which concludes this case.

[→E] We have $\bar{a} \equiv \mathsf{x}_1\mathsf{x}_2$ and thus $\mathsf{MSCA}(\bar{a}) \equiv_\alpha \mathsf{x}_1\mathsf{x}_2$.
As D is a canonical atomic derivation, we know that the second premise, $\Gamma \Vdash \mathsf{x}_2 : t_1$, is a [Inst∧≤] pattern with no [≤] node and whose premise is a [Ax$_\vee$] node. Thus, we know that there exists Σ_2 such that $\Gamma(\mathsf{x}_2)\Sigma_2 \simeq t_1$. Similarly, the first premise, $\Gamma \Vdash \mathsf{x}_1 : t_1 \to t_2$, is a [Inst∧≤] pattern whose premise is a [Ax$_\vee$] node. Thus, we know that there exists Σ_1 such that $\Gamma(\mathsf{x}_1)\Sigma_1 \leq t_1 \to t_2$.
Consequently, and by definition of $\circ$, we know that $(\Gamma(\mathsf{x}_1)\Sigma_1) \circ (\Gamma(\mathsf{x}_2)\Sigma_2) \leq t_2$. We can thus derive $\Gamma \vdash_{\mathscr{A}} [\mathsf{x}_1\mathsf{x}_2 \mid @(\Sigma_1, \Sigma_2)] : t'$ (with $t' \simeq (\Gamma(\mathsf{x}_1)\Sigma_1) \circ (\Gamma(\mathsf{x}_2)\Sigma_2)$) such that $t' \leq t_2$, which concludes this case.

[×I] We have $\bar{a} \equiv (\mathsf{x}_1, \mathsf{x}_2)$ and thus $\mathsf{MSCA}(\bar{a}) \equiv_\alpha (\mathsf{x}_1, \mathsf{x}_2)$.
As D is a canonical atomic derivation, both premises can only be [Ax$_\vee$] nodes. Thus, we can deduce that there exists two renamings of polymorphic type variables ρ_1 and ρ_2 such that $\Gamma(\mathsf{x}_1)\rho_1 \simeq t_1$ and $\Gamma(\mathsf{x}_2)\rho_2 \simeq t_2$. Thus, we can derive $\Gamma \vdash_{\mathscr{A}} [(\mathsf{x}_1, \mathsf{x}_2) \mid (\rho_1, \rho_2)] : t_1 \times t_2$.

[×E$_1$] We have $\bar{a} \equiv \pi_1\mathsf{x}$ and thus $\mathsf{MSCA}(\bar{a}) \equiv_\alpha \pi_1\mathsf{x}$.
As D is a canonical atomic derivation, we know that the premise, $\Gamma \Vdash \mathsf{x} : t_1 \times t_2$, is a [Inst∧≤] pattern whose premise is a [Ax$_\vee$] node. Thus, we know that there exists Σ such that $\Gamma(\mathsf{x})\Sigma \leq t_1 \times t_2$.
Consequently, and by definition of $\boldsymbol{\pi}_1$, we know that $\boldsymbol{\pi}_1(\Gamma(\mathsf{x})\Sigma) \leq t_1$. We can thus derive $\Gamma \vdash_{\mathscr{A}} [\pi_1\mathsf{x} \mid \pi(\Sigma)] : t'$ (with $t' \simeq \boldsymbol{\pi}_1(\Gamma(\mathsf{x})\Sigma)$) such that $t' \leq t_1$, which concludes this case.

[×E$_2$] Similar to the previous case.

[$\mathbb{0}$] We have $\bar{a} \equiv (\mathtt{x}{\in}\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2$ and thus $\mathsf{MSCA}(\bar{a}) \equiv_\alpha (\mathtt{x}{\in}\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2$.
As D is a canonical atomic derivation, we know that the premise, $\Gamma \Vdash \mathtt{x} : \mathbb{0}$, is a [Inst$\wedge\leq$] pattern with no [$\leq$] node and whose premise is a [Ax$_\vee$] node. Thus, we know that there exists Σ such that $\Gamma(\mathtt{x})\Sigma \simeq \mathbb{0}$.
We can thus derive $\Gamma \vdash_{\mathcal{A}} [(\mathtt{x}{\in}\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2 \mid \mathbb{0}(\Sigma)] : \mathbb{0}$.

