Polymorphic Functions with Set-Theoretic Types. Part 2: Local Type Inference and Type Reconstruction

Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Pietro Abate

Polymorphic Functions with Set-Theoretic Types

Part 2: Local Type Inference and Type Reconstruction

Giuseppe Castagna[1] Kim Nguyễn[2] Zhiwu Xu[1,3] Pietro Abate[1]

[1]CNRS, PPS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France [2]LRI, Université Paris-Sud, Orsay, France
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

This article is the second part of a two articles series about a calculus with higher-order polymorphic functions, recursive types with arrow and product type constructors and set-theoretic type connectives (union, intersection, and negation). In the first part, presented in a companion paper, we defined and studied the syntax, semantics, and evaluation of the explicitly-typed version of the calculus, in which type instantiation is driven by explicit instantiation annotations. In this second part we present a local type inference system that allows the programmer to omit explicit instantiation annotations, and a type reconstruction system that allows the programmer to omit explicit type annotations.

The work presented in the two articles provides the theoretical foundations and technical machinery needed to design and implement higher-order polymorphic functional languages for semi-structured data.

1. Introduction

Many recent XML processing languages, such as XDuce, ℂDuce, XQuery, OcamlDuce, XHaskell, XAct, are statically-typed functional languages. However, none of them provides full-fledged parametric polymorphism even though this feature has been repeatedly requested in different standardization groups. A major stumbling block to such an extension —*ie*, the definition of a subtyping relation for regular tree types with type variables— has been recently lifted by Castagna and Xu [4]. In this work we present the next logical step of that research, that is, the definition of a higher-order functional language that takes full advantage of Castagna and Xu's system. To that end we define and study a calculus with higher-order polymorphic functions and recursive types with union, intersection, and negation connectives. The approach is thus general and, as such, goes well beyond the simple application to XML processing languages. As a matter of facts, our motivating example developed all along this paper does not involve XML, but looks like a rather classic display of functional programming specimens:

```
map :: (α -> β) -> [α] -> [β]
map f l = case l of
          | [] -> []
          | (x : xs) -> (f x : map f xs)

even :: (Int -> Bool) ∧ ((α\Int) -> (α\Int))
even x = case x of
         | Int -> (x `mod` 2) == 0
         |   _ -> x
```

The first function is the classic `map` function defined in Haskell (we just used Greek letters to denote type variables). The second would be an Haskell function were it not for two oddities: its type contains type connectives (type intersection "∧" and type difference "\"); and the pattern in the `case` expression is a type, meaning that it matches all values returned by the matched expression that have that type. So what does the `even` function do? It checks whether its argument is an integer; if it is so it returns whether the integer is even or not, otherwise it returns its argument as it received it (although `even` may be considered as bad programming, it is a perfect minimal example to illustrate all the aspects of our system).

The goal of this work is to define a calculus and a type system that can pass three tests. The first test is that it can define the two functions above. The second, harder, test is that the type system must be able to verify that these functions have the types declared in their signatures. That `map` has the declared type will come as no surprise (in practice, we actually want the system to infer this type even in the absence of a signature given by the programmer: see Section 7). That `even` was given an intersection type means that it must have all the types that form the intersection. So it must be a function that when applied to an integer it returns a Boolean and that when applied to an argument of a type that does not contain any integer, it returns a result of the same type. In other terms, `even` is a polymorphic (dynamically bounded) overloaded function.

The third test, the hardest one, is that the type system must be able to *infer* the type of the partial application of `map` to `even`, and the inferred type must be equivalent to the following one[1]

```
map even :: ([Int] -> [Bool])  ∧
            ([α\Int] -> [α\Int]) ∧                    (1)
            ([α∨Int] -> [(α\Int)∨Bool])
```

since `map even` returns a function that when applied to a list of integers it returns a list of Booleans; when applied to a list that does not contain any integer then it returns a list of the same type (actually, the same list); and when it is applied to a list that may contain some integers (*eg*, a list of reals), then it returns a list of the same type, without the integers but with some Booleans instead (in the case of reals, a list with Booleans and reals that are not integers).

Technically speaking, the definition of such a calculus and its type system is difficult for two distinct reasons. First, for the reasons we explain in the next section, it demands to define an explicitly typed λ-calculus with intersection types, a task that, despite many attempts in the last 20 years, still lacked a satisfactory definition. Second, even if working with an explicitly typed setting may seem simpler, the system needs to solve "local type inference"[2], namely, the problem of checking whether the types of a function and of its argument can be made compatible and, if so, of inferring the type of their result as we did for (1). The presentation of our work is split in two parts, accordingly: in the first part (the companion paper [3]) we showed how to solve the problem of defining an explicitly-typed λ-calculus with intersection types and how to efficiently evaluate it. In this paper, the second part of our work, we show how to solve the problem of "local type inference" for a calculus with intersection types. In particular, we show how local type inference for our system reduces to the problem of finding two *sets* of type substi-

[1] This type is redundant since the first type of the intersection is an instance (*eg*, for α=`Int`) of the third. We included it for the sake of the presentation.

[2] There are different definitions for *local type inference*. Here we use it with the meaning of finding the type of an expression in which not all type annotations are specified. This is the acceptation used in Scala where type parameters for polymorphic methods can be omitted. In our specific problem, we will omit —and, thus, infer— the annotations that specify how function and argument types can be made compatible.

tutions $\{\sigma_i \mid i{\in}I\}$ and $\{\sigma'_j \mid j{\in}J\}$ such that for two given types s and t the relation $\bigwedge_{i\in I} s\sigma_i \leq \bigwedge_{j\in J} t\sigma'_j$ holds. Therefore, this second part is mainly devoted to solving this problem. The paper is accompanied by an appendix available on-line which contains all detailed proofs and complete definitions.

Next section outlines the various problems we met and how they were solved. The reader acquainted with the first part can skip directly to Section 2.2.

2. Overview

The driver of this work is the definition an XML processing functional language with high-order polymorphic functions, that is, in particular, a polymorphic version of the language ℂDuce [2]. The essence of ℂDuce is a λ-calculus with pairs, explicitly-typed recursive functions, and a type-case expression. Its types can be recursively defined and include the arrow and product type *constructors* and the intersection, union, and negation type *connectives*.

In a nutshell, we want to define the static and dynamic semantics of the language whose types and expressions are defined as follows.

Definition 2.1 (Types)**.** *Types are the regular trees coinductively generated by the following productions:*

$$t ::= b \mid t \to t \mid t \wedge t \mid t \vee t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha \tag{2}$$

and such that every infinite branch contains infinitely many occurrences of type constructors. We use $\mathscr{T}$ to denote the set of all types.

In the definition, b ranges over basic types (*eg*, `Int`, `Bool`), α ranges over type variables, and $\mathbb{0}$ 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). It also ensures that the binary relation $\rhd \subseteq \mathscr{T}^2$ 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 on $\mathscr{T}$ that we will use without any further explicit reference to the relation. We use $\mathsf{var}(t)$ to denote the *set of type variables* occurring in a type t. A type t is said to be *ground* or *closed* if and only if $\mathsf{var}(t)$ is empty. The subtyping relation for these types was defined by Castagna and Xu [4]. For this work it suffices to consider that ground types are interpreted as sets of *values* (*ie*, either constants or λ-abstractions) that have that type and subtyping is set containment (a ground type s is a subtype of a ground 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 they return a result, then this result is of type t (*eg*, $\mathbb{0}{\to}\mathbb{1}$ is the set of all functions[3] and $\mathbb{1}{\to}\mathbb{0}$ is the set of functions that diverge on all their arguments). Type connectives (union, intersection, negation) are interpreted as the corresponding set-theoretic operators (*eg*, $s{\vee}t$ is the union of the values of the two types) and subtyping is set containment. For what concerns non-ground types (*ie*, types with variables occurring in them) all the reader needs to know for this work is that the subtyping relation of Castagna and Xu is preserved by substitution of the type variables. Namely, if $s \leq t$, then $s\sigma \leq t\sigma$ for every type-substitution σ (the converse does not hold in general, while it holds for *semantic* type-substitutions in convex models: see [4]). Two types are equivalent if they are subtype one of each other (type equivalence is denoted by $\simeq$). Finally, notice that in this system $s \leq t$ if and only if $s{\wedge}\neg t \leq \mathbb{0}$.

Definition 2.2 (Expressions)**.** *Expressions are the terms inductively generated by the following grammar*

$$e ::= c \mid x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e \tag{3}$$

and such that in every expression $e{\in}t\,?\,e_1 : e_2$ the type t is closed.

In the definition, c ranges over constants (*eg*, `true`, `false`, `1`, `2`, ...) which are values of basic types (we use b_c to denote the basic type of the constant c); x ranges over expression variables; $e{\in}t\,?\,e_1 : e_2$ denotes the type-case expression that evaluates either e_1 or to e_2 according to whether the value returned by e (if any) is of type t or not; $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ is a value of type $\wedge_{i\in I} s_i \to t_i$ that denotes the function of parameter x and body e. An expression has an intersection type, if it has all the types that compose the intersection. Therefore, intuitively, $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ is a well-typed value if for all $i{\in}I$ the hypothesis that x is of type s_i implies that the body e has type t_i, that is to say, it is well typed if $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ has type $s_i \to t_i$ for all $i \in I$.

The core of monomorphic ℂDuce, dubbed "Coreℂ Duce", has exactly the same types and expressions as the above with two single differences: (i) types do not contain type variables, that is, they are as in Definition 2.1 but where the grammar (2) does not have the last production for variables and (ii) it includes product types and recursive functions, which we omitted here for brevity since our results can be easily extended to them (as sketched in Section 5 and shown in the appendixes).

From a strictly practical viewpoint recursive types, products, and type connectives are used to encode regular tree types, which subsume existing XML schema/types while, for what concerns expressions, the type-case is an abstraction of ℂDuce pattern matching (this uses regular expression patterns on types to define powerful capture primitives for XML data). The reasons why in Coreℂ Duce (and in its polymorphic extension we study here) there is a type-case expressions and why λ-expressions are explicitly annotated by their intersection types are explained in details in the companion paper that presents the first part of this work [3] and we invite the reader to refer to it.

The novelty of this work with respect to Coreℂ Duce, thus, is to allow type variables to occur in the types that annotate λ-abstractions. It becomes thus possible to define the polymorphic identity function as $\lambda^{\alpha\to\alpha}x.x$, while the classic "auto-application" term is written as $\lambda^{((\alpha\to\beta)\wedge\alpha)\to\beta}x.xx$. The intended meaning of using a type variable, such as α, is that a (well-typed) λ-abstraction not only has the type specified in its label (and by subsumption all its super-types) but also all types obtained by instantiating the type variables occurring in its label. So $\lambda^{\alpha\to\alpha}x.x$ has not only type $\alpha \to \alpha$ but also, for instance, by subsumption the types $\mathbb{0}{\to}\mathbb{1}$ (the type of all functions, which is a super-type of $\alpha{\to}\alpha$) and $\neg$`Int` (the type of all non integer values), and by instantiation the types `Int`$\to$`Int`, `Bool`$\to$`Bool`, etc. The addition of type variables and instantiation makes the calculus a full-fledged intersection type system (see Section 3.5 in the first part of this work): for instance, by combining intersections, instantiation, and subtyping, it is possible to deduce that $\lambda^{\alpha\to\alpha}x.x$ has type $(\texttt{Int}{\to}\texttt{Int}) \wedge (\texttt{Bool}{\to}\texttt{Bool}) \wedge \neg\texttt{Int}$.

The first problem we have to solve then is to define an *explicitly-typed* λ-calculus with intersection types. We outline in four points why this problem is hard (it has been studied for over 20 years) and summarize the solution we proposed in the first part of this work.

1. Polymorphism needs instantiation: To apply the polymorphic identity $\lambda^{\alpha\to\alpha}x.x$ to, say, 42, we must use the particular instance of the identity obtained by applying the type substitution $\{{}^{\texttt{Int}}/_{\alpha}\}$ (denoting the replacement of every occurrence of α by `Int`). Thus the application $(\lambda^{\alpha\to\alpha}x.x)42$ implicitly stands for $(\lambda^{\alpha\to\alpha}x.x)\{{}^{\texttt{Int}}/_{\alpha}\}42$, that is, $(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)42$: we have virtually "relabeled" $\lambda^{\alpha\to\alpha}x.x$ as $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$ before applying it to 42.

2. Type-case needs explicit relabeling: because of the presence of type-case expressions, however, relabeling cannot remain just virtual but must be effectively performed. The label

[3] Actually, for every type t, all types of the form $\mathbb{0}{\to}t$ are equivalent and each of them denotes the set of all functions.

of a λ-abstraction determines its type: testing whether the result of the application $(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)e$ has type $\texttt{Int}\to\texttt{Int}$ should succeed for e=42 and fail for e=true. In other words, $((\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42)\in\texttt{Int}\to\texttt{Int}\,?\,0:1$ must reduce to $(\lambda^{\texttt{Int}\to\texttt{Int}}y.42)\in\texttt{Int}\to\texttt{Int}\,?\,0:1$ and thus to 0, while the expression $((\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\texttt{true})\in\texttt{Int}\to\texttt{Int}\,?\,0:1$ must reduce to $(\lambda^{\texttt{Bool}\to\texttt{Bool}}y.\texttt{true})\in\texttt{Int}\to\texttt{Int}\,?\,0:1$ and thus to 1.

3. Relabeling must be propagated to function bodies: consider the following "daffy" —though well-typed— definition of the identity function:[4]

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x) \tag{4}$$

If we want to apply this function to, say, 3, then we have first to relabel it by applying the substitution $\{\texttt{Int}/_\alpha\}$. However, applying the relabeling only to the outer "λ" does not suffice since the application of (4) to 3 reduces to $(\lambda^{\alpha\to\alpha}y.3)3$ which is not well-typed (it is not possible to deduce the type $\alpha\to\alpha$ for $\lambda^{\alpha\to\alpha}y.3$, which is the constant function that always returns 3) although it is the reductum of a well-typed application. So we have to relabel also the body by applying the same type-substitution $\{\texttt{Int}/_\alpha\}$ to the body. This yields a reductum $(\lambda^{\texttt{Int}\to\texttt{Int}}y.3)3$ which is well typed.

4. Relabeling the body is not always straightforward: Two different problems may conjugate to make relabeling of function bodies difficult: (i) it may be necessary to apply more than a single type-substitution and (ii) the relabeling of the body may depend on the dynamic type of the actual argument of the function (both problems are better known as —or are instances of— the problem of determining expansions for intersection type systems [6]). Let us next discuss each problem in detail.

First of all, recall that $\lambda^{\alpha\to\alpha}x.x$ has type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$. If we have a second function that expects arguments of this intersection type, then it is safe to pass the identity function as argument to it. Before, however, we have to relabel $\lambda^{\alpha\to\alpha}x.x$ into $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$, which corresponds to apply two distinct type-substitutions $\{\texttt{Int}/_\alpha\}$ and $\{\texttt{Bool}/_\alpha\}$ to the annotation of the λ-abstraction and replace it by the intersection of the two instances. This explains why in the first part of this work relabeling is performed by *sets* of type-substitutions —delimited by square brackets. The application of such a set (*eg*, in the previous example $[\{\texttt{Int}/_\alpha\}, \{\texttt{Bool}/_\alpha\}]$) to a type t returns the intersection of all types obtained by applying each substitution in set to t (*eg*, in the example $t\{\texttt{Int}/_\alpha\} \wedge t\{\texttt{Bool}/_\alpha\}$). The problem is how to relabel function bodies by *sets* of type-substitutions: while the naive solution consisting of propagating the substitution to the body works for a single substitution, for *sets* of substitutions it is unsound. This can be seen by trying to perform on the daffy identity function in (4) the same relabeling as we just did on the classic identity $\lambda^{\alpha\to\alpha}x.x$. In this case, the naive solution, consisting of propagating $[\{\texttt{Int}/_\alpha\}, \{\texttt{Bool}/_\alpha\}]$ also to the body, yields

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x) \tag{5}$$

which is not well typed. That this term is not well typed is clear if we try applying it to, say, 3: the application of a function of type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ to an Int should have type Int, but here it reduces to $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.3)3$, and there is no way to deduce the type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ for the constant function $\lambda y.3$. We can also directly verify that it is not well typed, by trying to type the function in (5). This corresponds to prove that under the hypothesis $x : \texttt{Int}$ the term $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$ has type Int, and that under the hypothesis $x : \texttt{Bool}$ this same term has type Bool. Both checks fail because, in both cases, $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is ill-typed (it neither has type $\texttt{Int}\to\texttt{Int}$ when x:Bool, nor has it type $\texttt{Bool}\to\texttt{Bool}$ when x:Int). This example shows that in order to ensure that relabeling yields well-typed terms, the relabeling of the body *must change* according to the type of the value the parameter x is bound to. More precisely, $(\lambda^{\alpha\to\alpha}y.x)$ should be relabeled as $\lambda^{\texttt{Int}\to\texttt{Int}}y.x$ when x is of type Int, and as $\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x$ when x is of type Bool.

2.1 Summary of Part 1

The last example suggests the key idea of the first part of this work:

> *The relabeling of the body of a function must change according to the type of the argument of the function.*

In the specific case, when we apply the daffy identity function to an integer we must relabel its body by the type-substitution $\{\texttt{Int}/_\alpha\}$, while the type-substitution $\{\texttt{Bool}/_\alpha\}$ must be used when the function argument is a Boolean value. To obtain this behavior, in the first part of this work we proposed and studied the "lazy" relabeling of function bodies. The relabeling is lazy since it delays the propagation of the set of substitutions to the function body until the precise type of the function argument is known. This is obtained by decorating λ-abstractions by (sets of) type-substitutions. For example, in order to pass our daffy identity function (4) to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ we first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_\alpha\},\{\texttt{Bool}/_\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x). \tag{6}$$

The annotation in the outer "λ" indicates that the function must be relabeled and, therefore, that we are using the particular instance whose type is the one in the "interface" (*ie*, $\alpha\to\alpha$) to which we apply the set of type-substitutions in the annotation. The relabeling will be actually propagated to the body of the function at the moment of the reduction, only if and when the function is applied (relabeling is thus lazy). However, the new annotation is statically used by the type system to check soundness.

Formally, this is obtained in Part 1 by adding explicit sets of type-substitutions (ranged over by $[\sigma_j]_{j\in J}$) to the grammar (3) of Definition 2.2. Sets of type-substitutions can be applied directly to expressions (to produce a particular expansion/instantiation of the type variables occurring in them) or, as in (6), annotate "λ's (to implement the lazy relabeling of the function body). We thus defined a calculus whose syntax is

$$e ::= c \mid x \mid ee \mid \lambda^{\wedge_{i\in I}s_i\to t_i}_{[\sigma_j]_{j\in J}}x.e \mid e{\in}t\,?\,e:e \mid e[\sigma_j]_{j\in J} \tag{7}$$

where types are those in Definition 2.1 and with the restriction that the type tested in type-case expressions is closed. We call this calculus and its expressions the *explicitly-typed* calculus and expressions, respectively, in order to differentiate it from the one of Definition 2.2 which does not have explicit type-substitutions and, therefore, is called the *implicitly-typed* calculus.

Henceforth, given a λ-abstraction $\lambda^{\wedge_{i\in I}s_i\to t_i}_{[\sigma_j]_{j\in J}}x.e$ we call the type $\bigwedge_{i\in I} s_i\to t_i$ the *interface* of the function and the set of type-substitutions $[\sigma_j]_{j\in J}$ the *decoration* of the function. We write $\lambda^{\wedge_{i\in I}t_i\to s_i}x.e$ for short when the decoration is a singleton containing just the empty substitution. We use v to range over *values*, that is, either constants or λ-abstractions. Let e be an expression: we use $\mathsf{fv}(e)$ and $\mathsf{bv}(e)$ respectively to denote the sets of *free expression variables* and *bound expression variables* of the expression e; we use $\mathsf{tv}(e)$ to denote the set of *type variables* occurring in e.

As customary, we assume bound expression variables to be pairwise distinct and distinct from any free expression variable occurring in the expressions under consideration. Polymorphic variables can be bound by interfaces, but also by decorations: for example, in

[4] By convention a type variable is introduced by the outermost λ in which it occurs and this λ implicitly binds all inner occurrences of the variable. For instance, all the α's in the term (4) are the same while in a term such as $(\lambda^{\alpha\to\alpha}x.x)(\lambda^{\alpha\to\alpha}x.x)$ the variables in the function are distinct from those in its argument and, thus, can be α-converted separately, as $(\lambda^{\gamma\to\gamma}x.x)(\lambda^{\delta\to\delta}x.x)$.

$$
\text{(ALG-CONST)}\quad \frac{}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} c : b_c}
\qquad
\text{(ALG-VAR)}\quad \frac{}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} x : \Gamma(x)}
$$

$$
\text{(ALG-INST)}\quad \frac{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e : t}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ \sigma_j \sharp\, \Delta
\qquad
\text{(ALG-APPL)}\quad \frac{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_1 : t \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_2 : s}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_1 e_2 : t \cdot s}\ \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathsf{dom}(t) \end{array}
$$

$$
\text{(ALG-ABSTR)}\quad \frac{\Delta \cup \Delta' \,;\, \Gamma, (x : t_i\sigma_j) \vdash_{\mathscr{A}} e@[\sigma_j] : s'_{ij}}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} (t_i\sigma_j \to s_i\sigma_j)}\ \begin{array}{l} \Delta' = \mathsf{var}(\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \\ s'_{ij} \leq s_i\sigma_j, \quad i\in I, \quad j\in J \end{array}
$$

$$
\text{(ALG-CASE-FST)}\quad \frac{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e : t' \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} (e\in t \,?\, e_1 : e_2) : s_1}\ t' \leq t
$$

$$
\text{(ALG-CASE-SND)}\quad \frac{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e : t' \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_2 : s_2}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} (e\in t \,?\, e_1 : e_2) : s_2}\ t' \leq \neg t
\qquad
\text{(ALG-CASE-BOTH)}\quad \frac{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} e : t' \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1 \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{A}} e_2 : s_2}{\Delta \,;\, \Gamma \vdash_{\mathscr{A}} (e\in t \,?\, e_1 : e_2) : s_1 \vee s_2}\ \begin{array}{l} t' \not\leq \neg t \\ t' \not\leq t \end{array}
$$

Figure 1. Typing algorithm

$\lambda^{\beta\to\beta}_{[\{\alpha/\beta\}]} x.(\lambda^{\alpha\to\alpha} y.y)x$, the α occurring in the interface of the inner abstraction is "bound" by the decoration $[\{\alpha/\beta\}]$, and the whole expression is α-equivalent to $(\lambda^{\beta\to\beta}_{[\{\gamma/\beta\}]} x.(\lambda^{\gamma\to\gamma} y.y)x)$. If a type variable is bound by an outer abstraction, it cannot be instantiated; such a variable is called *monomorphic*. We assume that polymorphic variables are pairwise distinct and distinct from any monomorphic variable in the expressions under consideration. In particular, when substituting a value v for a variable x in an expression e, we suppose the polymorphic type variables of e to be distinct from the monomorphic and polymorphic type variables of v thus avoiding unwanted capture.

Both static and dynamic semantics for the explicitly-typed expressions in (7) are defined in terms of a *relabeling* operation "@" which takes an expression e and a set of type-substitutions $[\sigma_j]_{j\in J}$ and pushes $[\sigma_j]_{j\in J}$ down to all outermost λ-abstractions occurring in e (and collects and composes with the sets of type-substitutions it meets). Precisely, $e@[\sigma_j]_{j\in J}$ is defined for λ-abstractions and (inductively) for applications of type-substitutions as:

$$
\begin{array}{rcl}
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J} \circ [\sigma_k]_{k\in K}} x.e \\
(e[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I})
\end{array}
$$

(where $\circ$ denotes the pairwise composition of all substitutions of the two sets). It erases the set of type-substitutions when e is either a variable or a constant, and it is homomorphically applied on the remaining expressions (see Part 1 for comprehensive definitions). The dynamic semantics is given by the following notions of reduction (where v is a *value*), applied by a leftmost-outermost strategy:

$$
e[\sigma_j]_{j\in J} \ \rightsquigarrow\ e@[\sigma_j]_{j\in J} \tag{8}
$$

$$
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e)v \ \rightsquigarrow\ (e@[\sigma_j]_{j\in P})\{v/x\} \tag{9}
$$

$$
v\in t \,?\, e_1 : e_2 \ \rightsquigarrow\ \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases} \tag{10}
$$

where in (9) we have $P \stackrel{\text{def}}{=} \{j\in J \mid \exists i\in I, \vdash v : t_i\sigma_j\}$.