[$\in_1$] We have $\bar{a} \equiv (\mathtt{x}{\in}\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2$ and thus $\mathsf{MSCA}(\bar{a}) \equiv_\alpha (\mathtt{x}{\in}\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2$.
As D is a canonical atomic derivation, we know that the first premise, $\Gamma \Vdash \mathtt{x} : \tau$, is a [Inst$\wedge\leq$] pattern whose premise is a [Ax$_\vee$] node. Thus, we know that there exists Σ such that $\Gamma(\mathtt{x})\Sigma \leq \tau$. Similarly, the second premise, $\Gamma \Vdash \mathtt{x}_1 : t_1$, can only be a [Ax$_\vee$]. Thus, we know that there exists a renaming of polymorphic variables ρ such that $\Gamma(\mathtt{x}_1)\rho \simeq t_1$.
We can thus derive $\Gamma \vdash_{\mathcal{A}} [(\mathtt{x}{\in}\tau)\,?\,\mathtt{x}_1 : \mathtt{x}_2 \mid \in_1(\Sigma)] : \Gamma(\mathtt{x}_1)$ with $\Gamma(\mathtt{x}_1) \leq_{\mathsf{P}} t_1$.

[$\in_2$] Similar to the previous case.

[Ax$_\vee$] Trivial.

[$\leq$] Straightforward induction on the premise.

[Inst] Straightforward induction on the premise.

[$\wedge$] By induction on the premises, we get $\forall i \in I.\ \Gamma \vdash_{\mathcal{A}} [\mathsf{MSC}(e) \mid \mathbb{k}_i] : t_i'$ with $t_i' \leq_{\mathsf{P}} t_i$. Thus, we can derive $\Gamma \vdash_{\mathcal{A}} [\mathsf{MSC}(e) \mid \bigwedge(\{\mathbb{k}_i\}_{i\in I})] : \bigwedge_{i\in I} t_i'$ (with $\bigwedge_{i\in I} t_i' \leq_{\mathsf{P}} \bigwedge_{i\in I} t_i$).

[$\vee$] By using Lemma I.65, we know that there exists B such that $\mathsf{MSC}(e) \equiv_\alpha B[\mathtt{bind}\,\mathtt{x} = \mathsf{MSCA}(e_{\mathtt{x}} \setminus B)\ \mathtt{in}\ \mathsf{MSC}(e' \setminus B)\,]$ (with $e_{\mathtt{x}} \setminus B$ being an atomic source expression).
The unwiding of the top-level definitions in B are necessarily smaller than $e_{\mathtt{x}}$ by $\leq$. Thus, none of them can be defined by a [$\vee$] node in D. Consequently, and as in canonical form derivation a structural node can only appear in the first premise of a [$\vee$] node, none of the expressions defined in B are typed in D. Thus, these sub-expressions can easily be substituted, in D, by the associated binding variables in B. It yields a derivation for $\Gamma \Vdash (e'\{e_{\mathtt{x}}/\mathtt{x}\}) \setminus B : t$, or equivalently, for $\Gamma \Vdash (e' \setminus B)\{(e_{\mathtt{x}} \setminus B)/\mathtt{x}\} : t$.
By induction on the premises of this new derivation, we get $\Gamma \vdash_{\mathcal{A}} [\mathbb{MSCA}(e_{\mathtt{x}} \setminus B) \mid \mathbb{a}] : s'$ (with $s' \leq_{\mathsf{P}} s$) and $\forall i \in I.\ \Gamma, \mathtt{x} : s \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\mathsf{MSC}(e' \setminus B) \mid \mathbb{k}_i] : t_i$ (with $t_i \leq_{\mathsf{P}} t$). By monotonicity (Lemma I.58), we can derive $\forall i \in I.\ \Gamma, \mathtt{x} : s' \wedge \mathbf{u}_i \vdash_{\mathcal{A}} [\mathsf{MSC}(e' \setminus B) \mid \mathbb{k}_i'] : t_i'$ (with $t_i' \leq_{\mathsf{P}} t_i \leq_{\mathsf{P}} t$). We can thus derive $\Gamma \vdash_{\mathcal{A}} [\mathtt{bind}\,\mathtt{x} = \mathsf{MSCA}(e_{\mathtt{x}} \setminus B)\ \mathtt{in}\ \mathsf{MSC}(e' \setminus B) \mid \mathbb{k}] : \bigvee_{i\in I} t_i'$ with $\mathbb{k} = \mathtt{keep}\ (\mathbb{a}, \{(\mathbf{u}_i, \mathbb{k}_i')\}_{i\in I})$.
From that, we can easily derive $\Gamma \vdash_{\mathcal{A}} [\mathsf{MSC}(e) \mid \mathbb{k}'] : \bigvee_{i\in I} t_i'$ with $\mathbb{k}'$ obtained by inserting at the root of $\mathbb{k}$ a $\mathtt{skip}$ annotation for each definition in B.

□

Theorem I.67 (Completeness). *If $\Gamma \Vdash e : t$ with e a ground expression, then $\exists \mathbb{k}, t'.\ \Gamma \vdash_{\mathcal{A}} [\mathsf{MSC}(e) \mid \mathbb{k}] : t'$ with $t' \leq_{P} t$.*

Proof. Direct application of Lemma I.66 after using the normalisation theorem (Theorem I.22) on a derivation of $\Gamma \Vdash e : t$. □

I.4 Annotations Reconstruction System

This section contains proofs for the reconstruction system (Appendix H).

Theorem I.68 (Soundness). *If $\Gamma \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{k}$, then $\exists t.\ \Gamma \vdash_{\mathcal{A}} [\kappa \mid \mathbb{k}] : t$.*
If $\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$, then $\exists t.\ \Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$.

Proof. We proceed by structural induction on the derivation of $\Gamma \vdash_{\mathcal{P}} \langle \kappa \mid \mathcal{K} \rangle \Rightarrow \mathbb{k}$ or $\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$.

If the derivation $\Gamma \vdash_{\mathcal{P}} \langle a \mid \mathcal{A} \rangle \Rightarrow \mathbb{a}$ has a [App] root, we construct a derivation $\Gamma \vdash_{\mathcal{A}} [a \mid \mathbb{a}] : t$ (for some t) with a [$\to$E-Alg] root. In order to satisfy the guard-conditions of the rule [$\to$E-Alg], we need to prove that the annotation $@(\Sigma_1, \Sigma_2)$ generated by the [App] root satisfies $\Gamma(\mathtt{x}_1)\Sigma_1 \leq \mathbb{0} \to \mathbb{1}$ and $\Gamma(\mathtt{x}_2)\Sigma_2 \leq \mathsf{dom}(\Gamma(\mathtt{x}_1)\Sigma_1)$.

We have, in the premise of the [App] root, $\Sigma = \mathsf{tally}(\{\Gamma(\mathtt{x}_1)\rho_1 \dot{\leq} \Gamma(\mathtt{x}_2)\rho_2 \to \alpha\})$ and $\Sigma \neq \varnothing$. Let $\sigma \in \Sigma$. By definition of the tallying problem, we have $(\Gamma(\mathtt{x}_1)\rho_1)\sigma \leq (\Gamma(\mathtt{x}_2)\rho_2 \to \alpha)\sigma$, which can be rewritten $\Gamma(\mathtt{x}_1)(\sigma \circ \rho_1) \leq (\Gamma(\mathtt{x}_2)(\sigma \circ \rho_2)) \to (\alpha\sigma)$. From that subtyping relation, we can deduce $\Gamma(\mathtt{x}_1)(\sigma \circ \rho_1) \leq \mathbb{0} \to \mathbb{1}$, and by definition of $\mathsf{dom}(.)$, $\Gamma(\mathtt{x}_2)(\sigma \circ \rho_2) \leq \mathsf{dom}(\Gamma(\mathtt{x}_1)(\sigma \circ \rho_1))$. As $(\sigma \circ \rho_2) \in \Sigma_2$ and $(\sigma \circ \rho_1) \in \Sigma_1$, we deduce $\Gamma(\mathtt{x}_1)\Sigma_1 \leq \mathbb{0} \to \mathbb{1}$ and $\Gamma(\mathtt{x}_2)\Sigma_2 \leq \mathsf{dom}(\Gamma(\mathtt{x}_1)\Sigma_1)$ (we use the fact that $\mathsf{dom}(.)$ is monotonically non-increasing).