The first rule (8) performs relabeling, that is, it propagates the sets of type-substitutions down into the decorations of the outermost λ-abstractions. The second rule (9) states the semantics of applications: this is standard call-by-value β-reduction, with the difference that the substitution of the argument for the parameter is performed on the relabeled body of the function. Notice that relabeling depends on the type of the argument and keeps only those type-substitutions that make the type of the argument v match (at least one of) the input types defined in the interface of the function (*ie*, the set P which contains all substitutions σ_j such that the argument v has type $t_i\sigma_j$ for some i in I: the type system statically ensures that P will never be empty). For instance, take the daffy identity function (4), instantiate it as in (6) by both `Int` and `Bool`, and apply it to `42` —*ie*, $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)\mathtt{42}$—, then it reduces to $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} y.\mathtt{42})\mathtt{42}$, (which is observationally equivalent to $(\lambda^{\mathtt{Int}\to\mathtt{Int}} y.\mathtt{42})\mathtt{42}$) since the reduction discards the $\{\mathtt{Bool}/\alpha\}$ substitution. Finally, the third rule (10) checks whether the value returned by the expression in the type-case matches the specified type and selects the branch accordingly.

The static semantics is given by the rules in Figure 1 which form an algorithmic system (as stressed by $\vdash_{\mathscr{A}}$ and the names of the rules): in every case at most one rule applies, either because of the syntax of the term or because of mutually exclusive side conditions. We invite the reader to consult Part 1 for more details (there the reader will also find a non-algorithmic —and far more readable— system defined in terms of subsumption). Here we just comment the rules (ALG-ABSTR), (ALG-INST), and (ALG-APP). First of all notice Δ in judgments. This is the set of *monomorphic type variables*, that is, the variables that occur in the type of some outer λ-abstraction and, as such, cannot be instantiated; it must contain all the type variables occurring in Γ. Rule (ALG-ABSTR) checks that $\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e$ has the type declared by (the combination of) its interface and its decoration, that is, $\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$. To do that it first adds all the variables occurring in this type to the set Δ, (in the function body these variables are monomorphic). Then, it checks that for every possible input type —*ie*, for every possible combination of t_i and σ_j— the function body e relabeled with the single type-substitution σ_j under consideration (*ie*, $e@[\sigma_j]$), has (a subtype of) the corresponding output type.

Rule (ALG-INST) infers for $e[\sigma_j]_{j\in J}$ the type obtained by applying the set of type-substitutions to the type of e, provided that the type-substitutions do not instantiate monomorphic variables (*ie*, for all $j\in J$, $\mathsf{var}(\sigma_j) \cap \Delta = \varnothing$, noted as $\sigma_j \sharp\, \Delta$).

Rule (ALG-APPL) for applications checks that the type t of the function is a functional type (*ie*, $t \leq \mathbb{0}\to\mathbb{1}$). Then it checks that the type of the argument is a subtype of the domain of t (denoted by $\mathsf{dom}(t)$). Finally, it infers for the application the type $t \cdot s \stackrel{\text{def}}{=} \min\{u \mid t \leq s\to u\}$, that is, the smallest result type that can be obtained by subsuming t to an arrow type with domain s.[5] Even if $t \leq \mathbb{0}\to\mathbb{1}$, in general, t does not have the form of an arrow type (it could also be a union or an intersection or a negation of types) and the definition of $\mathsf{dom}(t)$ is not immediate. Formally, if $t \leq \mathbb{0}\to\mathbb{1}$, then $t \simeq \bigvee_{i\in I}(\bigwedge_{p\in P_i}(s_p\to t_p) \wedge \bigwedge_{n\in N_i} \neg(s_n\to t_n) \wedge \bigwedge_{q\in Q_i} \alpha_q \wedge \bigwedge_{r\in R_i} \neg\beta_r)$ where all P_i's are not empty (see Castagna and Xu [4]), and, for such a t, the domain is defined as $\mathsf{dom}(t) \stackrel{\text{def}}{=} \bigwedge_{i\in I} \bigvee_{p\in P_i} s_p$ (see Part 1 [3]).

The type system is sound (it satisfies both subject reduction and progress), it subsumes existing intersection type systems, and type inference is decidable. Furthermore the calculus can be compiled into an intermediate language which executes relabeling only by need and, thus, can be efficiently evaluated (again, see Part 1 *ibid.*).

Before proceeding we stress again that in this calculus type-substitutions and, thus, instantiation are *explicit*: $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]$ has type $\mathtt{Int}\to\mathtt{Int}$, but $(\lambda^{\alpha\to\alpha} x.x)$ does not (contrary to ML, a semantic subtyping relation $\leq$ does not account for instantiation).

[5] For every type t such that $t \leq \mathbb{0}\to\mathbb{1}$ and type s such that $s \leq \mathsf{dom}(t)$, the type $t \cdot s$ exists and can be effectively computed.

2.2 Overview of Part 2

Recall that we want the programmer to use the implicitly-typed expressions of grammar (3), and not those of grammar (7) which would require the programmer to write explicit type-substitutions. Therefore in Section 3 we define a local type inference system that, given an implicitly-typed expression produced by the grammar (3), checks whether and where some sets of type-substitutions can be inserted in this expression so as to make it a well-typed explicitly-typed expression of grammar (7). Thus, our local type inference consists of a type-substitution reconstruction system, insofar as it has to reconstruct the sets of type-substitutions that make an expression of grammar (3) a well-typed expression of grammar (7). In order to avoid ambiguity we reserve the word "reconstruction" for the problem of reconstructing *type* annotations (in particular, function interfaces) and speak of *inference of type-substitutions* for this problem. In particular, we show that this problem can be reduced to the problem of deciding whether for two types s and t there exist two sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that $s[\sigma_i]_{i\in I} \leq t[\sigma'_j]_{j\in J}$. We prove that when the cardinalities of I and J are given, the problem above is decidable and reduces to finding all substitutions σ such that $s'\sigma \leq t'\sigma$ for two given types s' and t' (we dub this problem the *tallying problem*). We show how to produce a sound and complete set of solutions for the latter problem. This is done by generating *sets* of constraint-sets that are then *normalized*, *merged*, and *solved*. The solution of the tallying problem immediately yields a semi-decision procedure (that tries all the cardinalities for I, J) for the local type inference system. Henceforth, to enhance readability, we will systematically use the metavariable "a" to denote expressions of the implicitly-typed calculus (*ie*, those of grammar (3)) and reserve the metavariable "e" for expressions of the <u>e</u>xplicitly-typed calculus (*ie*, those of grammar (7)).

Finally in Section 4 we show that the theory and algorithms developed in Section 3 can be reused to do ML-like type reconstruction, that is, to infer the interface of an unannotated λ-expression, in a pure λ-calculus with type-case.

Let us summarize all these passages on the motivating example of the introduction. First, note that the language defined in (7) passes our first test since the `even` function can be defined as

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\backslash\texttt{Int}\to\alpha\backslash\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x \qquad (11)$$

(where $s\backslash t$ is syntactic sugar for $s\wedge\neg t$) while —with the products and recursive function definitions of the Appendix— `map` is

$$\mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\ f = \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell)) \qquad (12)$$

where the type `nil` tested in the type case denotes the singleton type that contains just the constant `nil`, and $[\alpha]$ denotes the regular type that is the (least) solution of $X = (\alpha, X) \vee \texttt{nil}$.

If we feed these two expressions to the type-checker (the rules in Figure 1 suffice since no local type inference is needed to type these two functions) it confirms that both are well typed and have the types declared in their interfaces. To apply (the expression (12) defining) `map` to (the expression (11) defining) `even` we need to instantiate `map`, that is, to perform local type inference. The inference system of Section 3 infers the following set of type-substitutions $[\{(\alpha\backslash\texttt{Int})/_\alpha, (\alpha\backslash\texttt{Int})/_\beta\}, \{\alpha\vee\texttt{Int}/_\alpha, (\alpha\backslash\texttt{Int})\vee\texttt{Bool}/_\beta\}]$ and textually inserts it between the two terms (so that the type substitutions apply to the type variables of `map`) yielding a typing equivalent to the one in (1). The expression with the inserted set of type-substitutions is compiled into the intermediate language defined in Section 5 of the companion paper and executed as efficiently as if it were a monomorphic expression. Finally, in Section 4 we show that if the programmer had omitted the type declaration for `map` —*ie*, `map` `::` $(\alpha$ `->` $\beta)$ `->` $[\alpha]$ `->` $[\beta]$—, then it is possible to reuse the algorithms developed in Section 3 to reconstruct for `map` a type slightly more precise than the one above.

Contributions: The overall contribution of this work is the definition of a statically-typed calculus with polymorphic higher-order functions in a type system with recursive types and union, intersection, and negation type connectives and the definition of an efficient evaluation model. The contributions of this second part are:

– the definition of an algorithm that for any pair of polymorphic regular tree types t_1 and t_2 produces a sound and complete set of solutions to the problem of deciding whether there exists a substitution σ such that $t_1\sigma \leq t_2\sigma$. This is obtained by using the set-theoretic interpretation of types to reduce the problem to a unification problem on regular tree types.

– the definition of a type-substitution inference system sound and complete w.r.t. the type system of the explicitly-typed calculus.

– the definition of an algorithm for local type inference for the calculus. The algorithm yields a semi-decision procedure for the typeability of a λ-calculus with intersection and recursive types and with explicitly-typed λ-abstractions.

– the demonstration that the machinery developed for local type inference can be reused to perform type reconstruction.

3. Inference of type-substitutions

Since we want the programmer to program in the implicitly-typed calculus (3), then it is the task of the type-substitution inference system to check whether it is possible to insert some type-substitutions in appropriate places of the expression written by the programmer so that the resulting expression is a well-typed explicitly-typed expression of grammar (7). To define the type-substitution inference system we proceed in two steps.

1. The first step consists in defining a deduction system that checks whether and where it is possible to insert sets of type-substitutions into an implicitly-typed expression produced by the grammar (3) to make it a well-typed explicitly-typed expression of grammar (7). There will be a single exception: it will not try to insert type-substitutions into decorations, since it assumes that all λ-abstractions initially have empty decorations. There is no technical problem to infer also type-substitutions in decorations. Not doing so is just a design choice suggested by common sense so as to match the programmer's intuition: if we write an expression such as $\lambda^{\alpha\to\alpha}x.\mathtt{3}$ we want to infer that it is ill-typed; but if we allowed the system to infer type-substitutions for decorations, then the expression could be typed by inserting a decoration as in $\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_\alpha\}]}x.\mathtt{3}$.
   The set of places where the insertion of sets of type-substitutions must be tried is precisely given by the algorithm defined in Figure 1 (they correspond to the places where a subtyping relation is checked). This algorithm is used to define a deduction system that infers the type of the "implicitly-typed" (*ie*, without explicit type-substitutions) expressions: if a type is deduced for some expression, then there exists an explicitly-typed version of the expression that has that same type, and vice-versa.
2. The deduction system that will be given to solve the previous step will be syntax directed but will not yield an algorithm, yet, because it will use some operations that are not effective. In particular, these operations will require to solve the problem of whether there exist two sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $s[\sigma_i]_{i\in I} \leq t[\sigma_j]_{j\in J}$. If the cardinalities of I and J are known, then this problem can be reduced to what we dub the *tallying problem*: given two types s and t we say that s *tallies with* t if there exists a type-substitution σ such as $s\sigma \leq t\sigma$. We show how to decide the tallying problem and devise a semi-decision procedure for the more general problem with sets of type-substitutions which essentially tries all possible cardinalities of the two sets. We conjecture decidability also for this second problem though we are not able to prove it, yet.

(Inf-abstr)
$$\frac{\Delta\cup\Delta' \,;\, \Gamma, x : t_i \vdash_{\mathscr{I}} a : s_i'}{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i\to s_i} x.a : \bigwedge_{i\in I}(t_i \to s_i)} \quad \begin{array}{l}\Delta' = \mathsf{var}(\wedge_{i\in I} t_i\to s_i)\\ s_i' \sqsubseteq_{\Delta\cup\Delta'} s_i, \quad i\in I\end{array}$$

(Inf-case-both)
$$\frac{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_1 : s_1 \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_2 : s_2}{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} (a\in t\,?\,a_1 : a_2) : s_1 \vee s_2} \quad \begin{array}{l} t' \not\sqsubseteq_{\Delta} \neg t\\ t' \not\sqsubseteq_{\Delta} t\end{array}$$

(Inf-case-fst)
$$\frac{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_1 : s_1}{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} (a\in t\,?\,a_1 : a_2) : s_1}\, t' \sqsubseteq_{\Delta} t$$

(Inf-case-snd)
$$\frac{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_2 : s_2}{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} (a\in t\,?\,a_1 : a_2) : s_2}\, t' \sqsubseteq_{\Delta} \neg t$$

(Inf-appl)
$$\frac{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_1 : t \qquad \Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_2 : s}{\Delta \,;\, \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u} \quad u\in(t \bullet_{\Delta} s)$$

Figure 2. Inference system for type-substitutions

Each of these steps is developed in one of the following subsections.

3.1 Type substitution assignment

In this section we define an inference system for the implicitly-typed calculus of Definition 2.2. The system will be sound and complete with respect to explicitly-typed one modulo the single restriction we already mentioned, namely, we will consider only expressions in the explicitly-typed calculus in which all decorations are the singleton set that contains only the empty type-substitution. Recall that the reason of this restriction is common sense rather than technical. If the programmer specifies some interface for a function it seems reasonable to think that she wants the system to check whether the function conforms the interface rather than knowing whether there exists a type substitution that makes it conforming. In other words, if the programmer specified the signature `map::(`α`->`β`)->`γ, we expect the system to answer that the definition of `map` does not conform this signature, rather than it conforms the signature by substituting `[`α`]->[`β`]` for γ (alternatively, we must omit the signature altogether and let the system infer it: see Section 4).

We have to define a system that guesses where sets of type-substitutions must be inserted so that an implicitly-typed expression is transformed into a well-typed explicitly-typed expression in the system of Figure 1. The general role of type-substitutions is to make the type of some expression satisfy some subtyping constraints. Examples of this are the type of the body of a function which must match the result type declared in the interface, or the type of the argument of a function which must be a subtype of the domain of the function. Actually *all* the cases in which subtyping constraints must be satisfied are enumerated in Figure 1: they coincide with the subtyping relation checks occurring in the rules. Figure 1 is our Ariadne's thread through the definition of the type-substitution inference system: the rule (Alg-inst) must be removed and wherever the typing algorithm in Figure 1 checks whether for some types s and t the relation $s \leq t$ holds, then the type-substitution inference system must check whether there exists a set of type-substitutions $[\sigma_i]_{i\in I}$ for the polymorphic variables (*ie*, those not in Δ) that makes $s[\sigma_i]_{i\in I} \leq t$ hold. The reader may wonder why we apply the type-substitution only on the smaller type and not on both types. The reason can be understood by looking at the rules in Figure 1 and see that whenever a subtyping relation is specified, the larger type cannot be instantiated: either because it is a ground type (rules (Alg-case-*)) or because it is a type in an interface and inferring a type-substitution for it would correspond to inferring a type-substitution in a decoration (rule (Alg-abstr)). The only exception to this is the rule (Alg-appl) for application, but for it we will introduce a specific operator.

The essence of the type-substitutions inference system is all there. Things get slightly more complicated in the rule for application since the algorithm must find two sets of substitutions (one for the function and another for the argument) and the minimum of a set (to compute the type operator "·"). In order to ease the presentation it is handy to introduce a family of preorders $\sqsubseteq_\Delta$ that combine subtyping and instantiation:

Definition 3.1. *Let s and t be two types, Δ a set of type variables, and $[\sigma_i]_{i\in I}$ a set of type-substitutions. We define:*

$$[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t \quad \stackrel{\text{def}}{\iff} \quad \bigwedge_{i\in I} s\sigma_i \leq t \text{ and } \forall i{\in}I.\ \sigma_i \sharp \Delta$$
$$s \sqsubseteq_\Delta t \quad \stackrel{\text{def}}{\iff} \quad \exists[\sigma_i]_{i\in I} \text{ such that } [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t$$

Intuitively, it suffices to replace $\leq$ by $\sqsubseteq_\Delta$ and $\not\leq$ by $\not\sqsubseteq_\Delta$ in the algorithmic rules of Figure 1 (where Δ is the set of monomorphic variables used in the premises) to obtain the corresponding rules of type-substitution inference. This yields the system formed by the rules in Figure 2 plus the rules for constants and variables (omitted: they are the same as in Figure 1). Of particular interest is the rule (Inf-abstr) which has become simpler than in Figure 1 since it works under the hypothesis that λ-abstractions have empty decorations, and which uses the $\Delta\cup\Delta'$ set to compare the types of the body with the result types specified in the interface ($s_i' \sqsubseteq_{\Delta\cup\Delta'} s_i$). Notice that we do not require the sets of type-substitutions that make $s_i' \sqsubseteq_{\Delta\cup\Delta'} s_i$ satisfied to be the same for all $i{\in}I$: this is not a problem since the case of different sets of type-substitutions corresponds to using their union as sets of type-substitutions (*ie*, to intersecting them point-wise: see Definition A.9 and Corollary A.12 —hitherto, references starting with letters refer to appendixes).

It still remains the most delicate rule, (Inf-appl), the one for application. It is difficult because not only it must find two distinct sets of type-substitutions (one for the function type the other for the argument type) but also because the set of type-substitutions for the function type must enforce two distinct constraints: that the type is smaller than $\mathbb{0}\to\mathbb{1}$, and that its domain is compatible with the type of the argument. In order to solve all these constraints we collapse them into a unique definition which is the algorithmic counterpart of the set of types used in Section 2.1 to define the operation $t \cdot s$ occurring in the rule (Alg-appl). Precisely we define $t \bullet_\Delta s$ as the set of types for which there exist two sets of type-substitutions (for variables not in Δ) that make s compatible with the domain of t:

$$t \bullet_\Delta s \stackrel{\text{def}}{=} \left\{ u \;\middle|\; \begin{array}{l} [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{0}\to\mathbb{1} \\ [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \mathsf{dom}(t[\sigma_j]_{j\in J}) \\ u = t[\sigma_j]_{j\in J} \cdot s[\sigma_i]_{i\in I} \end{array} \right\}$$

In practice, this set takes all the pairs of sets of type-substitutions that make t a function type, and s an argument type compatible with t and collects all the possible result types. This set is closed by intersection (see Lemma A.8) which is an important property since it ensures that if we find two distinct solutions to type an application, then we can also use their intersection. Unfortunately, this property is not enough to ensure that this set has a minimum type (for that we also need to prove that the intersection of all the types in the set can be expressed as a *finite* intersection) which would imply the existence of a principal type (which is still an open problem). For the application of a function of type t to an argument of type s, the inference system deduces every type in $t \bullet_\Delta s$. This yields the inference rule (Inf-appl) of Figure 2.

These type-substitution inference rules are sound and complete with respect to the typing algorithm, modulo the restriction that all the decorations in the λ-abstractions are empty. Both of these properties are stated in terms of the *erase*(.) function that maps expressions of the explicitly-typed calculus into expressions of the implicitly-typed one by erasing in the former all occurrences of sets of type-substitutions.

Theorem 3.2 (Soundness of inference)**.** *Let a be an implicitly-typed expression. If $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t$, then there exists an explicitly-typed expression e such that erase$(e) = a$ and $\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t$.*

The proof of the soundness property is constructive: it builds along the derivation for the implicitly-typed expressions a an explicitly-typed expression e that satisfies the statement of the theorem; this expression is the one that is then compiled in the intermediate language of Part 1 and evaluated. Notice that $\sqsubseteq_\Delta$ gauges the generality of the solutions found by the inference system: the smaller the type found, the more general the solution is. As a matter of facts, adding to the system in Figure 2 a subsumption rule that uses the relation $\sqsubseteq_\Delta$ that is:

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t_1 \quad t_1 \sqsubseteq_\Delta t_2}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t_2}\ (\textit{subsumption})$$

is sound. This means that the set of solutions is upward closed with respect to $\sqsubseteq_\Delta$ and that from smaller solutions it is possible (by such a subsumption rule) to deduce the larger ones. In that respect, the completeness theorem that follows states that the inference system can always deduce for the erasure of an expression a solution that is at least as good as the one deduced for that expression by the type system for the explicitly-typed calculus.

Theorem 3.3 (Completeness of inference)**.** *Let e be an (explicitly-typed) expression in which all decorations are empty. If $\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t$, then there exists a type t' such that $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}}$ erase$(e) : t'$ and $t' \sqsubseteq_\Delta t$.*

The inference system is syntax directed and describes an algorithm that is parametric in the decision procedures for $\sqsubseteq_\Delta$ and $\bullet_\Delta$. The problem of deciding these two relations is tackled next.

3.2 Type tallying

We define the tallying problem as follows

Definition 3.4 (Tallying problem)**.** *Let C be a* constraint-set*, that is, a finite set of pairs of types (these pairs are called* constraints*), and Δ a finite set of type variables. A type-substitution σ is a solution for the* tallying problem of C and Δ *(noted $\sigma \Vdash_\Delta C$) if $\sigma \sharp \Delta$ and for all $(s,t) \in C$, $s\sigma \leq t\sigma$ holds.*

The definition of the tallying problem is the cornerstone of our type-substitution inference system, since every problem we have to solve to "implement" the rules of Figure 2 is reduced to different instances of this problem.

With the exception of (INF-APPL), it is not difficult to show that the "implementation" of the rules of the type-substitution inference system $\vdash_{\mathscr{I}}$ corresponds to finding and solving a particular tallying problem. First, notice that for the remaining rules the problem we have to solve is to prove (or disprove) the relation $s \sqsubseteq_\Delta t$ for given s and t. By definition this corresponds to find a set of n type-substitutions $[\sigma_i]_{i\leq n}$ such that $\bigwedge_{i\leq n} s\sigma_i \leq t$. We can split each type-substitution σ_i in two: a *renaming type-substitution* ρ_i that maps each variable of s not in Δ into a *fresh* type variable and a type substitution σ_i' such that $\sigma_i = \sigma_i' \circ \rho_i$. Thus the inequation becomes $\bigwedge_{i\leq n}(s\rho_i)\sigma_i' \leq t$. The domains of σ_i' are by construction pairwise disjoints (they are formed of distinct fresh variables) and disjoint from the variables in t; thus we can merge them into a single substitution $\sigma = \bigcup_{i\leq n}\sigma_i'$ and apply it to t with no effect, yielding the inequation $(\bigwedge_{i\leq n} s\rho_i)\sigma \leq t\sigma$. Let $u_n = \bigwedge_{i\leq n} s\rho_i$, we have just transformed the problem of proving the relation $s \sqsubseteq_\Delta t$ into the problem of finding an n for which there exists a solution to the tallying problem for $\{u_n \leq t\}$ and Δ. The way to proceed to find n is explained in Section 3.2.3.