The other cases are similar or straightforward. □

Now, we propose a sketch of proof justifying that the deduction rules for the reconstruction system define a terminating algorithm. The idea of this proof is similar to the proof of termination of the *Kirby-Paris hydra* game[12]: we can associate an ordinal number weight to each node, and this weight can only decrease as the game (or derivation) advances. Intuitively, this non-negative weight represents the advancement of the game (or derivation). Though sub-trees can sometimes be duplicated, their weight is always lowered before being duplicated, resulting in a lower weight overall.

For any type environment Γ, canonical form or atom η, and intermediate annotation $\mathcal{H}$ compatible with the structure of η, the weight $\mathtt{w}(\Gamma, \eta, \mathcal{H})$ is the ordinal number defined as follows:

$$\begin{aligned}
\mathtt{w}(\Gamma, \eta, \mathtt{typ}) &= 1 \\
\mathtt{w}(\Gamma, \eta, \mathtt{untyp}) &= 1 \\
\mathtt{w}(\Gamma, \eta, \bigwedge(S_1, S_2)) &= \textstyle\sum\{\mathtt{w}(\Gamma, \eta, \mathcal{H}) \mid \mathcal{H} \in S_1\}
\end{aligned}$$

$$\begin{aligned}
\mathtt{w}(\Gamma, c, \mathtt{infer}) &= \omega & \qquad \mathtt{w}(\Gamma, \lambda x.\kappa, \mathtt{infer}) &= \omega^{\mathtt{w}(\Gamma,\kappa,\mathtt{infer})} \\
\mathtt{w}(\Gamma, x, \mathtt{infer}) &= \omega & \qquad \mathtt{w}(\Gamma, \lambda x.\kappa, \lambda(\mathbf{u}, \mathcal{K})) &= \omega^{\mathtt{w}(\Gamma,\kappa,\mathcal{K})}
\end{aligned}$$

$$\begin{aligned}
\mathtt{w}(\Gamma, \pi_i\mathtt{x}, \mathtt{infer}) &= \omega & &\text{if } \mathtt{x} \in \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, \pi_i\mathtt{x}, \mathtt{infer}) &= \omega^2 & &\text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathtt{w}(\Gamma, \mathtt{x}_1\mathtt{x}_2, \mathtt{infer}) &= \omega & &\text{if } \{\mathtt{x}_1, \mathtt{x}_2\} \subseteq \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, \mathtt{x}_1\mathtt{x}_2, \mathtt{infer}) &= \omega^2 & &\text{otherwise, if } \mathtt{x}_1 \in \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, \mathtt{x}_1\mathtt{x}_2, \mathtt{infer}) &= \omega^2 & &\text{otherwise, if } \mathtt{x}_2 \in \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, \mathtt{x}_1\mathtt{x}_2, \mathtt{infer}) &= \omega^3 & &\text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathtt{w}(\Gamma, (\mathtt{x}_1, \mathtt{x}_2), \mathtt{infer}) &= \omega & &\text{if } \{\mathtt{x}_1, \mathtt{x}_2\} \subseteq \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, (\mathtt{x}_1, \mathtt{x}_2), \mathtt{infer}) &= \omega^2 & &\text{otherwise, if } \mathtt{x}_1 \in \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, (\mathtt{x}_1, \mathtt{x}_2), \mathtt{infer}) &= \omega^2 & &\text{otherwise, if } \mathtt{x}_2 \in \mathsf{dom}(\Gamma) \\
\mathtt{w}(\Gamma, (\mathtt{x}_1, \mathtt{x}_2), \mathtt{infer}) &= \omega^3 & &\text{otherwise}
\end{aligned}$$