The (INF-APPL) rule deserves to be dealt apart since it needs to solve a more difficult problem. A "solution" for the (INF-APPL) rule problem is *a pair* of sets of type-substitutions $[\sigma_i]_{i\in I}$, $[\sigma_j]_{j\in J}$ for variables not in Δ such that both $\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1}$ and $\bigwedge_{j\in J} s\sigma_j \leq \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i)$ hold. In this section we give an algorithm that produces a set of solutions for the (INF-APPL) rule problem that is sound (it finds only correct solutions) and complete (any other solution can be derived from those returned by the algorithm). To this end we proceed in three steps: (i) given a tallying problem, we show how to effectively produce a finite set of solutions that is sound (it contains only correct solutions) and complete (every other solution of the problem is less general—in the usual sense of unification, *ie*, it is larger wrt $\sqsubseteq$— than some solution in the set); (ii) we show that if we fix the cardinalities of I and J, then it is possible to reduce the (INF-APPL) rule problem to a tallying problem; (iii) from this we deduce a sound and complete algorithm to semi-decide the general (INF-APPL) rule problem and thus the whole inference system.

We solve each problem in one of the next subsections, but before we recall an important property of semantic subtyping systems [4, 12] which states that every type is equivalent to (and can be effectively transformed into) a type in *disjunctive normal form*, that is, a union of *uniform* intersections of *literals*. A literal is either an arrow, or a basic type, or a type variable, or a negation thereof. An intersection is uniform if it is composed of literals with the same constructor, that is, either it is an intersection of arrows, type variables, and their negations or it is an intersection of basic types, type variables, and their negations. In summary, a disjunctive normal form is a union of summands whose form is either

$$\bigwedge_{p\in P} b_p \wedge \bigwedge_{n\in N} \neg b_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \tag{13}$$

or

$$\bigwedge_{p\in P} (s_p\to t_p) \wedge \bigwedge_{n\in N} \neg(s_n\to t_n) \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \tag{14}$$

When either P' or N' is non empty, we call the variables α_q's and α_r's the *top-level variables* of the normal form.

3.2.1 Solution of the tallying problem.

In order to solve the tallying problem for given Δ and C, we first fix some total order $\preccurlyeq$ —any will do— on the type variables occurring in C and not in Δ (from now on, when speaking of type variables we will mean type variables not in Δ). Next, we produce sets of constraint-sets in a particular form by proceeding in four steps: first, we **normalize** the constraint-sets (so that at least one of the two types of every constraint is a type variable); second, we **merge** constraints that are on the same variables; third, we **solve** all these constraint-sets producing solvable sets of equations equivalent to the original problem and solve it; fourth, we combine these three steps into an **algorithm** that produces a sound and complete set of solutions of the tallying problem. To this end we define two operations on sets of constraint-sets:

Definition 3.5. *Let $\mathscr{S}_1, \mathscr{S}_2 \subseteq \mathscr{P}(\mathscr{T}\times\mathscr{T})$ be two sets of constraint-sets. We define*

$$\mathscr{S}_1 \sqcap \mathscr{S}_2 \stackrel{\text{def}}{=} \{C_1 \cup C_2 \mid C_1 \in \mathscr{S}_1, C_2 \in \mathscr{S}_2\}$$
$$\mathscr{S}_1 \sqcup \mathscr{S}_2 \stackrel{\text{def}}{=} \mathscr{S}_1 \cup \mathscr{S}_2$$

By convention the empty set of constraint-sets is unsolvable (it denotes failure in finding a solution), while the set containing the empty set is always satisfied.

We also define an auxiliary function single that singles out a given top-level variable of a normal form. More precisely, given a type t which is a summand of a normal form, that is, $t = \bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r$ and $k \in P' \cup N'$, we define $\mathsf{single}(\alpha_k, t)$ as the constraint equivalent to $t \leq \mathbb{0}$ in

0. $\mathsf{norm}(t, M)$ =

1. if $t \in M$ then return $\{\varnothing\}$ else

2. if $t = \bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r$ and α_k is the smallest variable (wrt $\preccurlyeq$) for k in $P'\cup N'$ then return$\{\mathsf{single}(\alpha_k, t)\}$ else

3. if $t = \bigwedge_{p\in P} b_p \wedge \bigwedge_{n\in N} \neg b_n$ then (if $\bigwedge_{p\in P} b_p \leq \bigvee_{n\in N} \neg b_n$ then return $\{\varnothing\}$ else return $\varnothing$) else

4. if $t = \bigwedge_{p\in P}(s_p\to t_p) \wedge \bigwedge_{n\in N} \neg(s_n\to t_n)$ then return

5. $$\bigsqcup_{n\in N}\ \bigsqcap_{P'\subset P}\Bigg(\Big(\big(\mathsf{norm}(s_n \wedge \bigwedge_{p\in P'} \neg s_p, M\cup\{t\}) \sqcup \mathsf{norm}(\bigwedge_{p\in P\setminus P'} t_p \wedge \neg t_n, M\cup\{t\})\big)\Big) \sqcap \mathsf{norm}(s_n \wedge \bigwedge_{p\in P'} \neg s_p, M\cup\{t\})\Bigg) \text{ else}$$

6. if $t = \bigvee_{i\in I} t_i$ then return $\bigsqcap_{i\in I} \mathsf{norm}(t_i, M)$ else let t' be the disjunctive normal form of t in return $\mathsf{norm}(t', M)$

Figure 3. Constraint normalization

which α_k is "singled-out", that is,
$\bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in(N'\setminus\{k\})} \neg\alpha_r \leq \alpha_k$
when $k \in N'$ and
$\alpha_k \leq \bigvee_{p\in P} \neg t_p \vee \bigvee_{n\in N} t_n \vee \bigvee_{q\in(P'\setminus\{k\})} \neg\alpha_q \wedge \bigwedge_{r\in N'} \alpha_r$
when $k \in P'$. Henceforth, to enhance readability we will often write $s \leq t$ for the constraint (s, t).

EXAMPLE. *We will show the various phases of the process by solving the tallying problem for the following constraint-set:*
$C = \{(\alpha \to \mathtt{Bool}, \beta \to \beta)\ ,\ (\mathtt{Int}\vee\mathtt{Bool} \to \mathtt{Int}\ ,\ \alpha \to \beta)\}$
and assume that $\alpha \preccurlyeq \beta$.

1. Constraint normalization. We define a function norm that takes a type t and generates a set of *normalized* constraint-sets — *ie*, constraint-sets formed by constraints whose form is either $\alpha{\leq}s$ or $s{\leq}\alpha$— whose set of solutions is sound and complete w.r.t. the constraint $t \leq \mathbb{0}$. This function is parametric in a *memoization* set M and the algorithm to compute it is given in Figure 3. If the input type t is not in normal form, then the algorithm is applied to the disjunctive normal form t' of t (end of line 6). Since a union is empty if and only if every summand that composes it is empty, then the algorithm generates a new constraint-set for the problem that equates all the summands to $\mathbb{0}$ (beginning of line 6). If a summand contains a top-level variable, then the smallest (wrt $\preccurlyeq$) top-level variable is singled out (line 2). If there is no top-level variable and there are only basic types, then the algorithm checks the constraint by calling the subtyping algorithm and, accordingly, it returns either the unsatisfiable set of constraint-sets ($\varnothing$) or the one that is always satisfied ($\{\varnothing\}$) (line 3). Finally, if there are only intersections of arrows and their negations, then the problem is decomposed into a set of subproblems by using the decomposition rule of the subtyping algorithm for semantic subtyping (see [12] for details), after having added t to the set of memoized types. The regularity of types ensures that the algorithm always terminates. Notice that, in line 2 the algorithm always singles out the smallest variable. Therefore, by construction, if norm generates a constraint (α, t) or (t, α), then every variable smaller than or equal to α may occur in t only under an arrow (equivalently, every top-level variable of t is strictly larger than α).

REMARK 3.1. *There is the special case of* (α, t) *or* (t, α) *in which* t *is itself a variable. In that case we give priority to the smallest variable and consider the larger variable be a bound for the lower one but not vice-versa. This point will be important for the merge.*

A constraint-set in which all constraints satisfy this property is said to be *well ordered* (*cf.* Definition B.16).

EXAMPLE (Cont'd). *The function* norm *works on single constraints (actually, on a type* t *representing the constraint* $t \leq \mathbb{0}$*), so let us apply it on the first constraint of the example. We want to normalize the constraint* $\alpha{\to}\mathtt{Bool} \leq \beta{\to}\beta$*, and thus we apply* norm *to the type* $(\alpha{\to}\mathtt{Bool}) \wedge \neg(\beta{\to}\beta)$*. Now, this constraint has two distinct solutions: either* (i) β *is the empty set, in which case the larger type becomes* $\mathbb{0}{\to}\mathbb{0}$ *that is the type of all functions (see Footnote 3) which contains every arrow type, in particular* $\alpha{\to}\mathtt{Bool}$*, or* (ii) *the types satisfy the usual covariant-contravariant rule for arrows, that is,* $\beta \leq \alpha$ *and* $\mathtt{Bool} \leq \beta$*. Since there are* two *distinct solutions, then* norm *generates a set of* two *constraint-sets. Precisely* $\mathsf{norm}((\alpha{\to}\mathtt{Bool}) \wedge \neg(\beta{\to}\beta), \varnothing)$ *returns* $\{\ \{(\beta, \mathbb{0})\}\ ,\ \{(\beta, \alpha)\ ,\ (\mathtt{Bool}, \beta)\}\ \}$. *Both constraint-sets are normalized and are computed by Line 5 in Figure 3: the first constraint-set is computed by the rightmost recursive call of* norm *(notice that* $P'{=}\varnothing$ *—since it ranges over the* strict subsets *of* P *which, in this case, is a singleton—so it requires* s_n*, that is* β*, to be empty), while the second constraint-set is obtained by the union of the first two recursive calls (which require* $s_n \leq s_p$ *and* $t_p \leq t_n$*).*

2. Constraint merging. Take a normalized constraint-set. Each constraint of this set isolates one particular variable. However, the same variable can be isolated by several distinct constraints in the set. We next want to transform this constraint-set into an equivalent one (*ie*, a constraint-set with exactly the same set of solutions) in which every variable is isolated in at most two constraints, one with the variable on the left of the constraint and one with it on the right. In other words, we want to obtain a normalized constraint-set in which each variable has at most one upper bound and at most one lower bound. In practice, this set represents a set of constraints of the form $\{\ s_i{\leq}\alpha_i{\leq}t_i \mid i \in I\ \}$ where the α_i's are pairwise distinct. This is done by the function $\mathsf{merge}(C, M)$ where C is a normalized constraint-set and M a memoization set containing types.

$\mathsf{merge}(C, M)$ =

1. Rewrite C by applying as long as possible the following rules according to the order $\preccurlyeq$ on the variables (smallest first) :
   – if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
   – if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1{\vee}s_2, \alpha)$;
2. if there exist two constraints (s, α) and (α, t) in C s.t. $s{\wedge}\neg t \notin M$,
   then let $\mathscr{S} = \{C\} \sqcap \mathsf{norm}(s{\wedge}\neg t, \varnothing)$
   in return $\bigsqcup_{C'\in\mathscr{S}} \mathsf{merge}(C', M{\cup}\{s{\wedge}\neg t\})$
   else return $\{C\}$

The function above is a little more complicated than what we described, since it returns a *set* of normalized constraint sets. The reason of this is the use of norm in the second step of the function. But let us proceed in order. The function merge performs two steps. In the first step it scans (using $\preccurlyeq$ so as to give priority to smaller variables, *cf.* Remark 3.1) the variables isolated by the normalized constraint-set C and for each such variable it merges all the constraints by unioning all its lower bounds and intersecting all its upper bounds. For instance, if C contains the following five constraints for α: (s_1, α), (s_2, α), (α, t_1), (α, t_2), (α, t_3), then the first step replaces them by $(s_1{\vee}s_2, \alpha)$ and $(\alpha, t_1{\wedge}t_2{\wedge}t_3)$, which corresponds to having the constraint $s_1{\vee}s_2 \leq \alpha \leq t_1{\wedge}t_2{\wedge}t_3$. Such a constraint is satisfiable only if the constraint that the lower bound of α is smaller than its upper bound is satisfiable. This is checked in the second step, which looks for pairs of constraints of the form (s, α) and (α, t) (thanks to the first step we know that for each variable there is at most one such pair) and then adds the constraint (s, t) to C. This constraint is equivalent to $(s{\wedge}\neg t, \mathbb{0})$ but neither it or (s, t) is normalized. Thus before adding it to C we normalize it by calling $\mathsf{norm}(s{\wedge}\neg t, \varnothing)$. Recall that norm returns a *set* of constraint-sets, each constraint-set corresponding to a distinct solution. So we add the constraints that are in C to all the constraint-sets that are the result of $\mathsf{norm}(s\wedge\neg t, \varnothing)$ via the $\sqcap$ operator (this is why merge returns a set of constraint-sets rather than a single one). The constraint-sets so obtained are normalized but they may be not merged, yet. So we recursively apply merge to all of them (via the operator $\sqcup$) and memoize $s\wedge\neg t$ in M. Of course, this step 2 is done only if the constraint (s, t) was not already embedded in C before, that is, only if $s\wedge\neg t$ is not already in M. Note that merge preserves the property that in every constraint (α, t) or (t, α), every variable smaller than or equal to α may occur in t only under an arrow.

EXAMPLE (Cont'd). *If we apply* norm *also to the second constraint of our example we obtain a second set of constraint-sets:* $\{\ \{(\alpha,\mathbb{0})\}\ ,\ \{(\alpha,\mathtt{Int}\vee\mathtt{Bool}),(\mathtt{Int},\beta)\}\ \}$. *To obtain a sound and complete set of solutions for our initial C we have to consider all the possible combinations (see Step 1 of the constraint solving algorithm later on) of the two sets obtained by normalizing C, that is, a set of* four *constraint-sets:*

$$\begin{array}{l}\{\quad \{(\alpha,\mathbb{0}),(\beta,\mathbb{0})\}\ ,\\ \quad\ \ \{(\alpha,\mathtt{Int}\vee\mathtt{Bool}),(\mathtt{Int},\beta),(\beta,\mathbb{0})\}\ ,\\ \quad\ \ \{(\mathtt{Bool},\beta),(\beta,\alpha),(\alpha,\mathbb{0})\}\ ,\\ \quad\ \ \{(\mathtt{Bool},\beta),(\mathtt{Int},\beta),(\beta,\alpha),(\alpha,\mathtt{Int}\vee\mathtt{Bool})\}\ \ \}\end{array}$$

The application of merge *to the first set leaves it unchanged. Merge on the second one returns an empty set of constraint-sets since at the second step it tries to solve* $\mathtt{Int}\leq\mathbb{0}$. *The same happens for the third since it first adds* $\beta\leq\mathbb{0}$ *and at the recursive call tries to solve* $\mathtt{Bool}\leq\mathbb{0}$. *The fourth one is more interesting: in step 1 it replaces* $(\mathtt{Bool},\beta)$ *and* $(\mathtt{Int},\beta)$ *by* $(\mathtt{Int}\vee\mathtt{Bool},\beta)$ *and at the second step adds* $(\beta,\mathtt{Int}\vee\mathtt{Bool})$ *obtained from* (β,α) *and* $(\alpha,\mathtt{Int}\vee\mathtt{Bool})$ *(it also checks* $(\mathtt{Int}\vee\mathtt{Bool},\mathtt{Int}\vee\mathtt{Bool})$ *which is always satisfied). So after merge we have* $\{\ \{(\alpha,\mathbb{0}),(\beta,\mathbb{0})\}\ ,\ \{(\beta,\alpha),(\alpha,\mathtt{Int}\vee\mathtt{Bool}),(\mathtt{Int}\vee\mathtt{Bool},\beta),(\beta,\mathtt{Int}\vee\mathtt{Bool})\}\ \}$. *Notice that we* did not *merge* (β,α) *and* $(\beta,\mathtt{Int}\vee\mathtt{Bool})$ *into* $(\beta,\alpha\wedge(\mathtt{Int}\vee\mathtt{Bool}))$*: since* $\alpha\preccurlyeq\beta$, *then* α *is not considered an upper bound of* β *(see Remark 3.1) and thanks to that the resulting constraint-set is well ordered.*

3. Constraint solving. norm and merge yield a set in which every constraint-set is of the form $C = \{s_i{\leq}\alpha_i{\leq}t_i \mid i \in I\}$ where α_i are pairwise distinct variables and s_i and t_i are respectively $\mathbb{0}$ or $\mathbb{1}$ whenever the corresponding constraint is absent. If there is a constraint on two variables, then again priority is given to the smaller variable. For instance, if $\alpha\preccurlyeq\beta$, then $\{(\alpha,\beta)\}$ will be considered to represent $\{(\mathbb{0}{\leq}\alpha{\leq}\beta),(\mathbb{0}{\leq}\beta{\leq}\mathbb{1})\}$. Thanks to this assumption the system so obtained is well ordered, that is, for every constraint $s{\leq}\alpha{\leq}t$ in it, the top-level variables of s and t are strictly larger than α. Notice that in doing that we do not lose any information: the bounds for larger variables are still recorded in those of smaller one and any bound for larger variables obtained by transitivity on the smaller variable is already in the system by step 2 of merge. The last step is to *solve* this constraint-set, that is, to transform it into a solvable set of equations that then we solve by a Unify algorithm that exploits the particular form of the equations obtained from a well-ordered constraint-set. Let C be a well-ordered constraint-set of the above form; we define $\mathsf{solve}(C)$ as follows:

$$\mathsf{solve}(C) = \{\alpha = (s\vee\beta)\wedge t \mid (s\leq\alpha\leq t)\in C, \beta \text{ fresh}\}$$

The function $\mathsf{solve}(C)$ takes every constraint $s\leq\alpha\leq t$ in C and replaces it by $\alpha = (s\vee\beta)\wedge t$ (with β fresh). It is clear that the constraint-set C has a solution for every possible assignment of α included between s and t if and only if the new constraint-set has a solution for every possible (unconstrained) assignment of β. At the end the constraint-set $\{s_i{\leq}\alpha_i{\leq}t_i \mid i \in I\}$ has become a set of equations of the form $\{\alpha_i = u_i \mid i\in I\}$ where the α_i's are pairwise distinct. By construction, this set of equations has the property that every variable that is smaller than or equal to (wrt $\preccurlyeq$) α_i may occur in u_i only under an arrow (as for constraint-sets we say that the set of equations is *well ordered*). This last property ensures the contractivity of the equation defining the smallest type variable. By Courcelle [8] (and Lemma B.44) there exists a solution of this set, namely, a substitution from the type variables $\alpha_1, ..., \alpha_n$ into (possibly recursive regular) types $t_1, ..., t_n$ whose variables are contained in the fresh β_i's variables introduced by solve (all universally quantified, *ie*, no upper or lower bound) and the type variables in Δ. This solution is given by the following Unify procedure in which we use μ-notation to denote regular types and where E is a well-ordered set of equations.

$\mathsf{Unify}(E)$=
 if $E = \varnothing$ then return $\{\}$ else
 – select in E the equation $\alpha = t_\alpha$ for the smallest α (wrt $\preccurlyeq$)
 – let E' be the set of equations obtained by replacing in $E\backslash\{\alpha = t_\alpha\}$ every occurrence of α by $\mu X.(t_\alpha\{^X/_\alpha\})$ (X fresh)
 – let $\sigma = \mathsf{Unify}(E')$ in return $\{\alpha = (\mu X.t_\alpha\{^X/_\alpha\})\sigma\}\cup\sigma$

Thanks to the well-ordering of E, Unify generates a set of solutions in which all types satisfy the contractivity condition on infinite branches of Definition 2.1. It solves the (contractive) recursive equation of the smallest variable α defined by E (if α does not occur in t_α, then the μ-abstraction can be omitted), replaces this solution in the remaining equations, solves this set of equations, and applies the solution so found to the solution of α so as to solve the other variables occurring in its definition.

4. The complete algorithm. The algorithm to solve the tallying problem for C and variables not in Δ, then, proceeds in three steps:

Step 1. Let $\mathscr{N} = \prod_{(s,t)\in C}\mathsf{norm}(s\wedge\neg t,\varnothing)$. If $\mathscr{N} = \varnothing$ then **fail** else proceed to the next step.

Step 2. Let $\mathscr{M} = \bigsqcup_{C\in\mathscr{N}}\mathsf{merge}(C,\varnothing)$. If $\mathscr{M} = \varnothing$ then **fail** else proceed to the next step.

Step 3. Let $\mathscr{S} = \bigsqcup_{C\in\mathscr{M}}\mathsf{solve}(C)$. Return $\{\mathsf{Unify}(E) \mid E{\in}\mathscr{S}\}$.

Let $\mathsf{Sol}_\Delta(C)$ denote the set of all substitutions obtained by the previous algorithm. They form a sound and complete set of solutions for the tallying problem:

Theorem 3.6 (Soundness and completeness)**.**

$$\begin{array}{rcl}\sigma\in\mathsf{Sol}_\Delta(C) & \Rightarrow & \sigma\Vdash_\Delta C\\ \sigma\Vdash_\Delta C & \Rightarrow & \exists\sigma'\in\mathsf{Sol}_\Delta(C),\sigma'',\ \textit{s.t.}\ \sigma\approx\sigma''\circ\sigma'\end{array}$$

where $\approx$ means that the two substitutions map the same variable into equivalent types. Regularity of types ensures the termination of the algorithm and, hence, the decidability of the tallying problem (the proof of these properties combines proofs of soundness, completeness, and termination of each step: see Appendix B).

EXAMPLE (End). *After Step 1 and 2 our initial tallying problem* $\{(\alpha{\to}\mathtt{Bool},\beta{\to}\beta)\ ,\ (\mathtt{Int}\vee\mathtt{Bool}\to\mathtt{Int},\alpha{\to}\beta)\}$ *has become* $\{\ \{(\alpha,\mathbb{0}),(\beta,\mathbb{0})\}\ ,\ \{(\beta,\alpha),(\alpha,\mathtt{Int}\vee\mathtt{Bool}),(\mathtt{Int}\vee\mathtt{Bool},\beta),(\beta,\mathtt{Int}\vee\mathtt{Bool})\}\ \}$. *Let us apply Step 3. The first constraint-set is trivial and it is easy to see that it yields the solution* $\{^{\mathbb{0}}/_\alpha, {}^{\mathbb{0}}/_\beta\}$. *The second constraint-set is* $\{(\beta{\leq}\alpha{\leq}\mathtt{Int}\vee\mathtt{Bool}),(\mathtt{Int}\vee\mathtt{Bool}{\leq}\beta\leq\mathtt{Int}\vee\mathtt{Bool})\}$. *We apply* solve *to the constraints for α obtaining* $\{\alpha = (\gamma\vee\beta)\wedge(\mathtt{Int}\vee\mathtt{Bool})\}$. *We find the solution for β (no need to substitute α since it does not occur in the constraints for β) which is* $\{\beta = \mathtt{Int}\vee\mathtt{Bool}\}$. *We replace β in the solution of α obtaining* $\{\alpha = (\gamma\vee\mathtt{Int}\vee\mathtt{Bool})\wedge(\mathtt{Int}\vee\mathtt{Bool})\}$. *The solution for this second constraint-set is then* $\{^{\mathtt{Int}\vee\mathtt{Bool}}/_\alpha, {}^{\mathtt{Int}\vee\mathtt{Bool}}/_\beta\}$, *which with* $\{^{\mathbb{0}}/_\alpha, {}^{\mathbb{0}}/_\beta\}$ *forms a sound and complete set of solutions for our initial tallying problem.*

Finally, solve introduces several fresh polymorphic variables which can be cleaned up when substitutions are applied: all variables that occur only in covariant (resp. contravariant) position, can be replaced by $\mathbb{0}$ (resp. $\mathbb{1}$). This is what we implicitly did in our example to solve β and eliminate γ from the constraint of α.

3.2.2 *Solution for application with fixed cardinalities*

It remains to solve the problem for the (INF-APPL) rule. We recall that given two types s and t, a solution for this problem is *a pair* of sets of type-substitutions (for variables not in Δ) that make both of these two inequations

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \qquad \bigwedge_{j\in J} s\sigma_j \leq \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) \qquad (15)$$

hold. Two complicacies are to be dealt with: (1) we must find *sets* of type substitutions, rather than a single substitution as in the tallying problem and (2) we have to get rid of the $\mathsf{dom}()$ function. If I and J have fixed cardinalities, then both difficulties can be easily surmounted and the whole problem be reduced to a tallying problem. To see how, consider the two inequations in (15). Since the two sets of substitutions are independent, then without loss of generality we can split each substitution σ_k (for $k\in I\cup J$) in two substitutions: a renaming substitution ρ_k that maps each variable in the domain of σ_k into a different fresh variable, and a second substitution σ'_k defined such that $\sigma_k = \sigma'_k \circ \rho_k$. The two inequations thus become $\bigwedge_{i\in I}(t\rho_i)\sigma'_i \leq \mathbb{0}\to\mathbb{1}$ and $\bigwedge_{j\in J}(s\rho_j)\sigma'_j \leq \mathsf{dom}(\bigwedge_{i\in I}(t\rho_i)\sigma'_i)$. Since the various σ'_k have disjoint domains, then we can union them into a single substitution $\sigma = \bigcup_{k\in I\cup J}\sigma'_k$, and the two inequations become $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \mathsf{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$. Now if we fix the cardinalities of I and J since the ρ_k are generic renamings, we have just transformed the problem in (15) into the problem of finding for two given types t_1 and t_2 all substitutions σ such that[6]

$$t_1\sigma \leq \mathbb{0}\to\mathbb{1} \qquad t_2\sigma \leq \mathsf{dom}(t_1\sigma) \qquad (16)$$

hold. Finally, we can prove (see Lemmas B.49 and B.50) that a type-substitution σ solves (16) if and only if it solves

$$t_1\sigma \leq \mathbb{0}\to\mathbb{1} \qquad t_1\sigma \leq (t_2\to\gamma)\sigma \qquad (17)$$

with γ fresh. We transformed the application problem (with fixed cardinalities) into the tallying problem for $\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$, whose set of solutions is a sound and complete set of solutions for the (INF-APPL) rule problem when I and J have fixed cardinalities.

3.2.3 *Solution of the application problem*

The algorithm to solve the general problem for the (INF-APPL) rule explores all the possible combinations of the cardinalities of I and J by, say, a dove-tail order. More precisely, we start with both I and J at cardinality 1 and:

Step A: Generate the constraint-set $\{(t_1, t_2\to\gamma)\}$ as explained in Subsection 3.2.2 (the constraint $t_1\leq\mathbb{0}\to\mathbb{1}$ is implied by this one since $\mathbb{0}\to\mathbb{1}$ contains every arrow type) and apply the tallying algorithm described in Subsection 3.2.1, yielding either a solution (a substitution for variables not in Δ) or a failure.

Step B: If all the constraint-sets failed at Step 1 of the algorithm of Subsection 3.2.1, then fail (the expression is not typeable). If they all failed but at least one did not fail in *Step 1*, then increase the cardinalities of I and J to their successor in the dove-tail order and start from *Step A* again. Otherwise all substitutions found by the algorithm are solutions of the application problem.

Notice that the algorithm returns a failure only if all the constraint-sets fail at Step 1 of the algorithm for the tallying problem. The reason is that up to Step 1 all the constraints at issue are on distinct occurrences of type variables: if they fail, there is no possible expansion that can make the constraint-set satisfiable. In Step 2 instead constraints of different occurrences of a same variable are merged. Thus even if the constraints fail, it may be the case that they will be satisfied by expanding different occurrences of a same variable into different variables. Therefore an expansion is tried. Solving the problem for $s \sqsubseteq_\Delta t$ is similar (there is just one set whose cardinality has to be increased at each step instead of two).

This constitutes a sound and complete semi-decision procedure for the application problem and, thus, for the type-substitution inference system (Theorem B.54). We defined some heuristics (omitted for space reasons: see Section B.2.3) to stop the algorithm when a solution seems unlikely. Whether these (or some coarser) halting conditions preserve completeness, that is, whether type-substitutions inference is decidable, is an open problem. We believe the system to be decidable. However, we fail to prove it when the type of the argument of an application is a union: its expansion distributes the union over the intersections thus generating new combinations of types. It comes as no surprise that the definitions of our heuristics are based on the cardinalities and depths of the unions occurring in the argument type.

Let us apply the algorithm to `map even`. We start with the constraint set $\{(\alpha_1\to\beta_1)\to\texttt{[}\alpha_1\texttt{]}\to\texttt{[}\beta_1\texttt{]} \leq t\to\gamma\}$ where $t = (\texttt{Int}\to\texttt{Bool})\wedge(\alpha\backslash\texttt{Int}\to\alpha\backslash\texttt{Int})$ is the type of `even` (we just renamed the variables of the type of `map`). After *Step A* the algorithm generates a set of nine constraint-sets (see Section 10.2.2 of [21] for more details on this example): one is unsatisfiable since it contains the constraint $t \leq \mathbb{0}$ (an intersection of arrows is never empty since it always contains $\mathbb{1}\to\mathbb{0}$ the type of the diverging functions); four of these are less general than some other (their solutions are included in the solutions of the other) and the remaining four are obtained by adding the constraint $\gamma \geq \texttt{[}\alpha_1\texttt{]}\to\texttt{[}\beta_1\texttt{]}$ respectively to $\{\alpha_1\leq\mathbb{0}\}$, $\{\alpha_1\leq\texttt{Int}, \texttt{Bool}\leq\beta_1\}$, $\{\alpha_1\leq\alpha\backslash\texttt{Int}, \alpha\backslash\texttt{Int}\leq\beta_1\}$, $\{\alpha_1\leq\alpha\vee\texttt{Int}, (\alpha\backslash\texttt{Int})\vee\texttt{Bool}\leq\beta_1\}$, yielding the following four solutions for γ: $\{\gamma=\texttt{[]}\to\texttt{[]}\}$, or $\{\gamma = \texttt{[Int]}\to\texttt{[Bool]}\}$, or $\{\gamma = \texttt{[}\alpha\backslash\texttt{Int]}\to\texttt{[}\alpha\backslash\texttt{Int]}\}$, or $\{\gamma = \texttt{[}\alpha\vee\texttt{Int]}\to\texttt{[(}\alpha\backslash\texttt{Int)}\vee\texttt{Bool]}\}$. Of these solutions only the last two are minimal. Since both are valid we could stop here and take their intersection, yielding the type expected in the introduction. If instead we dully follow the algorithm, then we have to perform a further iteration, expand the type of the function, yielding $\{((\alpha_1\to\beta_1)\to\texttt{[}\alpha_1\texttt{]}\to\texttt{[}\beta_1\texttt{]})\wedge((\alpha_2\to\beta_2)\to\texttt{[}\alpha_2\texttt{]}\to\texttt{[}\beta_2\texttt{]})\leq t\to\gamma\}$ for which the minimal solution is, as expected:
$\{\gamma = (\texttt{[}\alpha\backslash\texttt{Int]}\to\texttt{[}\alpha\backslash\texttt{Int]}) \wedge (\texttt{[}\alpha\vee\texttt{Int]}\to\texttt{[(}\alpha\backslash\texttt{Int)}\vee\texttt{Bool]})\}$

A final word on completeness. The theorem states that for every solution of the inference problem, our algorithm finds a solution that is more general. However this solution is not necessary the first one found by the algorithm: even if we find a solution, continuing with a further expansion may yield a more general solution. We have just seen that in the case of `map even` the good solution is the second one, although this solution could already have been deduced by intersecting the first minimal solutions we found. A simple example that shows that carrying on after a first solution may yield a better solution is the application of a function of type $(\alpha \times \beta) \to (\beta \times \alpha)$ to an argument of type $(\texttt{Int}\times\texttt{Bool}) \vee (\texttt{Bool}\times\texttt{Int})$. For this applications our algorithm (extended with product types) returns after one iteration the type $(\texttt{Int}\vee\texttt{Bool}) \times (\texttt{Int}\vee\texttt{Bool})$ (since it unifies α with β) while one further iteration allows the system to deduce the more precise type $(\texttt{Int}\times\texttt{Bool}) \vee (\texttt{Bool}\times\texttt{Int})$. Of course this raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist? This is a problem we did not tackle in this work, but if the answer to the previous question were negative, then it would be easy to prove the existence of a principal type: since at each iteration there are only finitely many solutions, then the principal type would be the intersection of the minimal solutions of the last iteration.

[6] Precisely, we have $t_1 = \bigwedge_{i=1..|I|} t_i^1$ and $t_2 = \bigwedge_{i=1..|J|} t_i^2$ where for $h = 1, 2$ each t_i^h is obtained from t_h by renaming the variables not in Δ into fresh variables.

Finally, notice that we did not give any reduction semantics for the implicitly-typed calculus. The reason is that its semantics is defined in terms of the semantics of the explicitly-typed calculus: the relabeling at run-time is an essential feature —independently from the fact that we started from an explicitly-typed expression or not— and we cannot avoid it. If we denote by $\textit{erase}^{-1}(a)$ the set of expressions e that satisfy the statement of Theorem 3.2, then the (big-step) semantics for an implicitly-typed expression a is given in terms of $\textit{erase}^{-1}(a)$: if an expression in $\textit{erase}^{-1}(a)$ reduces to v, so does a. As we see the result of computing an implicitly-typed expression is a value of the explicitly-typed calculus (so λ-abstractions may contain non-empty decorations) and this is unavoidable since it may be the result of a partial application (this can be made transparent by returning just the type and the "value" `<fun>` as in OCaml's toplevel). Also notice that the semantics is not deterministic since different expressions in $\textit{erase}^{-1}(a)$ may yield different results. However this may happen only in one particular case, namely, when an occurrence of a polymorphic function flows into a type-case and its type is tested. For instance the application $(\lambda^{(\texttt{Int}\to\texttt{Int})\to\texttt{Bool}} f.f{\in}\texttt{Bool}{\to}\texttt{Bool}\,?\,\texttt{true}:\texttt{false})(\lambda^{\alpha\to\alpha}x.x)$ results into `true` or `false` according to whether the polymorphic identity at the argument is instantiated by $[\{^{\texttt{Int}}/_{\alpha}\}]$ or by $[\{^{\texttt{Int}}/_{\alpha}\},\{^{\texttt{Bool}}/_{\alpha}\}]$. Once more this is unavoidable in a calculus that can dynamically test the types of polymorphic functions that admit several sound instantiations. This does not happen in practice since the inference algorithm always choose one particular instantiation (the existence of principal types would made this choice canonical and remove non determinism). So in practice the semantics is deterministic but implementation dependent.

In summary, programming in the implicitly-typed calculus corresponds to programming in the explicitly-typed one with the difference that we delegate to the system the task to write type-substitutions for us and with the caveat that by doing that we make the test of the type of a polymorphic function to be implementation dependent.

3.3 Examples

We implemented a prototype for the constraint solver which includes a basic simplification algorithm (it also performs type inference for implicitly-typed expressions). We use it to illustrate the behavior of our type-substitution inference on typical function interfaces.

```
1  map : (α → β) → [α] → [β]
2  pretty : Int → String
3  apply(map, pretty) >> ([] → []) ∧ ([Int] → [String])
4
5  even : (Int → Bool) ∧ (α\Int → α\Int)
6  apply(map, even) >> ([] → []) ∧
7                      ([Int] → [Bool]) ∧
8                      ([α\Int] → [α\Int]) ∧
9                      ([Int∨α] → [(Bool∨α)\Int])
10
11 f : (Int → Int) → Int → Int
12 id : α → α
13 apply(f, id) >> Int → Int
14
15 g : ((Int → Int) → Int → Int) ∧
16     ((Bool → Bool) → Bool → Bool)
17 apply(g, id) >> (Int → Int) ∧ (Bool → Bool)
18
19 fold : (α → β → β) → [α] → β → β
20 h : (Int → Int → Int) ∧ (Bool → Int → Int)
21 apply(fold, h) >> ([Int∨Bool] → Int → Int) ∧
22                    ([Int∨Bool] → Empty → Empty) ∧
23                    ([] → β → β)
```

The first example is the typical `map` function (our implementation also features product types, and `[`α`]` stands for the type $\mu X.\texttt{nil} \vee (\alpha, X)$). If we try to apply `map` to a `pretty` function (which converts integers into strings), we obtain an intersection type `([]`$\to$`[])`$\wedge$`([Int]`$\to$`[String])`. While the latter part of the type is the one an ML programmer would expect, our inference algorithm also deduces the special case `[]`$\to$`[]`. Interestingly, the constraint solver does not need to know the *body* of `map` to deduce that its output is empty when its input is; this is due to the particular encoding we used for lists: since lists are encoded as union types our system try to infer a result for every type in the union. The second example, the application of `map` to `even`, was described in details in the previous section (notice that, because of the naivety of the simplification algorithm, the first two types of the intersection returned by the prototype are redundant). We also see (Lines 11 to 17) that passing a polymorphic argument to a function with a monomorphic type (the other use of polymorphism) works as expected. Function `f` shows that when both types are expressible as ML types, our constraint solver behaves at least as well as ML type inference, and likewise for each separate piece of code of an overloaded function. Finally, in the last example (Lines 19 to 23) we apply the function `fold` to an overloaded function. The system returns quite a precise type for this application. Not only it returns the expected type `[Int`$\vee$`Bool]`$\to$`Int`$\to$`Int` (notice that the type of `h` is equivalent to `Int`$\vee$`Bool`$\to$`Int`$\to$`Int`) but it also states that (Line 22) if the initial value for the accumulator is a diverging expression (only diverging expressions have the empty type $\mathbb{0}$), then the whole application diverges and that (Line 23) if use as first argument the empty list, then the initial value of the accumulator is returned unchanged, whatever its type is.

4. Type reconstruction

Finally, the theory of type tallying we developed in Section 3 can be reused to perform type reconstruction, that is, to assign a type to functions whose interface is not specified —quite a desirable feature, especially for local functions—. The idea is to type the body of a function under the hypothesis that the function has the generic type $\alpha{\to}\beta$ and deduce the corresponding constraints. Formally, we consider expressions produced by the following grammar:

$$m \;::=\; c \mid x \mid mm \mid \lambda x.m \mid m{\in}t\,?\,m:m$$

together with the judgment $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$ that states that under the typing environment Γ, m has type t under the constraints in $\mathscr{S}$ (provided that $\mathscr{S}$ is satisfiable). These judgments are derived by the rules in Figure 4. These are quite standard apart from the fact that they derive multiple constraint-sets, rather than just one. This is due to the type reconstruction rule for type-cases, which explores four possible alternatives (m_0 diverges, it can match only the first, the second, or both cases). In this system the type of a well-typed expression is a type and a set of type-substitutions (*ie*, the set of all substitutions that are solution of the satisfiable constraint-sets in $\mathscr{S}$) and thus it is an intersection type obtained by applying this set of type-substitutions to the type.

The soundness of this system is a consequence of the results on the type-substitution inference of the previous sections. As a matter of facts, this system is precisely the same system as the one in the previous sections with the only difference that all interfaces are of the form $\alpha{\to}\beta$ and, to compensate that, we infer type-substitutions in decorations (we also used a different and more standard presentation to stress constraint generation). Of course, completeness does not hold: far from that. For instance, it is impossible, in general, to deduce for a function without type annotations the type $\mathbb{1}{\to}\mathbb{0}$ —the type of all diverging functions— since this would correspond to decide the halting problem (though our algorithm returns for $\mu f\,x{=}f(x)$ the same type as in ML, that is, $\alpha{\to}\beta$ which in our

$$\frac{}{\Gamma \vdash_{\mathscr{R}} c : b_c \rightsquigarrow \{\varnothing\}}\text{(R-CONST)} \qquad \frac{}{\Gamma \vdash_{\mathscr{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}}\text{(R-VAR)}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} m_1 m_2 : \alpha \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}}\text{(R-APPL)}$$

$$\frac{\Gamma, x : \alpha \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}}{\Gamma \vdash_{\mathscr{R}} \lambda x.m : \alpha \to \beta \rightsquigarrow \mathscr{S} \sqcap \{\{(t \leq \beta)\}\}}\text{(R-ABSTR)}$$

$$\text{(R-CASE)} \quad \frac{\begin{array}{rl} \mathscr{S} = & (\mathscr{S}_0 \sqcap \{\{(t_0 \leq \mathbb{0})\}\}) \\ \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0 \leq t), (t_1 \leq \alpha)\}\}) \\ \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0 \leq \neg t), (t_2 \leq \alpha)\}\}) \\ \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1 \vee t_2 \leq \alpha)\}\}) \end{array} \\ \Gamma \vdash_{\mathscr{R}} m_0 : t_0 \rightsquigarrow \mathscr{S}_0 \quad \Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \quad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} (m_0 \in t \,?\, m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}$$

where α, α_i and β in each rule are fresh type variables.

Figure 4. Type reconstruction rules

system is equivalent to $\mathbb{0}\to\mathbb{1}$). Likewise, completeness would imply decidability of reconstruction and thus imply decidability for intersection type systems, which are undecidable. Similarly, our reconstruction system cannot type the paradoxical functions we pointed out in the first part of this work (see Section 2 in [3]). However, if a function can be typed in ML-like type systems, then our type reconstruction rules can deduce a type at least as good as the ML one. Indeed, if we restrict our attention to the first four rules, the system produces a singleton set of constraints that is the same as in ML system (*cf.* [19]) and when constraint-sets are not circular (*ie*, their solution does not require recursive types), then our constraint solving algorithm coincides with unification (all fresh variables introduced by solve are simplified as we described at the end of Section 3.2.1 and solve directly produces a set of equations that are, in Martelli and Montanari's terminology [17], in *solved form*). Furthermore, since the types considered here are much richer than in ML (since they include unions, intersections, and negations), then our reconstruction may infer slightly better types. Type connectives alone bring, in particular, two advantages for type reconstruction: (i) the system deduces *sets* of type-substitutions (and thus deduces intersection types) and (ii) pattern matching (which can be seen as a type-case with singleton types) is typed more precisely (thanks in particular to intersections and negations). For instance, and contrary to ML, our type reconstruction types auto-application $\lambda x.xx$ for which it returns the recursive type $t = \mu X.(\alpha\wedge(X\to\beta)) \to \beta$. This type is a subtype of —thus, it is more precise than— the classic typing of auto-application in intersection type systems $t \leq (\alpha\wedge(\alpha\to\beta)) \to \beta$ (though it is not as precise as its subtype $\mu X.(\alpha\vee(X\to\beta)) \to \beta$ which can also type auto-application). As a final example, if we apply our type reconstruction algorithm (extended with products and recursive functions) to the type erasure of the map function defined in equation (12), then we obtain the type $((\alpha\to\beta) \to \texttt{[}\alpha\texttt{]}\to\texttt{[}\beta\texttt{]}) \wedge ((\mathbb{0}\to\mathbb{1}) \to \texttt{[]}\to\texttt{[]})$ (see the complete unfolding of the algorithm in Appendix C). Thanks to the precise typing of the type-case, our type is slightly more accurate than the ML type, since it states that the application of `map` to any function and the empty list returns the empty list.

Finally, the "type" returned by the type reconstruction algorithm is not always very readable and often needs to be simplified. For instance, the type we showed for `map` was obtained after some simplifications, few of which were done by hand (they simplified the type returned by the algorithm which is an intersection of eight arrow types) and defining an algorithm that does the right simplifications is not obvious (*eg*, how to detect that the type $(\alpha\wedge(\alpha\to\beta))\to\beta$ is much more readable than the type $\mu X.(\alpha\wedge(X\to\beta))\to\beta$ reconstructed for auto-application by our algorithm?). The simplification of types (or of type constraints) is a stand alone research topic that deserves further investigation. Nevertheless our reconstruction algorithm can already be used as is, to make type declaration of local functions optional. Indeed for local functions the system is not required to return a "readable" type to the programmer, but just to check whether there exists a typing for local functions that is compatible with their usage; and, for that, our system is enough.

5. Extensions

In this presentation we omitted two key features: pairs and recursive functions. Recursive functions do not pose any particular problem in the inference of type-substitutions while pairs are more challenging. The rule for pairs in inference system $\vdash_{\mathscr{I}}$ is the same as in the explicitly-typed calculus (this corresponds to disregarding sets of type-substitutions applied inside a pair, as they can equivalently be inferred outside the pair: $t_i \not\simeq \mathbb{0}$ and $(t_1{\times}t_2) \sqsubseteq_\Delta (s_1{\times}s_2) \Leftrightarrow t_i \sqsubseteq_\Delta s_i$). Instead, as expected, the rule for projection $\pi_i e$ needs some special care since if the type inferred for e is, say, t, then we need to find a set of substitutions $[\sigma_i]_{i\in I}$ such that $\bigwedge_{i\in I} t\sigma_i \leq \mathbb{1}{\times}\mathbb{1}$. This problem can be solved by using the very same technique we introduced for $\bullet_\Delta$, namely by solving a sequence of tallying problems generated by increasing at each step the cardinality of I. All the details can be found in the Appendix.

In the first part of this work we studied the extension of the explicitly-typed calculus with `let`-polymorphism, in particular, its typing and efficient execution (see Section 5.4 of [ANONYMIZED]). There we distinguished `let`-bound variables by underlining them. To extend our reconstruction to `let` we use a separate type environment Φ for these variables (while we reserve Γ for λ-abstracted variables). As in Damas-Milner $\mathscr{W}$ algorithm [9] we need to define $\overline{\Gamma}(t)$, the generalization (*closure* in [9]) of a type t wrt the type environment Γ, that is, $\overline{\Gamma}(t) \stackrel{\text{def}}{=} t[\{{}^{\gamma_i}\!/\!_{\alpha_i} \mid \alpha_i \in \mathsf{var}(t)\backslash\mathsf{var}(\Gamma)\}]$ where γ_i are fresh. Then the rules for type reconstruction are

$$(\textit{let-var}) \ \frac{}{\Phi \,;\, \Gamma \vdash_{\mathscr{R}} \underline{x} : \overline{\Gamma}(\Phi(\underline{x})) \rightsquigarrow \{\varnothing\}}$$

$$(\textit{let}) \ \frac{\Phi \,;\, \Gamma \vdash_{\mathscr{R}} e_1 : t_1 \rightsquigarrow \mathscr{S} \qquad \Phi, (\underline{x} : t_1) \,;\, \Gamma \vdash_{\mathscr{R}} e_2 : t_2 \rightsquigarrow \mathscr{S}'}{\Phi \,;\, \Gamma \vdash_{\mathscr{R}} \texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2 : t_2 \rightsquigarrow \mathscr{S} \sqcap \mathscr{S}'}$$

Finally, we want to stress there is at least one case in which we should have been *more* restrictive, that is, when an expression that is tested in a type-case has a polymorphic type. Our inference system may type it (by deducing a set of type-substitutions that makes it closed), even if this seems to go against the intuition: we are testing whether a polymorphic expression has a closed type. Although completeness ensures that in some cases it can be done, in practice it seems reasonable to consider ill-typed any type-case in which the tested expression has an open type (see Section A.3).

6. Related work

This section discusses the work on constraint-based type inference and inference for intersection type systems. Work on explicitly-typed intersection type systems and on XML processing languages is discussed in the first part.

Type inference in ML has essentially been considered as a constraint solving problem [18, 19]. We use a similar approach to solve the problem of type unification: finding a proper substitution that makes the type of the domain of a function compatible with the type of the argument it is applied to. Our type unification problem is essentially a specific set constraint problem [1]. This is applied in a much more complex setting with a complete set of type connectives and a rich set-theoretic subtyping relation. In particular, because of the presence of intersection types, to solve the problem of application one has to find *sets* of substitutions rather than just one substitution. This is reflected by the definition of our $\sqsubseteq$ relation which is much more thorough than the corresponding relation used in ML inference insofar as it encompasses instantiation, expansion, and subtyping. The important novelty of our work comes from the use of set-theoretic connectives, which allows us to turn sets of constraints of the form $s\leq\alpha\leq t$ into sets of equations of the form $\alpha = (\beta \vee s) \wedge t$. This set of equations is then solved using the Courcelle's work on infinite trees [8]. The use of type connectives also implies that we solve multiple sets of constraints, which account for different alternatives. Finally, it is worth noticing that [18, 19] use a richer language of constraints that includes binding. This allows separating constraint generation and constraint solving without compromising efficiency. Therefore an interesting direction of future research is either to re-frame our work into a richer language of constraints or to extend the work in [18, 19] to encompass our richer setting. This could be a first step towards the study of efficient constraint solving algorithms for our system.