[12] https://en.wikipedia.org/wiki/Hydra_game

$$
\begin{array}{rcll}
\mathtt{w}(\Gamma, (\mathtt{x}_0{\in}\tau) \mathrel{?} \mathtt{x}_1 : \mathtt{x}_2, \in_i) & = & \omega & \\
\mathtt{w}(\Gamma, (\mathtt{x}_0{\in}\tau) \mathrel{?} \mathtt{x}_1 : \mathtt{x}_2, \mathtt{infer}) & = & \omega^2 & \text{if } \Gamma(\mathtt{x}_0) \leq \tau \\
\mathtt{w}(\Gamma, (\mathtt{x}_0{\in}\tau) \mathrel{?} \mathtt{x}_1 : \mathtt{x}_2, \mathtt{infer}) & = & \omega^2 & \text{otherwise, if } \Gamma(\mathtt{x}_0) \leq \neg\tau \\
\mathtt{w}(\Gamma, (\mathtt{x}_0{\in}\tau) \mathrel{?} \mathtt{x}_1 : \mathtt{x}_2, \mathtt{infer}) & = & \omega^3 & \text{otherwise, if } \mathtt{x}_0 \in \mathrm{dom}(\Gamma) \\
\mathtt{w}(\Gamma, (\mathtt{x}_0{\in}\tau) \mathrel{?} \mathtt{x}_1 : \mathtt{x}_2, \mathtt{infer}) & = & \omega^4 & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rl}
\mathtt{w}(\Gamma, \mathtt{bind}\,\mathtt{x} = a \,\mathtt{in}\, \kappa, \mathtt{skip}\ (\mathcal{K})) = & \omega^{\mathtt{w}(\Gamma,\kappa,\mathcal{K})} \\
\mathtt{w}(\Gamma, \mathtt{bind}\,\mathtt{x} = a \,\mathtt{in}\, \kappa, \mathtt{keep}\ (\mathcal{A}, \mathcal{S}_1, \mathcal{S}_2)) = & \omega^{\alpha} \\
& \text{with } \alpha = \sum\{\mathtt{w}((\Gamma, \mathtt{x} : \mathbf{u}), \kappa, \mathcal{K}) \mid (\mathbf{u}, \mathcal{K}) \in \mathcal{S}_1\} \\
\mathtt{w}(\Gamma, \mathtt{bind}\,\mathtt{x} = a \,\mathtt{in}\, \kappa, \mathtt{propagate}\ (\mathcal{A}, \mathbb{F}, \mathcal{S}_1, \mathcal{S}_2)) = & \omega^{\alpha+|\mathbb{F}|} \\
& \text{with } \alpha = \sum\{\mathtt{w}((\Gamma, \mathtt{x} : \mathbf{u}), \kappa, \mathcal{K}) \mid (\mathbf{u}, \mathcal{K}) \in \mathcal{S}_1\} \\
\mathtt{w}(\Gamma, \mathtt{bind}\,\mathtt{x} = a \,\mathtt{in}\, \kappa, \mathtt{try\text{-}keep}\ (\mathcal{A}, \mathcal{K}_1, \mathcal{K}_2)) = & \omega^{\alpha} \\
& \text{with } \alpha = \sum\{\mathtt{w}(\Gamma, a, \mathcal{A}),\ \mathtt{w}((\Gamma, \mathtt{x} : \mathbb{1}), \kappa, \mathcal{K}_1),\ \mathtt{w}(\Gamma, \kappa, \mathcal{K}_2)\} \\
\mathtt{w}(\Gamma, \mathtt{bind}\,\mathtt{x} = a \,\mathtt{in}\, \kappa, \mathtt{try\text{-}skip}\ (\mathcal{K})) = & \omega^{\alpha} \\
& \text{with } \alpha = \sum\{\mathtt{w}(\Gamma, a, \mathtt{infer}),\ \mathtt{w}(\Gamma, \kappa, \mathcal{K})\}
\end{array}
$$

$$
\begin{array}{rcll}
\mathtt{w}(\Gamma, \mathtt{x}, \mathtt{infer}) & = & \omega & \text{if } \mathtt{x} \in \mathrm{dom}(\Gamma) \\
\mathtt{w}(\Gamma, \mathtt{x}, \mathtt{infer}) & = & \omega^2 & \text{otherwise}
\end{array}
$$

where, for any multiset $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, $\sum\{\alpha_1, \alpha_2, \ldots, \alpha_n\} \stackrel{\text{def}}{=} \alpha_1 + \alpha_2 + \cdots + \alpha_n$ with $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_n$.