Finally we want to stress that works on type reconstruction for intersection type systems are weakly related to our study. The reason is that the core of our technique consists in solving type (in-)equations by *recursive types*. With recursive types pure interesection type systems are trivially decidable since all terms can be typed by the type $\mu X.X\to X$. The problem we tackle here, thus, is fundamentally different, namely, we check whether it is safe to apply to each other expressions of two *explicitly given* (and possibly recursive) types in which some *basic types* may occur. There are however few similarities with some techniques developed for pure intersection type system that we discuss below.

Coppo and Giannini [7] presented a decidable type checking algorithm for simple intersection type system where intersection is used in the left-hand side of an arrow and only a term variable is allowed to have different types in its different occurrences. They introduced labeled intersections and labeled intersection schemes, which are intended to represent potential intersections. During an application $M\,N$, the labeled intersection schemes of M and N would be unified to make them match successfully, yielding a transformation, a combination of substitutions and expansions. An expansion expands a labeled intersection into an explicit intersection. The intersection here acts like a variable binding similar to a quantifier in logic. Our rule (ALG-INST) is similar to the transformation. We instantiate a quantified type into several instances according to different situations (*ie*, the argument types), and then combine them as an intersection type. The difference is that we instantiate a parametric polymorphic function into a function with intersection types, while Coppo and Giannini transform a potential intersection into an explicit intersection. Besides, as the general intersection type system is not decidable [5], to get a decidable type checking algorithm, Coppo and Giannini used the intersection in a limited way, while we give some explicit type annotations for functions. Likewise, Jim [14] proposed a type inference algorithm for a polar type system where intersection is allowed only in negative positions and System F-like quantification only in positive ones.

Restricting intersection types to finite ranks (using Leivant's notion of rank [16]) also yields decidable systems. Van Bakel [20] gave the first unification-based inference algorithm for a rank 2 intersection type system. Jim [13] studied a decidable rank 2 intersection type system extended with recursion and parametric polymorphism. Kfoury and Wells proved decidability of type inference for intersection type systems of arbitrary finite rank [15]. As a future work, we want to investigate decidability of rank-restricted versions of our calculus.

7. Conclusion

The work presented in this and in its companion paper [3] provides the theoretical basis and all the algorithmic tools needed to design and implement polymorphic functional languages for semi-structured data and, more generally, for functional languages with recursive types and set-theoretic unions, intersections, and negations. In particular, our results pave the way to the polymorphic extension of ℂDuce [2] and to the definition of a real type system for XQuery 3.0 [10] (not just one in which all higher-order functions have type "`function()`"). Thanks to type reconstruction, these languages can have a syntax and semantics very close to those of OCaml or Haskell, but will include primitives (in particular, complex patterns) to exploit the great expressive power of full-fledged set-theoretic types.

Some problems are still open, notably the decidability of type-substitution inference, but these are of theoretical nature and should not have any impact in practice (as a matter of facts people program in Java and Scala even though the decidability of their type systems is still an open problem). The only problem open in this second part of the work, that is the non determinism of the implicitly typed calculus, should have a negligible practical impact, insofar as it is theoretical (in practice, the semantics is deterministic but implementation dependent) and it concerns only the case when the type of (an instance of) a polymorphic function is tested at run-time: in our programming experience with ℂDuce we never met a single typecase for a function type. Nevertheless, it may be interesting to study how to remove such a latitude either by defining a canonical choice for the instances deduced by the inference system (a problem related to the existence of principal types), or by imposing reasonable restrictions, or by checking the flow of polymorphic functions by a static analysis.

On the practical side the most interesting directions of research are the study of heuristics to simplify types generated from constraints, so as to make type reconstruction for top-level functions human friendly; the generation of meaningful type error messages; the study of efficient implementation of constraint-solving; the definition and implementation of the polymorphic version of ℂDuce.

Acknowledgments. This work was partially supported by the ANR TYPEX project n. ANR-11-BS02-007. Zhiwu Xu was also partially supported by an Eiffel scholarship of French Ministry of Foreign Affairs and by the grant n. 61070038 of the National Natural Science Foundation of China.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93*, pages 31–41, 1993.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03*. ACM Press, 2003.

[3] G. Castagna, K. Nguyễn, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In *POPL '14*, January 2014. To appear.

[4] G. Castagna and Z. Xu. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11*, 2011.

[5] M. Coppo and M. Dezani. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19:139–156, 1978.

[6] M. Coppo, M. Dezani, and B. Venneri. Principal type schemes and lambda-calculus semantics. In *To H.B. Curry. Essays on Combinatory Logic, Lambda-calculus and Formalism*. Academic Press, 1980.

[7] M. Coppo and P. Giannini. Principal types and unification for simple intersection type systems. *Inf. Comput.*, 122:70–96, 1995.

[8] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82*, pages 207–212, 1982.

[10] J. Robie et al. Xquery 3.0: An XML query language (working draft 2010/12/14), 2010. http://www.w3.org/TR/xquery-30/.

[11] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, U. Paris 7, 2004.

[12] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of ACM*, 55(4):1–64, 2008.

[13] T. Jim. What are principal typings and what are they good for? In *POPL '96*, pages 42–53. ACM Press, 1996.

[14] T. Jim. A polar type system. In *ICALP Satellite Workshops*, pages 323–338, 2000.

[15] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL '99*, pages 161–174, 1999.

[16] D. Leivant. Polymorphic type inference. In *POPL '83*. ACM, 1983.

[17] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4(2):258–282, 1982.

[18] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[19] F. Pottier and D. Rémy. The essence of ML type inference. In B.C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[20] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, CU Nijmegen, 1993.

[21] Z. Xu. *Parametric Polymorphism for XML Processing Languages*. PhD thesis, Université Paris Diderot, 2013. Available at http://tel.archives-ouvertes.fr/tel-00858744.

Appendix

A. Implicitly-Typed Calculus

We want sets of type-substitutions to be inferred by the system, not written by the programmer. To this end, we define a calculus without type substitutions (called *implicitly-typed*, in contrast to the calculus in (7) in Section 2.1, which we henceforth call *explicitly-typed*), for which we define a type-substitutions inference system. As explained in Section 3, we do not try to infer decorations in λ-abstractions, and we therefore look for completeness of the type-substitutions inference system with respect to the expressions written according to the following grammar:

$$e \ ::= \ c \mid x \mid (e,e) \mid \pi_i(e) \mid e\,e \mid \lambda^{\wedge_{i\in I} t_i \to s_i} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}.$$

We write $\mathscr{E}_0$ for the set of such expressions. The implicitly-typed calculus defined in this section corresponds to the type-substitution erasures of the expressions of $\mathscr{E}_0$. These are the terms generated by the grammar above without using the last production, that is, without the application of sets of type-substitutions. We then define the type-substitutions inference system by determining where the rule (ALG-INST) have to be used in the typing derivations of explicitly-typed expressions. Finally, we propose an incomplete but more tractable restriction of the type-substitutions inference system, which, we believe, is powerful enough to be used in practice.

A.1 Implicitly-typed Calculus

Definition A.1. *An* implicitly-typed expression a *is an expression without any type substitutions. It is inductively generated by the following grammar:*

$$a \ ::= \ c \mid x \mid (a,a) \mid \pi_i(a) \mid a\,a \mid \lambda^{\wedge_{i\in I} t_i \to s_i} x.a \mid a{\in}t\,?\,a : a$$

where t_i, s_i range over types and $t \in \mathscr{T}_0$ is a ground type. We write $\mathscr{E}_A$ to denote the set of all implicitly-typed expressions.

Clearly, $\mathscr{E}_A$ is a proper subset of $\mathscr{E}_0$.

The erasure of explicitly-typed expressions to implicitly-typed expressions is defined as follows:

Definition A.2. *The* erasure *is the mapping from $\mathscr{E}_0$ to $\mathscr{E}_A$ defined as*

$$\begin{array}{rcl} \mathit{erase}(c) & = & c \\ \mathit{erase}(x) & = & x \\ \mathit{erase}((e_1,e_2)) & = & (\mathit{erase}(e_1), \mathit{erase}(e_2)) \\ \mathit{erase}(\pi_i(e)) & = & \pi_i(\mathit{erase}(e)) \\ \mathit{erase}(\lambda^{\wedge_{i\in I} t_i\to s_i} x.e) & = & \lambda^{\wedge_{i\in I} t_i\to s_i} x.\mathit{erase}(e) \\ \mathit{erase}(e_1 e_2) & = & \mathit{erase}(e_1)\mathit{erase}(e_2) \\ \mathit{erase}(e{\in}t\,?\,e_1 : e_2) & = & \mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2) \\ \mathit{erase}(e[\sigma_j]_{j\in J}) & = & \mathit{erase}(e) \end{array}$$

Prior to introducing the type inference rules, we define a preorder on types, which is similar to the type variable instantiation in ML but with respect to *a set* of type substitutions.

Definition A.3. *Let s and t be two types and Δ a set of type variables. We define the following relations:*

$$[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t \quad \stackrel{\text{def}}{\iff} \quad \bigwedge_{i\in I} s\sigma_i \leq t \text{ and } \forall i{\in}I.\ \sigma_i \sharp \Delta$$

$$s \sqsubseteq_\Delta t \quad \stackrel{\text{def}}{\iff} \quad \exists [\sigma_i]_{i\in I} \text{ such that } [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t$$

We write $s \not\sqsubseteq_\Delta t$ if it does not exist a set of type substitutions $[\sigma_i]_{i\in I}$ such that $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t$. We now prove some properties of the preorder $\sqsubseteq_\Delta$.

Lemma A.4. *Let t_1 and t_2 be two types and Δ a set of type variables. If $t_1 \sqsubseteq_\Delta s_1$ and $t_2 \sqsubseteq_\Delta s_2$, then $(t_1 \wedge t_2) \sqsubseteq_\Delta (s_1 \wedge s_2)$ and $(t_1 \times t_2) \sqsubseteq_\Delta (s_1 \times s_2)$.*

Proof. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t_1 \sqsubseteq_\Delta s_1$ and $[\sigma_{i_2}]_{i_2\in I_2} \Vdash t_2 \sqsubseteq_\Delta s_2$. Then

$$\begin{array}{rcl} \bigwedge_{i\in I_1\cup I_2}(t_1\wedge t_2)\sigma_i & \simeq & (\bigwedge_{i\in I_1\cup I_2} t_1\sigma_i) \wedge (\bigwedge_{i\in I_1\cup I_2} t_2\sigma_i) \\ & \leq & (\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \wedge (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2}) \\ & \leq & s_1 \wedge s_2 \end{array}$$

and

$$\begin{array}{rcl} \bigwedge_{i\in I_1\cup I_2}(t_1\times t_2)\sigma_i & \simeq & ((\bigwedge_{i\in I_1\cup I_2} t_1\sigma_i) \times (\bigwedge_{i\in I_1\cup I_2} t_2\sigma_i)) \\ & \leq & ((\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \times (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2})) \\ & \leq & (s_1 \times s_2) \end{array}$$

□

Lemma A.5. *Let t_1 and t_2 be two types and Δ a set of type variables such that $(\mathsf{var}(t_1) \setminus \Delta) \cap (\mathsf{var}(t_2) \setminus \Delta) = \emptyset$. If $t_1 \sqsubseteq_\Delta s_1$ and $t_2 \sqsubseteq_\Delta s_2$, then $t_1 \vee t_2 \sqsubseteq_\Delta s_1 \vee s_2$.*

Proof. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t_1 \sqsubseteq_\Delta s_1$ and $[\sigma_{i_2}]_{i_2\in I_2} \Vdash t_2 \sqsubseteq_\Delta s_2$. Then we construct another set of type substitutions $[\sigma_{i_1,i_2}]_{i_1\in I_1, i_2\in I_2}$ such that

$$\sigma_{i_1,i_2}(\alpha) = \begin{cases} \sigma_{i_1}(\alpha) & \text{if } \alpha \in (\mathsf{var}(t_1) \setminus \Delta) \\ \sigma_{i_2}(\alpha) & \text{if } \alpha \in (\mathsf{var}(t_2) \setminus \Delta) \\ \alpha & \text{otherwise} \end{cases}$$

So we have

$$\begin{array}{rcl} \bigwedge_{i_1\in I_1, i_2\in I_2}(t_1 \vee t_2)\sigma_{i_1,i_2} & \simeq & \bigwedge_{i_1\in I_1}(\bigwedge_{i_2\in I_2}(t_1 \vee t_2)\sigma_{i_1,i_2}) \\ & \simeq & \bigwedge_{i_1\in I_1}(\bigwedge_{i_2\in I_2}((t_1\sigma_{i_1,i_2}) \vee (t_2\sigma_{i_1,i_2}))) \\ & \simeq & \bigwedge_{i_1\in I_1}(\bigwedge_{i_2\in I_2}(t_1\sigma_{i_1} \vee t_2\sigma_{i_2})) \\ & \simeq & \bigwedge_{i_1\in I_1}(t_1\sigma_{i_1} \vee (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2})) \\ & \simeq & (\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \vee (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2}) \\ & \leq & s_1 \vee s_2 \end{array}$$

□

Notice that two successive instantiations can be safely merged into one (see Lemma A.6). Henceforth, we assume that there are no successive instantiations in a given derivation tree. In order to guess where to insert sets of type-substitutions in an implicitly-typed expression, we consider each typing rule of the explicitly-typed calculus used in conjunction with the instantiation rule (ALG-INST). If instantiation can be moved through a given typing rule without affecting typability or changing the result type, then it is not necessary to infer type substitutions at the level of this rule.

Lemma A.6. *Let e be an explicitly-typed expression and $[\sigma_i]_{i\in I}$, $[\sigma_j]_{j\in J}$ two sets of type substitutions. Then*

$$\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : t \iff \Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) : t$$

Proof. $\Rightarrow$: consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e : s} \quad \sigma_i \sharp \Delta}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e[\sigma_i]_{i\in I} : \bigwedge_{i\in I} s\sigma_i} \quad \sigma_j \sharp \Delta}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(\bigwedge_{i\in I} s\sigma_i)\sigma_j}$$

As $\sigma_i \sharp \Delta$, $\sigma_j \sharp \Delta$ and $\mathsf{dom}(\sigma_j \circ \sigma_i) = \mathsf{dom}(\sigma_i) \cup \mathsf{dom}(\sigma_j)$, we have $\sigma_j \circ \sigma_i \sharp \Delta$. Then by (ALG-INST), we have $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e([\sigma_j \circ \sigma_i]_{j\in J, i\in I}) : \bigwedge_{j\in J, i\in I} s(\sigma_j \circ \sigma_i)$, that is $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) : \bigwedge_{j\in J}(\bigwedge_{i\in I} s\sigma_i)\sigma_j$.

$\Leftarrow$: consider the following derivation:

$$\dfrac{\dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e : s} \quad \sigma_j \circ \sigma_i \sharp \Delta}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) : \bigwedge_{j\in J, i\in I} s(\sigma_j \circ \sigma_i)}$$

As $\sigma_j \circ \sigma_i \sharp \Delta$ and $\mathsf{dom}(\sigma_j \circ \sigma_i) = \mathsf{dom}(\sigma_i) \cup \mathsf{dom}(\sigma_j)$, we have $\sigma_i \sharp \Delta$ and $\sigma_j \sharp \Delta$. Then applying the rule (ALG-INST) twice, we have $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(\bigwedge_{i\in I} s\sigma_i)\sigma_j$, that is $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : \bigwedge_{j\in J, i\in I} s(\sigma_j \circ \sigma_i)$.

□

First of all, consider a typing derivation ending with (ALG-PAIR) where both of its sub-derivations end with (ALG-INST)[7]:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e_1 : t_1} \quad \forall j_1 \in J_1.\, \sigma_{j_1} \sharp \Delta}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} t_1\sigma_{j_1}} \quad \dfrac{\dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e_2 : t_2} \quad \forall j_2 \in J_2.\, \sigma_{j_2} \sharp \Delta}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e_2[\sigma_{j_2}]_{j_2\in J_2} : \bigwedge_{j_2\in J_2} t_1\sigma_{j_2}}}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e_1[\sigma_{j_1}]_{j_1\in J_1}, e_2[\sigma_{j_2}]_{j_2\in J_2}) : (\bigwedge_{j_1\in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2\in J_2} t_1\sigma_{j_2})}$$

We rewrite such a derivation as follows:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e_1 : t_1} \quad \dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e_2 : t_2}}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2} \quad \forall j \in J_1 \cup J_2.\, \sigma_j \sharp \Delta}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} (e_1, e_2)[\sigma_j]_{j\in J_1\cup J_2} : \bigwedge_{j\in J_1\cup J_2}(t_1 \times t_2)\sigma_j}$$

Clearly, $\bigwedge_{j\in J_1\cup J_2}(t_1 \times t_2)\sigma_j \leq (\bigwedge_{j_1\in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2\in J_2} t_1\sigma_{j_2})$. Then by subsumption we can deduce that $(e_1, e_2)[\sigma_j]_{j\in J_1\cup J_2}$ also has the type $(\bigwedge_{j_1\in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2\in J_2} t_1\sigma_{j_2})$. Therefore, we can disregard the sets of type substitutions that are applied inside a pair, since inferring them outside the pair is equivalent. Hence, we can use the following inference rule for pairs.

$$\dfrac{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} a_1 : t_1 \quad \Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} a_2 : t_2}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}$$

[7] If one of the sub-derivations does not end with (ALG-INST), we can apply a trivial instance of (ALG-INST) with an identity substitution σ_{id}.

Next, consider a derivation ending of (ALG-PROJ) whose premise is derived by (ALG-INST):

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} e : t} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j} \qquad (\bigwedge_{j\in J} t\sigma_j) \leq \mathbb{1} \times \mathbb{1}}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} \pi_i(e[\sigma_j]_{j\in J}) : \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)}$$

According to Lemma C.8 in the companion paper [3], we have $\boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j) \leq \bigwedge_{j\in J} \boldsymbol{\pi}_i(t)\sigma_j$, but the converse does not necessarily hold. For example, $\boldsymbol{\pi}_1(((t_1 \times t_2) \vee (s_1 \times \alpha \setminus s_2))\{s_2/_\alpha\}) = t_1\{s_2/_\alpha\}$ while $(\boldsymbol{\pi}_1((t_1 \times t_2) \vee (s_1 \times \alpha \setminus s_2)))\{s_2/_\alpha\} = (t_1 \vee s_1)\{s_2/_\alpha\}$. So we cannot exchange the instantiation and projection rules without losing completeness. However, as $(\bigwedge_{j\in J} t\sigma_j) \leq \mathbb{1} \times \mathbb{1}$ and $\forall j \in J.\ \sigma_j \sharp \Delta$, we have $t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$. This indicates that for an implicitly-typed expression $\pi_i(a)$, if the inferred type for a is t and there exists $[\sigma_j]_{j\in J}$ such that $[\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$, then we infer the type $\boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)$ for $\pi_i(a)$. Let $\amalg^i_\Delta(t)$ denote the set of such result types, that is,

$$\amalg^i_\Delta(t) = \{u \mid [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}, u = \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)\}$$

Formally, we have the following inference rule for projections

$$\dfrac{\Delta \,⨾\, \Gamma \vdash_{\mathscr{I}} a : t \quad u \in \amalg^i_\Delta(t)}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}$$

The following lemma tells us that $\amalg^i_\Delta(t)$ is "morally" closed by intersection, in the sense that if we take two solutions in $\amalg^i_\Delta(t)$, then we can take also their intersection as a solution, since there always exists in $\amalg^i_\Delta(t)$ a solution at least as precise as their intersection.

Lemma A.7. *Let t be a type and Δ a set of type variables. If $u_1 \in \amalg^i_\Delta(t)$ and $u_2 \in \amalg^i_\Delta(t)$, then $\exists u_0 \in \amalg^i_\Delta(t).\ u_0 \leq u_1 \wedge u_2$.*

Proof. Let $[\sigma_{j_k}]_{j_k\in J_k} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$ and $u_k = \boldsymbol{\pi}_i(\bigwedge_{j_k\in J_k} t\sigma_{j_k})$ for $k = 1, 2$. Then $[\sigma_j]_{j\in J_1\cup J_2} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$. So $\boldsymbol{\pi}_i(\bigwedge_{j\in J_1\cup J_2} t\sigma_j) \in \amalg^i_\Delta(t)$. Moreover, by Lemma C.6 in the companion paper [3], we have

$$\boldsymbol{\pi}_i(\bigwedge_{j\in J_1\cup J_2} t\sigma_j) \leq \boldsymbol{\pi}_i(\bigwedge_{j_1\in J_1} t\sigma_{j_1}) \wedge \boldsymbol{\pi}_i(\bigwedge_{j_2\in J_2} t\sigma_{j_2}) = u_1 \wedge u_2$$

□

Since we only consider λ-abstractions with empty decorations, we can consider the following simplified version of (ALG-ABSTR) that does not use relabeling

$$\dfrac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i\in I}(t_i \to s_i)) \,⨾\, \Gamma, x : t_i \vdash_{\mathscr{A}} e : s_i' \text{ and } s_i' \leq s_i}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.e : \bigwedge_{i\in I}(t_i \to s_i)} \text{(ALG-ABSTR0)}$$

Suppose the last rule used in the sub-derivations is (ALG-INST).

$$\dfrac{\forall i \in I.\ \left\{ \begin{array}{l} \dfrac{\dfrac{\cdots}{\Delta' \,⨾\, \Gamma, x : t_i \vdash_{\mathscr{A}} e : s_i'} \quad \forall j \in J.\ \sigma_j \sharp \Delta'}{\Delta' \,⨾\, \Gamma, x : t_i \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s_i'\sigma_j} \\ \bigwedge_{j\in J} s_i'\sigma_j \leq s_i \end{array} \right. \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I}(t_i \to s_i))}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.e[\sigma_j]_{j\in J} : \bigwedge_{i\in I}(t_i \to s_i)}$$

From the side conditions, we deduce that $s_i' \sqsubseteq_{\Delta'} s_i$ for all $i \in I$. Instantiation may be necessary to bridge the gap between the computed type s_i' for e and the type s_i required by the interface, so inferring type substitutions at this stage is mandatory. Therefore, we propose the following inference rule for abstractions.

$$\dfrac{\forall i \in I.\ \left\{ \begin{array}{l} \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,⨾\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s_i' \\ s_i' \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)} s_i \end{array} \right.}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}$$

In the application case, suppose both sub-derivations end with (ALG-INST):

$$\dfrac{\begin{array}{ll} \dfrac{\dfrac{\cdots}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} e_1 : t} \quad \forall j_1 \in J_1.\ \sigma_{j_1} \sharp \Delta}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} t\sigma_{j_1}} & \dfrac{\dfrac{\cdots}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} e_2 : s} \quad \forall j_2 \in J_2.\ \sigma_{j_2} \sharp \Delta}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} e_2[\sigma_{j_2}]_{j_2\in J_2} : \bigwedge_{j_2\in J_2} s\sigma_{j_2}} \\ \bigwedge_{j_1\in J_1} t\sigma_{j_1} \leq \mathbb{0} \to \mathbb{1} & \bigwedge_{j_2\in J_2} s\sigma_{j_2} \leq \mathsf{dom}(\bigwedge_{j_1\in J_1} t\sigma_{j_1}) \end{array}}{\Delta \,⨾\, \Gamma \vdash_{\mathscr{A}} (e_1[\sigma_{j_1}]_{j_1\in J_1})(e_2[\sigma_{j_2}]_{j_2\in J_2}) : (\bigwedge_{j_1\in J_1} t\sigma_{j_1}) \cdot (\bigwedge_{j_2\in J_2} s\sigma_{j_2})}$$

Instantiation may be needed to bridge the gap between the (domain of the) function type and its argument (e.g., to apply $\lambda^{\alpha\to\alpha}x.x$ to $\mathtt{42}$). The side conditions imply that $[\sigma_{j_1}]_{j_1\in J_1} \Vdash t \sqsubseteq_\Delta \mathbb{0} \to \mathbb{1}$ and $[\sigma_{j_2}]_{j_2\in J_2} \Vdash s \sqsubseteq_\Delta \mathsf{dom}(\bigwedge_{j_1\in J_1} t\sigma_{j_1})$. Therefore, given an implicitly-typed application $a_1 a_2$ where

a_1 and a_2 are typed with t and s respectively, we have to find two sets of substitutions $[\sigma_{j_1}]_{j_1\in J_1}$ and $[\sigma_{j_2}]_{j_2\in J_2}$ verifying the above preorder relations to be able to type the application. If such sets of substitutions exist, then we can type the application with $(\bigwedge_{j_1\in J_1} t\sigma_{j_1})\cdot(\bigwedge_{j_2\in J_2} s\sigma_{j_2})$. Let $t \bullet_\Delta s$ denote the set of such result types, that is,

$$t \bullet_\Delta s \overset{\text{def}}{=} \left\{ u \;\middle|\; \begin{array}{l} [\sigma_i]_{i\in I} \Vdash t \sqsubseteq_\Delta \mathbb{0}\to\mathbb{1} \\ [\sigma_j]_{j\in J} \Vdash s \sqsubseteq_\Delta \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) \\ u = \bigwedge_{i\in I} t\sigma_i \cdot \bigwedge_{j\in J} s\sigma_j \end{array} \right\}$$

This set is closed under intersection (see Lemma A.8). Formally, we get the following inference rule for applications

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : t \quad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : s \quad u \in t \bullet_\Delta s}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u}$$

Lemma A.8. *Let t, s be two types and Δ a set of type variables. If $u_1 \in t \bullet_\Delta s$ and $u_2 \in t \bullet_\Delta s$, then $\exists u_0 \in t \bullet_\Delta s.\ u_0 \leq u_1 \wedge u_2$.*

Proof. Let $u_k = (\bigwedge_{i_k\in I_k} t\sigma_{i_k})\cdot(\bigwedge_{j_k\in J_k} s\sigma_{j_k})$ for $k = 1, 2$. According to Lemma C.18 in the companion paper [3], we have $(\bigwedge_{i\in I_1\cup I_2} t\sigma_i)\cdot(\bigwedge_{j\in J_1\cup J_2} s\sigma_j) \in t \bullet_\Delta s$ and $(\bigwedge_{i\in I_1\cup I_2} t\sigma_i)\cdot(\bigwedge_{j\in J_1\cup J_2} s\sigma_j) \leq \bigwedge_{k=1,2}(\bigwedge_{i_k\in I_k} t\sigma_{i_k})\cdot(\bigwedge_{j_k\in J_k} s\sigma_{j_k}) = u_1 \wedge u_2$. □

For type cases, we distinguish the four possible behaviours: (i) no branch is selected, (ii) the first branch is selected, (iii) the second branch is selected, and (iv) both branches are selected. In all these cases, we assume that the premises end with (ALG-INST). In case (i), we have the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \quad \bigwedge_{j\in J} t'\sigma_j \leq \mathbb{0}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}){\in}t\,?\,e_1 : e_2 : \mathbb{0}}$$

Clearly, the side conditions implies $t' \sqsubseteq_\Delta \mathbb{0}$. The type inference rule for implicitly-typed expressions corresponding to this case is then

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \quad t' \sqsubseteq_\Delta \mathbb{0}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \mathbb{0}}$$

For case (ii), consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t'} \quad \sigma_j \sharp \Delta}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \quad \bigwedge_{j\in J} t'\sigma_j \leq t \quad \dfrac{\dfrac{\cdots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1} \quad \sigma_{j_1} \sharp \Delta}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}){\in}t\,?\,(e_1[\sigma_{j_1}]_{j_1\in J_1}) : e_2 : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}$$

First, such a derivation can be rewritten as

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t'} \quad \sigma_j \sharp \Delta}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \quad \bigwedge_{j\in J} t'\sigma_j \leq t \quad \dfrac{\cdots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}){\in}t\,?\,e_1 : e_2 : s_1} \quad \sigma_{j_1} \sharp \Delta}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} ((e[\sigma_j]_{j\in J}){\in}t\,?\,e_1 : e_2)[\sigma_{j_1}]_{j_1\in J_1}) : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}$$

This indicates that it is equivalent to apply the substitutions $[\sigma_{j_1}]_{j_1\in J_1}$ to e_1 or to the whole type case expression. Looking at the derivation for e, for the first branch to be selected we must have $t' \sqsubseteq_\Delta t$. Note that if $t' \sqsubseteq_\Delta \neg t$, we would have $t' \sqsubseteq_\Delta \mathbb{0}$ by Lemma A.4, and no branch would be selected. Consequently, the type inference rule for a type case where the first branch is selected is as follows.

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \quad t' \sqsubseteq_\Delta t \quad t' \not\sqsubseteq_\Delta \neg t \quad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : s}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s}$$

Case (iii) is similar to case (ii) where t is replaced by $\neg t$.

At last, consider a derivation of Case (iv):

$$\dfrac{\left\{\begin{array}{l}\dfrac{\dfrac{\ldots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq \neg t \quad \text{and} \quad \dfrac{\dfrac{\ldots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_1 : s_1} \quad \forall j_1 \in J_1.\ \sigma_{j_1} \sharp \Delta}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq t \quad \text{and} \quad \dfrac{\dfrac{\ldots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_2 : s_2} \quad \forall j_2 \in J_2.\ \sigma_{j_2} \sharp \Delta}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_2[\sigma_{j_2}]_{j_2\in J_2} : \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}}\end{array}\right.}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}{\in}t\,?\,(e_1[\sigma_{j_1}]_{j_1\in J_1}) : (e_2[\sigma_{j_2}]_{j_2\in J_2})) : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}}$$

Using α-conversion if necessary, we can assume that the polymorphic type variables of e_1 and e_2 are distinct, and therefore we have $(\mathsf{var}(s_1) \setminus \Delta) \cap (\mathsf{var}(s_2) \setminus \Delta) = \emptyset$. According to Lemma A.5, we get $s_1 \vee s_2 \sqsubseteq_\Delta \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. Let $[\sigma_{j_{12}}]_{j_{12}\in J_{12}} \Vdash s_1 \vee s_2 \sqsubseteq_\Delta \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. We can rewrite this derivation as

$$\dfrac{\dfrac{\left\{\begin{array}{l}\dfrac{\dfrac{\ldots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq \neg t \quad \text{and} \quad \dfrac{\ldots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_1 : s_1} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq t \quad \text{and} \quad \dfrac{\ldots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_2 : s_2}\end{array}\right.}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2} \qquad \forall j_{12} \in J_{12}.\ \sigma_{j_{12}} \sharp \Delta}{\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} ((e[\sigma_j]_{j\in J}{\in}t\,?\,e_1 : e_2)[\sigma_{j_{12}}]_{j_{12}\in J_{12}}) : \bigwedge_{j_{12}\in J_{12}} (s_1 \vee s_2)\sigma_{j_{12}}}$$

As $\bigwedge_{j_{12}\in J_{12}} (s_1 \vee s_2)\sigma_{j_{12}} \leq \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$, by subsumption, we can deduce that $(e[\sigma_j]_{j\in J}{\in}t\,?\,e_1 : e_2)[\sigma_{j_{12}}]_{j_{12}\in J_{12}}$ has the type $\bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. Hence, we eliminate the substitutions that are applied to these two branches.

We now consider the part of the derivation tree which concerns e. With the specific set of substitutions $[\sigma_j]_{j\in J}$, we have $\bigwedge_{j\in J} t'\sigma_j \not\leq \neg t$ and $\bigwedge_{j\in J} t'\sigma_j \not\leq t$, but it does not mean that we have $t' \not\sqsubseteq_\Delta t$ and $t' \not\sqsubseteq_\Delta \neg t$ in general. If $t' \sqsubseteq_\Delta t$ and/or $t' \sqsubseteq_\Delta \neg t$ hold, then we are in one of the previous cases $(i) - (iii)$ (i.e., we type-check at most one branch), and the inferred result type for the whole type case belongs to $\mathbb{0}$, s_1 or s_2. We can then use subsumption to type the whole type-case expression with $s_1 \vee s_2$. Otherwise, both branches are type-checked, and we deduce the corresponding inference rule as follows.

$$\dfrac{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a : t' \quad \left\{\begin{array}{lll} t' \not\sqsubseteq_\Delta \neg t & \text{and} & \Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_1 : s_1 \\ t' \not\sqsubseteq_\Delta t & \text{and} & \Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_2 : s_2 \end{array}\right.}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}$$

From the study above, we deduce the type-substitution inference rules for implicitly-typed expressions given in Figure 5, which are the same as those in Section 3 except for the rules for products.

A.2 Soundness and Completeness

We now prove that the inference rules of the implicitly-typed calculus given in Figure 5 are sound and complete with respect to the type system of the explicitly-typed calculus (*i.e.*, Figure 1 extended with the standard rules for products).

To construct an explicitly-typed expression from an implicitly-typed one a, we have to insert sets of substitutions in a each time a preorder check is performed in the rules of Figure 5. For an abstraction $\lambda^{\wedge_{i\in I} t_i \to s_i} x.a$, different sets of substitutions may be constructed when type checking the body under the different hypotheses $x : t_i$. For example, let $a = \lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})} x.(\lambda^{\alpha\to\alpha} y.y)x$. When a is type-checked against $\mathtt{Int} \to \mathtt{Int}$, that is, x is assumed to have type $\mathtt{Int}$, we infer the type substitution $\{\mathtt{Int}/_\alpha\}$ for $(\lambda^{\alpha\to\alpha} y.y)$. Similarly, we infer $\{\mathtt{Bool}/_\alpha\}$ for $(\lambda^{\alpha\to\alpha} y.y)$, when a is type-checked against $\mathtt{Bool} \to \mathtt{Bool}$. We have to collect these two different substitutions when constructing the explicitly-typed expression e which corresponds to a. To this end, we introduce an intersection operator $e \sqcap e'$ of expressions which is defined only for pair of expressions that have similar structure but different type substitutions. For example, the intersection of $(\lambda^{\alpha\to\alpha} y.y)[\{\mathtt{Int}/_\alpha\}]x$ and $(\lambda^{\alpha\to\alpha} y.y)[\{\mathtt{Bool}/_\alpha\}]x$ will be $(\lambda^{\alpha\to\alpha} y.y)[\{\mathtt{Int}/_\alpha\}, \{\mathtt{Bool}/_\alpha\}]x$.

$$\frac{}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} c : b_c}(\text{Inf-const}) \qquad \frac{}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} x : \Gamma(x)}(\text{Inf-var})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : t_1 \qquad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : t_2}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}(\text{Inf-pair}) \qquad \frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t \qquad u \in \amalg^i_{\Delta}(t)}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}(\text{Inf-proj})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : t \qquad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : s \qquad u \in t \bullet_{\Delta} s}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u}(\text{Inf-appl})$$

$$\frac{\forall i \in I. \begin{cases} \Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i) \,⨟\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i)} s_i \end{cases}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.a : \bigwedge_{i \in I} t_i \to s_i}(\text{Inf-abstr})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad t' \sqsubseteq_{\Delta} \mathbb{0}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a \in t \,?\, a_1 : a_2) : \mathbb{0}}(\text{Inf-case-none})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad t' \sqsubseteq_{\Delta} t \qquad t' \not\sqsubseteq_{\Delta} \neg t \qquad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : s}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a \in t \,?\, a_1 : a_2) : s}(\text{Inf-case-fst})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad t' \sqsubseteq_{\Delta} \neg t \qquad t' \not\sqsubseteq_{\Delta} t \qquad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : s}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a \in t \,?\, a_1 : a_2) : s}(\text{Inf-case-snd})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad \begin{cases} t' \not\sqsubseteq_{\Delta} \neg t & \text{and} \quad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : s_1 \\ t' \not\sqsubseteq_{\Delta} t & \text{and} \quad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : s_2 \end{cases}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a \in t \,?\, a_1 : a_2) : s_1 \vee s_2}(\text{Inf-case-both})$$

Figure 5. Type-substitution inference rules

Definition A.9. *Let $e, e' \in \mathscr{E}_0$ be two expressions. Their* intersection $e \sqcap e'$ *is defined by induction as:*

$$\begin{array}{rcll}
c \sqcap c &=& c & \\
x \sqcap x &=& x & \\
(e_1, e_2) \sqcap (e'_1, e'_2) &=& ((e_1 \sqcap e'_1), (e_2 \sqcap e'_2)) & \\
\pi_i(e) \sqcap \pi_i(e') &=& \pi_i(e \sqcap e') & \\
e_1 e_2 \sqcap e'_1 e'_2 &=& (e_1 \sqcap e'_1)(e_2 \sqcap e'_2) & \\
(\lambda^{\wedge_{i \in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i \in I} t_i \to s_i} x.e') &=& \lambda^{\wedge_{i \in I} t_i \to s_i} x.(e \sqcap e') & \\
(e_0 \in t \,?\, e_1 : e_2) \sqcap (e'_0 \in t \,?\, e'_1 : e'_2) &=& e_0 \sqcap e'_0 \in t \,?\, e_1 \sqcap e'_1 : e_2 \sqcap e'_2 & \\
(e_1[\sigma_j]_{j \in J}) \sqcap (e'_1[\sigma_j]_{j \in J'}) &=& (e_1 \sqcap e'_1)[\sigma_j]_{j \in J \cup J'} & \\
e \sqcap (e'_1[\sigma_j]_{j \in J'}) &=& (e[\sigma_{id}]) \sqcap (e'_1[\sigma_j]_{j \in J'}) & \textit{if } e \neq e_1[\sigma_j]_{j \in J} \\
(e_1[\sigma_j]_{j \in J}) \sqcap e' &=& (e_1[\sigma_j]_{j \in J}) \sqcap (e'[\sigma_{id}]) & \textit{if } e' \neq e'_1[\sigma_j]_{j \in J'}
\end{array}$$

where σ_{id} is the identity type substitution and is undefined otherwise.

The intersection of a same constant or a same variable is the constant or the variable itself. If e and e' have the same form, then their intersection is defined if their intersections of the corresponding sub-expressions are defined. In particular when e is form of $e_1[\sigma_j]_{j \in J}$ and e' is form of $e'_1[\sigma_j]_{j \in J'}$, we merge the sets of substitutions $[\sigma_j]_{j \in J}$ and $[\sigma_j]_{j \in J'}$ into one set $[\sigma_j]_{j \in J \cup J'}$. Otherwise, e and e' have different forms. The only possible case for their intersection is they have similar structure but one with instantiations and the other without (*i.e.*, $e = e_1[\sigma_j]_{j \in J}, e' \neq e'_1[\sigma_j]_{j \in J'}$ or $e \neq e_1[\sigma_j]_{j \in J}, e' = e'_1[\sigma_j]_{j \in J'}$). To keep the inferred information and reuse the defined cases above, we add the identity substitution σ_{id} to the one without substitutions (*i.e.*, $e'[\sigma_{id}]$ or $e[\sigma_{id}]$) to make them have the same form. Note that σ_{id} is important so as to keep the information we have inferred. Let us infer the substitutions for the abstraction $\lambda^{(t_1 \to s_1) \wedge (t_2 \to s_2)} x.e$. Assume that we have inferred some substitutions for the body e under $t_1 \to s_1$ and $t_2 \to s_2$ respectively, yielding two explicitly-typed expressions e_1 and $e_2[\sigma_j]_{j \in J}$. If we did not add the identity substitution σ_{id} for the intersection of e_1 and $e_2[\sigma_j]_{j \in J}$, that is, $e_1 \sqcap (e_2[\sigma_j]_{j \in J})$ were $(e_1 \sqcap e_2)[\sigma_j]_{j \in J}$ rather than $(e_1 \sqcap e_2)([\sigma_{id}] \cup [\sigma_j]_{j \in J})$, then the substitutions we inferred under $t_1 \to s_1$ would be lost since they may be modified by $[\sigma_j]_{j \in J}$.

Lemma A.10. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, then $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$.*

Proof. By induction on the structures of e and e'. Because $erase(e) = erase(e')$, the two expressions have the same structure up to their sets of type substitutions.

$\underline{c, c}$**:** straightforward.
$\underline{x, x}$**:** straightforward.
$\underline{(e_1, e_2), (e_1', e_2')}$**:** we have $erase(e_i) = erase(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $erase(e_i \sqcap e_i') = erase(e_i) = erase(e_i')$. Therefore $(e_1, e_2) \sqcap (e_1', e_2')$ exists and

$$\begin{array}{rl} erase((e_1, e_2) \sqcap (e_1', e_2')) = & erase(((e_1 \sqcap e_1'), (e_2 \sqcap e_2'))) \\ = & (erase(e_1 \sqcap e_1'), erase(e_2 \sqcap e_2')) \\ = & (erase(e_1), erase(e_2)) \\ = & erase((e_1, e_2)) \end{array}$$

Similarly, we also have $erase((e_1, e_2) \sqcap (e_1', e_2')) = erase((e_1', e_2'))$.
$\underline{\pi_i(e), \pi_i(e')}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $\pi_i(e) \sqcap \pi_i(e')$ exists and

$$\begin{array}{rl} erase(\pi_i(e) \sqcap \pi_i(e')) = & erase(\pi_i(e \sqcap e')) \\ = & \pi_i(erase(e \sqcap e')) \\ = & \pi_i(erase(e)) \\ = & erase(\pi_i(e)) \end{array}$$

Similarly, we also have $erase(\pi_i(e) \sqcap \pi_i(e')) = erase(\pi_i(e'))$.
$\underline{e_1 e_2, e_1' e_2'}$**:** we have $erase(e_i) = erase(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $erase(e_i \sqcap e_i') = erase(e_i) = erase(e_i')$. Therefore $e_1 e_2 \sqcap e_1' e_2'$ exists and

$$\begin{array}{rl} erase((e_1 e_2) \sqcap (e_1' e_2')) = & erase((e_1 \sqcap e_1')(e_2 \sqcap e_2')) \\ = & erase(e_1 \sqcap e_1') erase(e_2 \sqcap e_2') \\ = & erase(e_1) erase(e_2) \\ = & erase(e_1 e_2) \end{array}$$

Similarly, we also have $erase((e_1 e_2) \sqcap (e_1' e_2')) = erase(e_1' e_2')$.
$\underline{\lambda^{\wedge_{i \in I} t_i \to s_i} x.e, \lambda^{\wedge_{i \in I} t_i \to s_i} x.e'}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $(\lambda^{\wedge_{i \in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i \in I} t_i \to s_i} x.e')$ exists and

$$\begin{array}{rl} erase((\lambda^{\wedge_{i \in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i \in I} t_i \to s_i} x.e')) = & erase(\lambda^{\wedge_{i \in I} t_i \to s_i} x.(e \sqcap e')) \\ = & \lambda^{\wedge_{i \in I} t_i \to s_i} x.erase((e \sqcap e')) \\ = & \lambda^{\wedge_{i \in I} t_i \to s_i} x.erase(e) \\ = & erase(\lambda^{\wedge_{i \in I} t_i \to s_i} x.e) \end{array}$$

Similarly, we also have

$$erase((\lambda^{\wedge_{i \in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i \in I} t_i \to s_i} x.e')) = erase(\lambda^{\wedge_{i \in I} t_i \to s_i} x.e')$$

$\underline{e_0 \in t \,?\, e_1 : e_2, e_0' \in t \,?\, e_1' : e_2'}$**:** we have $erase(e_i) = erase(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $erase(e_i \sqcap e_i') = erase(e_i) = erase(e_i')$. Therefore $(e_0 \in t \,?\, e_1 : e_2) \sqcap (e_0' \in t \,?\, e_1' : e_2')$ exists and

$$\begin{array}{rl} erase((e_0 \in t \,?\, e_1 : e_2) \sqcap (e_0' \in t \,?\, e_1' : e_2')) = & erase((e_0 \sqcap e_0') \in t \,?\, (e_1 \sqcap e_1') : (e_2 \sqcap e_2')) \\ = & erase(e_0 \sqcap e_0') \in t \,?\, erase(e_1 \sqcap e_1') : erase(e_2 \sqcap e_2') \\ = & erase(e_0) \in t \,?\, erase(e_1) : erase(e_2) \\ = & erase(e_0 \in t \,?\, e_1 : e_2) \end{array}$$

Similarly, we also have

$$erase((e_0 \in t \,?\, e_1 : e_2) \sqcap (e_0' \in t \,?\, e_1' : e_2')) = erase(e_0' \in t \,?\, e_1' : e_2')$$

$\underline{e[\sigma_j]_{j \in J}, e'[\sigma_j]_{j \in J'}}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $(e[\sigma_j]_{j \in J}) \sqcap (e'[\sigma_j]_{j \in J'})$ exists and

$$\begin{array}{rl} erase((e[\sigma_j]_{j \in J}) \sqcap (e'[\sigma_j]_{j \in J'})) = & erase((e \sqcap e')[\sigma_j]_{j \in J \cup J'}) \\ = & erase(e \sqcap e') \\ = & erase(e) \\ = & erase(e[\sigma_j]_{j \in J}) \end{array}$$

Similarly, we also have $erase((e[\sigma_j]_{j \in J}) \sqcap (e'[\sigma_j]_{j \in J'})) = erase(e'[\sigma_j]_{j \in J'})$.
$\underline{e, e'[\sigma_j]_{j \in J'}}$**:** a special case of $e[\sigma_j]_{j \in J}$ and $e'[\sigma_j]_{j \in J'}$ where $[\sigma_j]_{j \in J} = [\sigma_{id}]$.
$\underline{e[\sigma_j]_{j \in J}, e'}$**:** a special case of $e[\sigma_j]_{j \in J}$ and $e'[\sigma_j]_{j \in J'}$ where $[\sigma_j]_{j \in J'} = [\sigma_{id}]$.

□

Lemma A.11. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, $\Delta \,ĝ\, \Gamma \vdash e : t$, $\Delta' \,ĝ\, \Gamma' \vdash e' : t'$, $e \sharp \Delta'$ and $e' \sharp \Delta$, then $\Delta \,ĝ\, \Gamma \vdash e \sqcap e' : t$ and $\Delta' \,ĝ\, \Gamma' \vdash e \sqcap e' : t'$.*

Proof. According to Lemma A.10, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. We only prove $\Delta \mathbin{⨟} \Gamma \vdash e \sqcap e' : t$ as the other case is similar. For simplicity, we just consider one set of type substitutions. For several sets of type substitutions, we can either compose them or apply (*instinter*) several times. The proof proceeds by induction on $\Delta \mathbin{⨟} \Gamma \vdash e : t$.

(*const*): $\Delta \mathbin{⨟} \Gamma \vdash c : b_c$. As $erase(e') = c$, e' is either c or $c[\sigma_j]_{j\in J}$. If $e' = c$, then $e \sqcap e' = c$, and the result follows straightforwardly. Otherwise, we have $e \sqcap e' = c[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \mathbin{⨟} \Gamma \vdash c[\sigma_{id}, \sigma_j]_{j\in J} : b_c \wedge \bigwedge_{j\in J} b_c\sigma_j$, that is, $\Delta \mathbin{⨟} \Gamma \vdash c[\sigma_{id}, \sigma_j]_{j\in J} : b_c$.

(*var*): $\Gamma \vdash x : \Gamma(x)$. As $erase(e') = x$, e' is either x or $x[\sigma_j]_{j\in J}$. If $e' = x$, then $e \sqcap e' = x$, and the result follows straightforwardly. Otherwise, we have $e \sqcap e' = x[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \mathbin{⨟} \Gamma \vdash x[\sigma_{id}, \sigma_j]_{j\in J} : \Gamma(x) \wedge \bigwedge_{j\in J} \Gamma(x)\sigma_j$, that is, $\Delta \mathbin{⨟} \Gamma \vdash x[\sigma_{id}, \sigma_j]_{j\in J} : \Gamma(x)$.