Then, we define a weight $\mathtt{w}(\Gamma, \eta, \mathbb{R})$ for any type environment Γ, canonical form or atom η, and result $\mathbb{R}$ compatible with the structure of η:

$$
\begin{array}{rcl}
\mathtt{w}(\Gamma, \eta, \mathtt{Ok}(\mathcal{H})) & = & 1 \\
\mathtt{w}(\Gamma, \eta, \mathtt{Fail}) & = & 1 \\
\mathtt{w}(\Gamma, \eta, \mathtt{Split}(\Gamma', \mathcal{H}_1, \mathcal{H}_2)) & = & \sum\ \{\mathtt{w}(\Gamma \wedge \Gamma', \eta, \mathcal{H}_1)\} \cup \{\mathtt{w}((\Gamma, \mathtt{x} : \neg\mathbf{u}), \eta, \mathcal{H}_2) \mid (\mathtt{x} : \mathbf{u}) \in \Gamma'\} \\
\mathtt{w}(\Gamma, \eta, \mathtt{Subst}(\{\psi_i\}_{i\in I}, \mathcal{H}_1, \mathcal{H}_2)) & = & \sum\ \{\mathtt{w}(\Gamma, \eta, \mathcal{H}_2)\} \cup \{\mathtt{w}(\Gamma\psi_i, \eta, \mathcal{H}_1\psi_i) \mid i \in I\} \\
\mathtt{w}(\Gamma, \eta, \mathtt{Var}\ (\mathtt{x}, \mathcal{H}_1, \mathcal{H}_2)) & = & \sum\{\mathtt{w}((\Gamma, \mathtt{x} : \mathbb{1}), \eta, \mathcal{H}_1),\ \mathtt{w}(\Gamma, \eta, \mathcal{H}_2)\}
\end{array}
$$

Lemma I.69. *For any $\Gamma, \eta, \mathcal{H}$, and Γ' such that $\Gamma' \leq \Gamma$, we have $\mathtt{w}(\Gamma', \eta, \mathcal{H}) \leq \mathtt{w}(\Gamma, \eta, \mathcal{H})$.*

Proof. Straightforward induction. □

Lemma I.70. *For any $\Gamma, \eta, \mathcal{H}$, and ψ, we have $\mathtt{w}(\Gamma\psi, \eta, \mathcal{H}\psi) \leq \mathtt{w}(\Gamma, \eta, \mathcal{H})$.*

Proof. Straightforward induction (we recall that test types τ do not contain type variables). □

Lemma I.71. *If $\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$ or $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$, then $\mathtt{w}(\Gamma, \eta, \mathcal{H}) \gtrsim \mathtt{w}(\Gamma, \eta, \mathbb{R})$.*

Proof. Structural induction on the derivation of $\Gamma \vdash_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$ or $\Gamma \vdash^*_{\mathcal{R}} \langle \eta \mid \mathcal{H} \rangle \Rightarrow \mathbb{R}$. □

Theorem I.72 (Termination). *The deduction rules $\vdash^*_{\mathcal{R}}$ and $\vdash_{\mathcal{R}}$ define a terminating algorithm: it can either fail (if no rule applies at some point) or return a result $\mathbb{R}$.*

Proof. There can only be finitely many [Iterate$_1$] and [Iterate$_2$] nodes applied on a given canonical form or atom, otherwise, according to the previous lemmas, we could extract from them an infinite decreasing chain of ordinal numbers. □

Received 2023-07-11; accepted 2023-11-07