(*pair*): consider the following derivation:

$$\dfrac{\dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_1 : t_1} \quad \dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_2 : t_2}}{\Delta \mathbin{⨟} \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

As $erase(e') = (erase(e_1), erase(e_2))$, e' is either (e_1', e_2') or $(e_1', e_2')[\sigma_j]_{j\in J}$ such that $erase(e_i') = erase(e_i)$. By induction, we have $\Delta \mathbin{⨟} \Gamma \vdash e_i \sqcap e_i' : t_i$. Then by (*pair*), we have $\Delta \mathbin{⨟} \Gamma \vdash (e_1 \sqcap e_1', e_2 \sqcap e_2') : (t_1 \times t_2)$. If $e' = (e_1', e_2')$, then $e \sqcap e' = (e_1 \sqcap e_1', e_2 \sqcap e_2')$. So the result follows.
Otherwise, $e \sqcap e' = (e_1 \sqcap e_1', e_2 \sqcap e_2')[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \mathbin{⨟} \Gamma \vdash (e_1 \sqcap e_1', e_2 \sqcap e_2')[\sigma_{id}, \sigma_j]_{j\in J} : (t_1 \times t_2) \wedge \bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$. Finally, by (*subsum*), we get $\Delta \mathbin{⨟} \Gamma \vdash (e_1 \sqcap e_1', e_2 \sqcap e_2')[\sigma_{id}, \sigma_j]_{j\in J} : (t_1 \times t_2)$.

(*proj*): consider the following derivation:

$$\dfrac{\dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_0 : t_1 \times t_2}}{\Delta \mathbin{⨟} \Gamma \vdash \pi_i(e_0) : t_i}\ (proj)$$

As $erase(e') = \pi_i(erase(e_0))$, e' is either $\pi_i(e_0')$ or $\pi_i(e_0')[\sigma_j]_{j\in J}$ such that $erase(e_0') = erase(e_0)$. By induction, we have $\Delta \mathbin{⨟} \Gamma \vdash e_0 \sqcap e_0' : (t_1 \times t_2)$. Then by (*proj*), we have $\Delta \mathbin{⨟} \Gamma \vdash \pi_i(e_0 \sqcap e_0') : t_i$. If $e' = \pi_i(e_0')$, then $e \sqcap e' = \pi_i(e_0 \sqcap e_0')$. So the result follows.
Otherwise, $e \sqcap e' = \pi_i(e_0 \sqcap e_0')[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \mathbin{⨟} \Gamma \vdash \pi_i(e_0 \sqcap e_0')[\sigma_{id}, \sigma_j]_{j\in J} : t_i \wedge \bigwedge_{j\in J} t_i\sigma_j$. Finally, by (*subsum*), we get $\Delta \mathbin{⨟} \Gamma \vdash \pi_i(e_0 \sqcap e_0')[\sigma_{id}, \sigma_j]_{j\in J} : t_i$.

(*appl*): consider the following derivation:

$$\dfrac{\dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_1 : t \to s} \quad \dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_2 : t}}{\Delta \mathbin{⨟} \Gamma \vdash e_1 e_2 : s}\ (pair)$$

As $erase(e') = erase(e_1)erase(e_2)$, e' is either $e_1'e_2'$ or $(e_1'e_2')[\sigma_j]_{j\in J}$ such that $erase(e_i') = erase(e_i)$. By induction, we have $\Delta \mathbin{⨟} \Gamma \vdash e_1 \sqcap e_1' : t \to s$ and $\Delta \mathbin{⨟} \Gamma \vdash e_2 \sqcap e_2' : t$. Then by (*appl*), we have $\Delta \mathbin{⨟} \Gamma \vdash (e_1 \sqcap e_1')(e_2 \sqcap e_2') : s$. If $e' = e_1'e_2'$, then $e \sqcap e' = (e_1 \sqcap e_1')(e_2 \sqcap e_2')$. So the result follows.
Otherwise, $e \sqcap e' = ((e_1 \sqcap e_1')(e_2 \sqcap e_2'))[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \mathbin{⨟} \Gamma \vdash ((e_1 \sqcap e_1')(e_2 \sqcap e_2'))[\sigma_{id}, \sigma_j]_{j\in J} : s \wedge \bigwedge_{j\in J} s\sigma_j$. Finally, by (*subsum*), we get $\Delta \mathbin{⨟} \Gamma \vdash ((e_1 \sqcap e_1')(e_2 \sqcap e_2'))[\sigma_{id}, \sigma_j]_{j\in J} : s$.

(*abstr*): consider the following derivation:

$$\dfrac{\begin{array}{c}\forall i \in I.\ \dfrac{\ldots}{\Delta'' \mathbin{⨟} \Gamma, (x : t_i) \vdash e_0 : s_i} \\ \Delta'' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)\end{array}}{\Delta \mathbin{⨟} \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 : \bigwedge_{i\in I} t_i \to s_i}\ (abstr)$$

As $erase(e') = \lambda^{\wedge_{i\in I} t_i \to s_i} x.erase(e_0)$, e' is either $\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0'$ or $(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0')[\sigma_j]_{j\in J}$ such that $erase(e_0') = erase(e_0)$. As $\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0'$ is well-typed under Δ' and Γ', $e_0' \sharp \Delta' \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$. By induction, we have $\Delta'' \mathbin{⨟} \Gamma, (x : t_i) \vdash e_0 \sqcap e_0' : s_i$. Then by (*abstr*), we have $\Delta \mathbin{⨟} \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e_0' : \bigwedge_{i\in I} t_i \to s_i$. If $e' = \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0'$, then $e \sqcap e' = \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e_0'$. So the result follows.
Otherwise, $e \sqcap e' = (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e_0')[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \mathbin{⨟} \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e_0')[\sigma_{id}, \sigma_j]_{j\in J} : (\bigwedge_{i\in I} t_i \to s_i) \wedge \bigwedge_{j\in J}(\bigwedge_{i\in I} t_i \to s_i)\sigma_j$.
Finally, by (*subsum*), we get $\Delta \mathbin{⨟} \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e_0')[\sigma_{id}, \sigma_j]_{j\in J} : \bigwedge_{i\in I} t_i \to s_i$.

(*case*): consider the following derivation

$$\dfrac{\dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_0 : t'} \quad \left\{\begin{array}{lcl} t' \not\leq \neg t & \Rightarrow & \dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow & \dfrac{\ldots}{\Delta \mathbin{⨟} \Gamma \vdash e_2 : s}\end{array}\right.}{\Delta \mathbin{⨟} \Gamma \vdash (e_0 \in t\, ?\, e_1 : e_2) : s}\ (case)$$

As *erase*$(e') = $ *erase*$(e_0)\in t\,?\,$*erase*$(e_1) : $ *erase*(e_2), e' is either $e'_0\in t\,?\,e'_1 : e'_2$ or $(e'_0\in t\,?\,e'_1 : e'_2)[\sigma_j]_{j\in J}$ such that *erase*$(e'_i) = $ *erase*(e_i). By induction, we have $\Delta\,{\S}\,\Gamma \vdash e_0 \sqcap e'_0 : t'$ and $\Delta\,{\S}\,\Gamma \vdash e_i \sqcap e'_i : s$. Then by (*case*), we have $\Delta\,{\S}\,\Gamma \vdash ((e_0 \sqcap e'_0)\in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)) : s$. If $e' = e'_0\in t\,?\,e'_1 : e'_2$, then $e \sqcap e' = (e_0 \sqcap e'_0)\in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)$. So the result follows.
Otherwise, $e \sqcap e' = ((e_0 \sqcap e'_0)\in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,{\S}\,\Gamma \vdash ((e_0 \sqcap e'_0)\in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s \wedge \bigwedge_{j\in J} s\sigma_j$. Finally, by (*subsum*), we get $\Delta\,{\S}\,\Gamma \vdash ((e_0 \sqcap e'_0)\in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s$.

(*instinter*): consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash e_0 : t} \qquad \sigma_j \sharp \Delta}{\Delta\,{\S}\,\Gamma \vdash e_0[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (instinter)$$

As *erase*$(e') = $ *erase*(e_0), e' is either e'_0 or $e'_0[\sigma_j]_{j\in J'}$ such that *erase*$(e'_0) = $ *erase*(e_0). By induction, we have $\Delta\,{\S}\,\Gamma \vdash e_0 \sqcap e'_0 : t$. If $e' = e'_0$, then $e \sqcap e' = (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J}$. By (*instinter*), we have $\Delta\,{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J} : \bigwedge_{j\in J} t\sigma_j \wedge t$. Finally, by (*subsum*), we get $\Delta\,{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$.
Otherwise, $e \sqcap e' = (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$ for all $j \in J'$. By (*instinter*), we have $\Delta\,{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'} : \bigwedge_{j\in J\cup J'} t\sigma_j$. Finally, by (*subsum*), we get $\Delta\,{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'} : \bigwedge_{j\in J} t\sigma_j$.

(*subsum*): there exists a type s such that

$$\frac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash e : s} \qquad s \le t}{\Delta\,{\S}\,\Gamma \vdash e : t}\ (subsum)$$

By induction, we have $\Delta\,{\S}\,\Gamma \vdash e \sqcap e' : s$. Then the rule (*subsum*) gives us $\Delta\,{\S}\,\Gamma \vdash e \sqcap e' : t$.

□

Corollary A.12. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If erase$(e) = $ erase(e'), $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t$, $\Delta'\,{\S}\,\Gamma' \vdash_{\mathscr{A}} e' : t'$, $e \sharp \Delta'$ and $e' \sharp \Delta$, then*

1. *there exists s such that $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e \sqcap e' : s$ and $s \le t$.*
2. *there exists s' such that $\Delta'\,{\S}\,\Gamma' \vdash_{\mathscr{A}} e \sqcap e' : s'$ and $s' \le t'$.*

Proof. Immediate consequence of Lemma A.11 and Theorems C.22 and C.23 in the companion paper [3]. □

These type-substitution inference rules are sound and complete with respect to the typing algorithm in Section C.2 in the companion paper [3], modulo the restriction that all the decorations in the λ-abstractions are empty.

Theorem A.13 (Soundness). *If $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a : t$, then there exists an explicitly-typed expression $e \in \mathscr{E}_0$ such that erase$(e) = a$ and $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t$.*

Proof. By induction on the derivation of $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a : t$. We proceed by a case analysis of the last rule used in the derivation.

(INF-CONST): straightforward (take e as c).
(INF-VAR): straightforward (take e as x).
(INF-PAIR): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a_1 : t_1} \qquad \dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a_2 : t_2}}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}$$

Applying the induction hypothesis, there exists an expression e_i such that *erase*$(e_i) = a_i$ and $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_i : t_i$. Then by (ALG-PAIR), we have $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2$. Moreover, according to Definition A.2, we have *erase*$((e_1, e_2)) = ($*erase*$(e_1), $*erase*$(e_2)) = (a_1, a_2)$.

(INF-PROJ): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a : t} \qquad u \in \amalg^i_\Delta(t)}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}$$

By induction, there exists an expression e such that *erase*$(e) = a$ and $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t$. Let $u = \boldsymbol{\pi}_i(\bigwedge_{i\in I} t\sigma_i)$. As $\sigma_i \sharp \Delta$, by (ALG-INST), we have $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e[\sigma_i]_{i\in I} : \bigwedge_{i\in I} t\sigma_i$. Moreover, since $\bigwedge_{i\in I} t\sigma_i \le \mathbb{1} \times \mathbb{1}$, by (ALG-PROJ), we get $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} \pi_i(e[\sigma_i]_{i\in I}) : \boldsymbol{\pi}_i(\bigwedge_{i\in I} t\sigma_i)$. Finally, according to Definition A.2, we have *erase*$(\pi_i(e[\sigma_i]_{i\in I})) = \pi_i($*erase*$(e[\sigma_i]_{i\in I})) = \pi_i($*erase*$(e)) = \pi_i(a)$.

(INF-APPL): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_1 : t} \quad \dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_2 : s} \quad u \in t \bullet_\Delta s}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_1 a_2 : u}$$

By induction, we have that (i) there exists an expression e_1 such that $erase(e_1) = a_1$ and $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_1 : t$ and (ii) there exists an expression e_2 such that $erase(e_2) = a_2$ and $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_2 : s$. Let $u = (\bigwedge_{i\in I} t\sigma_i) \cdot (\bigwedge_{j\in J} s\sigma_j)$. As $\sigma_h \sharp \Delta$ for $h \in I \cup J$, applying (ALG-INST), we get $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_1[\sigma_i]_{i\in I} : \bigwedge_{i\in I} t\sigma_i$ and $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_2[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Then by (ALG-APPL), we have $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} (e_1[\sigma_i]_{i\in I})(e_2[\sigma_j]_{j\in J}) : (\bigwedge_{i\in I} t\sigma_i) \cdot (\bigwedge_{j\in J} s\sigma_j)$. Furthermore, according to Definition A.2, we have $erase((e_1[\sigma_i]_{i\in I})(e_2[\sigma_j]_{j\in J})) = erase(e_1)erase(e_2) = a_1 a_2$.

(INF-ABSTR): consider the derivation

$$\dfrac{\forall i \in I. \left\{ \begin{array}{l} \dfrac{\cdots}{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)\,\S\,\Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i} \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)} s_i \end{array} \right.}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}$$

Let $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$ and $[\sigma_{j_i}]_{j_i\in J_i} \Vdash s'_i \sqsubseteq_{\Delta'} s_i$. By induction, there exists an expression e_i such that $erase(e_i) = a$ and $\Delta'\,\S\,\Gamma, (x : t_i) \vdash_{\mathscr{A}} e_i : s'_i$ for all $i \in I$. Since $\sigma_{j_i} \sharp \Delta'$, by (ALG-INST), we have $\Delta'\,\S\,\Gamma, (x : t_i) \vdash_{\mathscr{A}} e_i[\sigma_{j_i}]_{j_i\in J_i} : \bigwedge_{j_i\in J_i} s'_i\sigma_{j_i}$. Clearly, $e_i[\sigma_{j_i}]_{j_i\in J_i} \sharp \Delta'$ and $erase(e_i[\sigma_{j_i}]_{j_i\in J_i}) = erase(e_i) = a$. Then by Lemma A.10, the intersection $\bigsqcap_{i\in I}(e_i[\sigma_{j_i}]_{j_i\in J_i})$ exists and we have $erase(\bigsqcap_{i\in I'}(e_i[\sigma_{j_i}]_{j_i\in J_i})) = a$ for any non-empty $I' \subseteq I$. Let $e = \bigsqcap_{i\in I}(e_i[\sigma_{j_i}]_{j_i\in J_i})$. According to Corollary A.12, there exists a type t'_i such that $\Delta'\,\S\,\Gamma, (x : t_i) \vdash_{\mathscr{A}} e : t'_i$ and $t'_i \leq \bigwedge_{j_i\in J_i} s'_i\sigma_{j_i}$ for all $i \in I$. Moreover, since $t'_i \leq \bigwedge_{j_i\in J_i} s'_i\sigma_{j_i} \leq s_i$, by (ALG-ABSTR), we have $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.e : \bigwedge_{i\in I}(t_i \to s_i)$. Finally, according to Definition A.2, we have

$$erase(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) = \lambda^{\wedge_{i\in I} t_i \to s_i} x.erase(e) = \lambda^{\wedge_{i\in I} t_i \to s_i} x.a.$$

(INF-CASE-NONE): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a : t'} \quad t' \sqsubseteq_\Delta \mathbb{0}}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \mathbb{0}}$$

By induction, there exists an expression e such that $erase(e) = a$ and $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e : t'$. Let $[\sigma_i]_{i\in I} \Vdash t' \sqsubseteq_\Delta \mathbb{0}$. Since $\sigma_i \sharp \Delta$, by (ALG-INST), we have $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e[\sigma_i]_{i\in I} : \bigwedge_{i\in I} t'\sigma_i$. Let e_1 and e_2 be two expressions such that $erase(e_1) = a_1$ and $erase(e_2) = a_2$. Then we have

$$erase((e[\sigma_i]_{i\in I}){\in}t\,?\,e_1 : e_2) = (a{\in}t\,?\,a_1 : a_2).$$

Moreover, since $\bigwedge_{i\in I} t'\sigma_i \leq \mathbb{0}$, by (ALG-CASE-NONE), we have

$$\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} ((e[\sigma_i]_{i\in I}){\in}t\,?\,e_1 : e_2) : \mathbb{0}.$$

(INF-CASE-FST): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a : t'} \quad t' \sqsubseteq_\Delta t \quad t' \not\sqsubseteq_\Delta \neg t \quad \dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_1 : s}}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s}$$

By induction, there exist e, e_1 such that $erase(e) = a$, $erase(e_1) = a_1$, $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e : t'$, and $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_1 : s$. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t' \sqsubseteq_\Delta t$. Since $\sigma_{i_1} \sharp \Delta$, applying (ALG-INST), we get $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e[\sigma_{i_1}]_{i_1\in I_1} : \bigwedge_{i_1\in I_1} t'\sigma_{i_1}$. Let e_2 be an expression such that $erase(e_2) = a_2$. Then we have

$$erase((e[\sigma_{i_1}]_{i_1\in I_1}){\in}t\,?\,e_1 : e_2) = (a{\in}t\,?\,a_1 : a_2).$$

Finally, since $\bigwedge_{i_1\in I_1} t'\sigma_{i_1} \leq t$, by (ALG-CASE-FST), we have

$$\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} ((e[\sigma_{i_1}]_{i_1\in I_1}){\in}t\,?\,e_1 : e_2) : s.$$

(INF-CASE-SND): similar to the case of (INF-CASE-FST).

(INF-CASE-BOTH): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a : t'} \quad \left\{ \begin{array}{lll} t' \not\sqsubseteq_\Delta \neg t & \text{and} & \dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_1 : s_1} \\ t' \not\sqsubseteq_\Delta t & \text{and} & \dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} a_2 : s_2} \end{array} \right.}{\Delta\,\S\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}$$

By induction, there exist e, e_i such that $erase(e) = a$, $erase(e_i) = a_i$, $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e : t'$, and $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} e_i : s_i$. According to Definition A.2, we have $erase((e{\in}t\,?\,e_1 : e_2)) = (a{\in}t\,?\,a_1 : a_2)$. Clearly $t' \not\simeq \mathbb{0}$. We claim that $t' \not\leq \neg t$. Let σ_{id} be any identity type substitution. If $t' \leq \neg t$, then $t'\sigma_{id} \simeq t' \leq \neg t$, i.e., $t' \sqsubseteq_\Delta \neg t$, which is in contradiction with $t' \not\sqsubseteq_\Delta \neg t$. Similarly we have $t' \not\leq t$. Therefore, by (ALG-CASE-SND), we have $\Delta\,\S\,\Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2$.

$\square$

The proof of the soundness property constructs along the derivation for a some expression e that satisfies the statement of the theorem. We denote by $erase^{-1}(a)$ the set of expressions e that satisfy the statement.

Theorem A.14 (Completeness). *Let $e \in \mathscr{E}_0$ be an explicitly-typed expression. If $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e : t$, then there exists a type t' such that $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e) : t'$ and $t' \sqsubseteq_\Delta t$.*

Proof. By induction on the typing derivation of $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e : t$. We proceed by a case analysis on the last rule used in the derivation.

(ALG-CONST): take t' as b_c.
(ALG-VAR): take t' as $\Gamma(x)$.
(ALG-PAIR): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e_1 : t_1} \qquad \dfrac{\cdots}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e_2 : t_2}}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2}$$

Applying the induction hypothesis twice, we have

$$\begin{array}{l}\exists t_1'.\ \Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e_1) : t_1' \text{ and } t_1' \sqsubseteq_\Delta t_1,\\ \exists t_2'.\ \Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e_2) : t_2' \text{ and } t_2' \sqsubseteq_\Delta t_2.\end{array}$$

Then by (INF-PAIR), we have $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} (erase(e_1), erase(e_2)) : t_1' \times t_2'$, that is, $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase((e_1, e_2)) : t_1' \times t_2'$. Finally, Applying Lemma A.4, we have $(t_1' \times t_2') \sqsubseteq_\Delta (t_1 \times t_2)$.

(ALG-PROJ): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e : t} \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)}$$

By induction, we have

$$\exists t', [\sigma_k]_{k\in K}.\ \Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e) : t' \text{ and } [\sigma_k]_{k\in K} \Vdash t' \sqsubseteq_\Delta t.$$

It is clear that $\bigwedge_{k\in K} t'\sigma_k \leq \mathbb{1} \times \mathbb{1}$. So $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \in \amalg^i_\Delta(t')$. Then by (INF-PROJ), we have $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} \pi_i(erase(e)) : \boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k)$, that is, $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(\pi_i(e)) : \boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k)$. According to Lemma C.5 in the companion paper [3], $t \leq (\boldsymbol{\pi}_1(t), \boldsymbol{\pi}_2(t))$. Then $\bigwedge_{k\in K} t'\sigma_k \leq (\boldsymbol{\pi}_1(t), \boldsymbol{\pi}_2(t))$. Finally, applying Lemma C.5 again, we get $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \leq \boldsymbol{\pi}_i(t)$ and *a fortiori* $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \sqsubseteq_\Delta \boldsymbol{\pi}_i(t)$.

(ALG-APPL): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e_1 : t} \qquad \dfrac{\cdots}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e_2 : s} \qquad t \leq \mathbb{0} \to \mathbb{1} \qquad s \leq \mathsf{dom}(t)}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} e_1 e_2 : t \cdot s}$$

Applying the induction hypothesis twice, we have

$$\begin{array}{l}\exists t_1', [\sigma_k^1]_{k\in K_1}.\ \Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e_1) : t_1' \text{ and } [\sigma_k^1]_{k\in K_1} \Vdash t_1' \sqsubseteq_\Delta t,\\ \exists t_2', [\sigma_k^2]_{k\in K_2}.\ \Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e_2) : t_2' \text{ and } [\sigma_k^2]_{k\in K_2} \Vdash t_2' \sqsubseteq_\Delta s.\end{array}$$

It is clear that $\bigwedge_{k\in K_1} t_1'\sigma_k^1 \leq \mathbb{0} \to \mathbb{1}$, that is, $\bigwedge_{k\in K_1} t_1'\sigma_k^1$ is a function type. So we get $\mathsf{dom}(t) \leq \mathsf{dom}(\bigwedge_{k\in K_1} t_1'\sigma_k^1)$. Then we have $\bigwedge_{k\in K_2} t_2'\sigma_k^2 \leq s \leq \mathsf{dom}(t) \leq \mathsf{dom}(\bigwedge_{k\in K_1} t_1'\sigma_k^1)$. Therefore, $(\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \in t_2' \bullet_\Delta t_1'$. Then applying (INF-APPL), we have

$$\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e_1)erase(e_2) : (\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2),$$

that is, $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(e_1e_2) : (\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2)$. Moreover, as $\bigwedge_{k\in K_2} t_2'\sigma_k^2 \leq \mathsf{dom}(t)$, $t \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2)$ exists. According to Lemma C.14 in the companion paper [3], we have

$$(\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \leq t \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \leq t \cdot s.$$

Thus, $(\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \sqsubseteq_\Delta t \cdot s$.

(ALG-ABSTR0): consider the derivation

$$\dfrac{\forall i \in I.\ \dfrac{\cdots}{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \mathbin{⨟} \Gamma, (x : t_i) \vdash_{\mathscr{A}} e : s_i'} \quad \text{and } s_i' \leq s_i}{\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.e : \bigwedge_{i\in I} t_i \to s_i}$$

Let $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$. By induction, for each $i \in I$, we have

$$\exists t_i'.\ \Delta' \mathbin{⨟} \Gamma, (x : t_i) \vdash_{\mathscr{I}} erase(e) : t_i' \text{ and } t_i' \sqsubseteq_{\Delta'} s_i'.$$

Clearly, we have $t_i' \sqsubseteq_{\Delta'} s_i$. By (INF-ABSTR), we have

$$\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.erase(e) : \bigwedge_{i\in I} t_i \to s_i,$$

that is, $\Delta \mathbin{⨟} \Gamma \vdash_{\mathscr{I}} erase(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) : \bigwedge_{i\in I} t_i \to s_i$.

(ALG-CASE-NONE): consider the derivation

$$\frac{\frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e : \mathbb{0}}}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : \mathbb{0}}$$

By induction, we have

$$\exists t'_0.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e) : t'_0 \text{ and } t'_0 \sqsubseteq_{\Delta} \mathbb{0}.$$

By (INF-CASE-NONE), we have $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} (\textit{erase}(e){\in}t\,?\,\textit{erase}(e_1) : \textit{erase}(e_2)) : \mathbb{0}$, that is, $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$.

(ALG-CASE-FST): consider the derivation

$$\frac{\frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e : t'} \quad t' \leq t \quad \frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1}}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1}$$

Applying the induction hypothesis twice, we have

$$\exists t'_0.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e) : t'_0 \text{ and } t'_0 \sqsubseteq_{\Delta} t',$$
$$\exists t'_1.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e_1) : t'_1 \text{ and } t'_1 \sqsubseteq_{\Delta} s_1.$$

Clearly, we have $t'_0 \sqsubseteq_{\Delta} t$. If $t'_0 \sqsubseteq_{\Delta} \neg t$, then by Lemma A.4, we have $t'_0 \leq_{\Delta} \mathbb{0}$. By (INF-CASE-NONE), we get

$$\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} (\textit{erase}(e){\in}t\,?\,\textit{erase}(e_1) : \textit{erase}(e_2)) : \mathbb{0},$$

that is, $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$. Clearly, we have $\mathbb{0} \sqsubseteq_{\Delta} s_1$.
Otherwise, by (INF-CASE-FST), we have

$$\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} (\textit{erase}(e){\in}t\,?\,\textit{erase}(e_1) : \textit{erase}(e_2)) : t'_1,$$

that is, $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1$. The result follows as well.

(ALG-CASE-SND): similar to the case of (ALG-CASE-FST).

(ALG-CASE-BOTH): consider the derivation

$$\frac{\frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e : t'} \qquad \begin{cases} t' \not\leq \neg t & \text{and} \quad \frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1} \\ t' \not\leq t & \text{and} \quad \frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e_2 : s_2} \end{cases}}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2}$$

By induction, we have

$$\exists t'_0.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e) : t'_0 \text{ and } t'_0 \sqsubseteq_{\Delta} t',$$
$$\exists t'_1.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e_1) : t'_1 \text{ and } t'_1 \sqsubseteq_{\Delta} s_1,$$
$$\exists t'_2.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e_2) : t'_2 \text{ and } t'_2 \sqsubseteq_{\Delta} s_2.$$

If $t'_0 \sqsubseteq_{\Delta} \mathbb{0}$, then by (INF-CASE-NONE), we get

$$\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} (\textit{erase}(e){\in}t\,?\,\textit{erase}(e_1) : \textit{erase}(e_2)) : \mathbb{0},$$

that is, $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$. Clearly, we have $\mathbb{0} \sqsubseteq_{\Delta} s_1 \vee s_2$.
If $t'_0 \sqsubseteq_{\Delta} t$, then by (INF-CASE-FST), we get

$$\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} (\textit{erase}(e){\in}t\,?\,\textit{erase}(e_1) : \textit{erase}(e_2)) : t'_1,$$

that is, $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1$. Moreover, it is clear that $t'_1 \sqsubseteq_{\Delta} s_1 \vee s_2$, the result follows as well. Similarly for $t'_0 \sqsubseteq_{\Delta} \neg t$.
Otherwise, by (INF-CASE-BOTH), we have

$$\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} (\textit{erase}(e){\in}t\,?\,\textit{erase}(e_1) : \textit{erase}(e_2)) : t'_1 \vee t'_2,$$

that is, $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1 \vee t'_2$. Using α-conversion, we can assume that the polymorphic type variables of t'_1 and t'_2 (and of e_1 and e_2) are distinct, i.e., $(\mathsf{var}(t'_1) \setminus \Delta) \cap (\mathsf{var}(t'_1) \setminus \Delta) = \emptyset$. Then applying Lemma A.5, we have $t'_1 \vee t'_2 \sqsubseteq_{\Delta} t_1 \vee t_2$.

(ALG-INST): consider the derivation

$$\frac{\frac{\dots}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e : t} \quad \forall j \in J.\ \sigma_j \sharp \Delta \quad |J| > 0}{\Delta \,{\S}\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

By induction, we have

$$\exists t', [\sigma_k]_{k\in K}.\ \Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e) : t' \text{ and } [\sigma_k]_{k\in K} \Vdash t' \sqsubseteq_{\Delta} t.$$

Since $\textit{erase}(e[\sigma_j]_{j\in J}) = \textit{erase}(e)$, we have $\Delta \,{\S}\, \Gamma \vdash_{\mathscr{I}} \textit{erase}(e[\sigma_j]_{j\in J}) : t'$. As $\bigwedge_{k\in K} t'\sigma_k \leq t$, we have $\bigwedge_{j\in J}(\bigwedge_{k\in K} t'\sigma_k)\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$, that is $\bigwedge_{k\in K, j\in J} t'(\sigma_j \circ \sigma_k) \leq \bigwedge_{j\in J} t\sigma_j$. Moreover, it is clear that $\sigma_j \circ \sigma_k \sharp \Delta$. Therefore, we get $t' \sqsubseteq_{\Delta} \bigwedge_{j\in J} t\sigma_j$.

□

The inference system is syntax directed and describes an algorithm that is parametric in the decision procedures for $\sqsubseteq_{\Delta}$, $\amalg^i_{\Delta}(t)$ and $t \bullet_{\Delta} s$. The problem of deciding them is tackled in Section B.2.

Finally, notice that we did not give any reduction semantics for the implicitly typed calculus. The reason is that its semantics is defined in terms of the semantics of the explicitly-typed calculus: the relabeling at run-time is an essential feature —independently from the fact that we started from an explicitly typed expression or not— and we cannot avoid it. The (big-step) semantics for a is then given in expressions of $\textit{erase}^{-1}(a)$: if an expression in $\textit{erase}^{-1}(a)$ reduces to v, so does a. As we see the result of computing an implicitly-typed expression is a value of the explicitly typed calculus (so λ-abstractions may contain non-empty decorations) and this is unavoidable since it may be the result of a partial application. Also notice that the semantics is not deterministic since different expressions in $\textit{erase}^{-1}(a)$ may yield different results. However this may happen only in one particular case, namely, when an occurrence of a polymorphic function flows into a type-case and its type is tested. For instance the application $(\lambda^{(\texttt{Int}\to\texttt{Int})\to\texttt{Bool}} f.f{\in}\texttt{Bool}{\to}\texttt{Bool}\,?\,\texttt{true} : \texttt{false})(\lambda^{\alpha\to\alpha}x.x)$ results into `true` or `false` according to whether the polymorphic identity at the argument is instantiated by $[\{{}^{\texttt{Int}}/_{\alpha}\}]$ or by $[\{{}^{\texttt{Int}}/_{\alpha}\}, \{{}^{\texttt{Bool}}/_{\alpha}\}]$. Once more this is unavoidable in a calculus that can dynamically test the types of polymorphic functions that admit several sound instantiations.

A.3 A More Tractable Type Inference System

With the rules of Figure 5, when type-checking an implicitly-typed expression, we have to compute sets of type substitutions for projections, applications, abstractions and type cases. Because type substitutions inference is a costly operation, we would like to perform it as less as possible. To this end, we give in this section a restricted version of the inference system, which is not complete but still sound and powerful enough to be used in practice.

First, we want to simplify the type inference rule for projections:

$$\frac{\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} a : t \quad u \in \amalg^i_\Delta(t)}{\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}$$

where $\amalg^i_\Delta(t) = \{u \mid [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1}\times\mathbb{1},\ u = \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)\}$. Instead of picking any type in $\amalg^i_\Delta(t)$, we would like to simply project t, i.e., assign the type $\boldsymbol{\pi}_i(t)$ to $\pi_i(a)$. By doing so, we lose completeness on pair types that contain top-level variables. For example, if $t = (\texttt{Int}\times\texttt{Int})\wedge\alpha$, then $\texttt{Int}\wedge\texttt{Bool} \in \amalg^i_\Delta(t)$ (because α can be instantiated with $(\texttt{Bool}\times\texttt{Bool})$), but $\boldsymbol{\pi}_t(t) = \texttt{Int}$. We also lose typability if t is not a pair type, but can be instantiated in a pair type. For example, the type of $(\lambda^{\alpha\to(\alpha\vee((\beta\to\beta)\setminus(\texttt{Int}\to\texttt{Int})))}x.x)(42,3)$ is $(\texttt{Int}\times\texttt{Int})\vee((\beta\to\beta)\setminus(\texttt{Int}\to\texttt{Int}))$, which is not a pair type, but can be instantiated in $(\texttt{Int}\times\texttt{Int})$ by taking $\beta = \texttt{Int}$. We believe these kinds of types will not be written by programmers, and it is safe to use the following projection rule in practice.

$$\frac{\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} a : t \qquad t \leq \mathbb{1}\times\mathbb{1}}{\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} \pi_i(a) : \boldsymbol{\pi}_i(t)}\,(\textsc{Inf-proj'})$$

We now look at the type inference rules for the type case $a{\in}t\,?\,a_1 : a_2$. The four different rules consider the different possible instantiations that make the type t' inferred for a fit t or not. For the sake of simplicity, we decide not to infer type substitutions for polymorphic arguments of type cases. Indeed, in the expression $(\lambda^{\alpha\to\alpha}x.x) \in \texttt{Int}\to\texttt{Int}\ ?\ \texttt{true}\ :\ \texttt{false}$, we assume the programmer wants to do a type case on the polymorphic identity, and not on one of its instance (otherwise, he would have written the instantiated interface directly), so we do not try to instantiate it. And in any case there is no real reason for which the inference system should choose to instantiate the identity by Int→Int (and thus make the test succeed) rather than Bool→Bool (and thus make the test fail). If we decide not to infer types for polymorphic arguments of type-case expression, then since $\alpha\to\alpha$ is not a subtype of $\texttt{Int}\to\texttt{Int}$ (we have $\alpha\to\alpha \sqsubseteq_\emptyset \texttt{Int}\to\texttt{Int}$ but $\alpha\to\alpha \not\leq \texttt{Int}\to\texttt{Int}$) the expression evaluates to `false`. With this choice, we can merge the different inference rules into the following one.

$$\frac{\begin{array}{l}\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} a : t' \quad t_1 = t'\wedge t \quad t_2 = t'\wedge\neg t\\ t_i \not\simeq \mathbb{0} \;\Rightarrow\; \Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} a_i : s_i\end{array}}{\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \bigvee_{t_i\not\simeq\mathbb{0}} s_i}\,(\textsc{Inf-case'})$$

Finally, consider the inference rule for abstractions:

$$\frac{\forall i\in I.\ \left\{\begin{array}{l}\Delta\cup\mathsf{var}(\bigwedge_{i\in I} t_i\to s_i)\,{}_{\S}\,\Gamma,(x:t_i) \vdash_{\mathscr{I}} a : s'_i\\ s'_i \sqsubseteq_{\Delta\cup\mathsf{var}(\bigwedge_{i\in I} t_i\to s_i)} s_i\end{array}\right.}{\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i\to s_i}x.a : \bigwedge_{i\in I} t_i\to s_i}$$

We verify that the abstraction can be typed with each arrow type $t_i\to s_i$ in the interface. Meanwhile, we also infer a set of type substitutions to tally the type s'_i we infer for the body expression with s_i. In practice, similarly, we expect that the abstraction is well-typed only if the type s'_i we infer for the body expression is a subtype of s_i. For example, the expression

$$\lambda^{\texttt{Bool}\to(\texttt{Int}\to\texttt{Int})}x.x{\in}\texttt{true}\,?\,(\lambda^{\alpha\to\alpha}y.y) : (\lambda^{\alpha\to\alpha}y.(\lambda^{\alpha\to\alpha}z.z)y)$$

is not well-typed while

$$\lambda^{\texttt{Bool}\to(\alpha\to\alpha)}x.x{\in}\texttt{true}\,?\,(\lambda^{\alpha\to\alpha}y.y) : (\lambda^{\alpha\to\alpha}y.(\lambda^{\alpha\to\alpha}z.z)y)$$

is well-typed. So we use the following restricted rule for abstractions instead:

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,⨟\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s_i' \text{ and } s_i' \leq s_i}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i} (\text{INF-ABSTR'})$$

In conclusion, we restrict the inference of type substitutions to applications. We give in Figure 6 the inference rules of the system which respects the above restrictions. With these new rules, the system remains sound, but it is not complete.

$$\frac{}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} c : b_c} (\text{INF-CONST}) \qquad \frac{}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} x : \Gamma(x)} (\text{INF-VAR})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : t_1 \qquad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : t_2}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2} (\text{INF-PAIR}) \qquad \frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \pi_i(a) : \boldsymbol{\pi}_i(t)} (\text{INF-PROJ'})$$

$$\frac{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 : t \qquad \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_2 : s \qquad u \in t \bullet_\Delta s}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u} (\text{INF-APPL})$$

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,⨟\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s_i' \text{ and } s_i' \leq s_i}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i} (\text{INF-ABSTR'})$$

$$\frac{\begin{array}{l}\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a : t' \quad t_1 = t' \wedge t \quad t_2 = t' \wedge \neg t \\ t_i \not\simeq \mathbb{0} \;\Rightarrow\; \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} a_i : s_i\end{array}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \bigvee_{t_i \not\simeq \mathbb{0}} s_i} (\text{INF-CASE'})$$

Figure 6. Restricted type-substitution inference rules

Theorem A.15. *If $\Gamma \vdash_{\mathscr{I}} a : t$, then there exists an expression $e \in \mathscr{E}_0$ such that erase$(e) = a$ and $\Gamma \vdash_{\mathscr{A}} e : t$.*

Proof. Similar to the proof of Theorem A.13. □

B. Type Tallying

Given two types t and s, the goal of this section is to find pairs of sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $\bigwedge_{j\in J} s\sigma_j \leq \bigvee_{i\in I} t\sigma_i$. Assuming that the cardinalities of I and J are known, then this problem can be reduced to a *type tallying* problem, that we define and solve first. We then explain how we can reduce the original problem to the type tallying problem, and provide a semi-algorithm for the original problem. Finally, we give some heuristics to establish upper bounds (which depend on t and s) for the cardinalities of I and J.

B.1 Type Tallying Problem

Given a finite set C of pairs of types and a finite set Δ of type variables, the tallying problem for C and Δ consists in verifying whether there exists a substitution σ such that $\sigma \sharp \Delta$ and for all $(s,t) \in C$, $s\sigma \leq t\sigma$ holds. In this section we denote constraints as triples. The notation is different from the one used in Section 3 in that it also specifies the symbol of the relation. So a pair of types $(s,t) \in C$ corresponds to the constraint $(s, \leq, t)$:

Definition B.1 (Constraints). *A* constraint (t, c, s) *is a triple belonging to $\mathscr{T} \times \{\leq, \geq\} \times \mathscr{T}$. Let $\mathscr{C}$ denote the set of all constraints. Given a constraint-set $C \subseteq \mathscr{C}$, the* set of type variables *occurring in C is defined as*

$$\mathit{var}(C) = \bigcup_{(t,c,s)\in C} \mathit{var}(t) \cup \mathit{var}(s)$$

Definition B.2 (**Normalized constraint**). *A constraint (t, c, s) is said to be* normalized *if t is a type variable. A constraint-set $C \subseteq \mathscr{C}$ is said to be normalized if every constraint $(t, c, s) \in C$ is normalized. Given a normalized constraint-set C, its domain is defined as $\mathsf{dom}(C) = \{\alpha \mid \exists c, s.\ (\alpha, c, s) \in C\}$.*

Definition B.3 (**Constraint solution**). *Let $C \subseteq \mathscr{C}$ be a constraint-set. A solution to C is a substitution σ such that*

$$\forall (t, \leq, s) \in C\ .\ t\sigma \ \leq\ s\sigma \textit{ holds} \quad \textit{and} \quad \forall (t, \geq, s) \in C\ .\ s\sigma \ \leq\ t\sigma \textit{ holds.}$$

If σ is a solution to C, we write $\sigma \Vdash C$.

Definition B.4. *Given two sets of constraint-sets $\mathscr{S}_1, \mathscr{S}_2 \subseteq \mathscr{P}(\mathscr{C})$, we define their* union *as*

$$\mathscr{S}_1 \sqcup \mathscr{S}_2 = \mathscr{S}_1 \cup \mathscr{S}_2$$

and their intersection as

$$\mathscr{S}_1 \sqcap \mathscr{S}_2 = \{C_1 \cup C_2 \mid C_1 \in \mathscr{S}_1, C_2 \in \mathscr{S}_2\}$$

Given a constraint-set C, the constraint solving algorithm produces the set of all the solutions of C by following the algorithm given in Section 3.2.1. Let us examine each step of the algorithm on some examples.

Step 1: constraint normalization.
Because normalized constraints are easier to solve than regular ones, we first turn each constraint into an equivalent set of normalized constraint-sets according to the decomposition rules in [4]. For example, the constraint $c_1 = (\alpha \times \alpha) \leq ((\texttt{Int} \times \mathbb{1}) \times (\mathbb{1} \times \texttt{Int}))$ can be normalized into the set $\mathscr{S}_1 = \{\{(\alpha, \leq, \mathbb{0})\}; \{(\alpha, \leq, (\texttt{Int} \times \mathbb{1})), (\alpha, \leq, (\mathbb{1} \times \texttt{Int}))\}\}$. Another example is the constraint $c_2 = ((\beta \times \beta) \to (\texttt{Int} \times \texttt{Int}), \leq, \alpha \to \alpha)$, which is equivalent to the following set of normalized constraint-sets $\mathscr{S}_2 = \{\{(\alpha, \leq, \mathbb{0})\}; \{(\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}\}$. Then we join all the sets of constraint-sets by (constraint-set) intersections, yielding the normalization of the original constraint-set. For instance, the normalization $\mathscr{S}$ of $\{c_1, c_2\}$ is $\mathscr{S}_1 \sqcap \mathscr{S}_2$. It is easy to see that the constraint-set $C_1 = \{(\alpha, \leq, (\texttt{Int} \times \mathbb{1})), (\alpha, \leq, (\mathbb{1} \times \texttt{Int})), (\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}$ is in $\mathscr{S}$ (see Definition B.4).

Step 2: constraint merging.

Step 2.1: merge the constraints with a same type variable.
In each constraint-set of the normalization of the original constraint-set, there may be several constraints of the form $(\alpha, \geq, t_i)$ (resp. $(\alpha, \leq, t_i)$), which give different lower bounds (resp. upper bounds) for α. We merge all these constraints into one using unions (resp. intersections). For example, the constraint-set C_1 of the previous step can be merged as $C_2 = \{(\alpha, \leq, (\texttt{Int} \times \mathbb{1}) \wedge (\mathbb{1} \times \texttt{Int}) \wedge (\beta \times \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}$, which is equivalent to $\{(\alpha, \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}$.

Step 2.2: saturate the lower and upper bounds of a same type variable.
If a type variable has both a lower bound s and an upper bound t in a constraint-set, then the solutions we are looking for must satisfy the constraint $(s, \leq, t)$ as well. Therefore, we have to saturate the constraint-set with $(s, \leq, t)$, which has to be normalized, merged, and saturated itself first. Take C_2 for example. We have to saturate C_2 with $((\texttt{Int} \times \texttt{Int}), \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta))$, whose normalization is $\{\{(\beta, \geq, \texttt{Int})\}\}$. Thus, the saturation of C_2 is $\{C_2\} \sqcap \{\{(\beta, \geq, \texttt{Int})\}\}$, which contains only one constraint-set $C_3 = \{(\alpha, \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int})), (\beta, \geq, \texttt{Int})\}$.

Step 3: constraint solving.

Step 3.1: transform each constraint-set into an equation system.
To transform constraints into equations, we use the property that some set of constraints is satisfied for all assignments of α included between s and t if and only if the same set in which we replace α by $(s \vee \alpha') \wedge t$[8] is satisfied for all possible assignments of α' (with α' fresh). Of course such a transformation works only if $s \leq t$, but remember that we "checked" that this holds at the moment of the saturation. By performing this replacement for each variable we obtain a system of equations. For example, the constraint set C_3 is equivalent to the following equation system E:

$$\begin{aligned} \alpha &= \quad ((\texttt{Int} \times \texttt{Int}) \vee \alpha') \wedge (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta) \\ \beta &= \quad \texttt{Int} \vee \beta' \end{aligned}$$

where α', β' are fresh type variables.

Step 3.2: extract a substitution from each equation system.
Finally, using the Courcelle's work on infinite trees [8], we solve each equation system, which gives us a substitution which is a solution of the original constraint-set. For example, we can solve the equation system E, yielding the type-substitution $\{{}^{(\texttt{Int} \times \texttt{Int})}/_{\alpha}, {}^{\texttt{Int} \vee \beta'}/_{\beta}\}$, which is a solution of C_3 and thus of $\{c_1, c_2\}$.

In the following subsections we study in details each step of the algorithm.

[8] Or by $s \vee (\alpha' \wedge t)$.

B.1.1 Constraint Normalization

The type tallying problem is quite similar to the subtyping problem presented in [4]. We therefore reuse most of the technology developed in [4] such as, for example, the transformation of the subtyping problem into an emptiness decision problem, the elimination of top-level constructors, and so on. One of the main differences is that we do not want to eliminate top-level type variables from constraints, but, rather, we want to isolate them to build sets of normalized constraints (from which we then construct sets of substitutions).

In general, normalizing a constraint generates a set of constraints. For example, $(\alpha\vee\beta, \geq, \mathbb{0})$ holds if and only if $(\alpha, \geq, \mathbb{0})$ or $(\beta, \geq, \mathbb{0})$ holds; therefore the constraint $(\alpha \vee \beta, \geq, \mathbb{0})$ is equivalent to the normalized constraint-set $\{(\alpha, \geq, \mathbb{0}), (\beta, \geq, \mathbb{0})\}$. Consequently, the normalization of a constraint-set C yields a set $\mathscr{S}$ of normalized constraint-sets.

Several normalized sets may be suitable replacements for a given constraint; for example, $\{(\alpha, \leq, \beta \vee t_1), (\beta, \leq, \alpha \vee t_2)\}$ and $\{(\alpha, \leq, \beta \vee t_1), (\alpha, \geq, \beta \setminus t_2)\}$ are clearly equivalent normalized sets. However, the equation systems generated by the algorithm for these two sets are completely different, and different equation systems yield different substitutions (see Section B.1.3 for more details). Concretely, $\{(\alpha, \leq, \beta \vee t_1), (\beta, \leq, \alpha \vee t_2)\}$ generates the equation system $\{\alpha = \alpha' \wedge (\beta \vee t_1), \beta = \beta' \wedge (\alpha \vee t_2)\}$, which in turn gives the substitution σ_1 such that

$$\begin{array}{l}\sigma_1(\alpha) = \mu x.\ ((\alpha' \wedge \beta' \wedge x) \vee (\alpha' \wedge \beta' \wedge t_2) \vee (\alpha' \wedge t_1))\\ \sigma_1(\beta) = \mu x.\ ((\beta' \wedge \alpha' \wedge x) \vee (\beta' \wedge \alpha' \wedge t_1) \vee (\beta' \wedge t_2))\end{array}$$

where α' and β' are fresh type variables and we used the μ notation to denote regular recursive types. These recursive types are not valid in our calculus, because x does not occur under a type constructor (this means that the unfolding of the type does not satisfy the property that every infinite branch contains infinitely many occurrences of type constructors). In contrast, the equation system built from $\{(\alpha, \leq, \beta\vee t_1), (\alpha, \geq, \beta\setminus t_2)\}$ is $\alpha = ((\beta \setminus t_2) \vee \alpha') \wedge (\beta \vee t_1)$, and the corresponding substitution is $\sigma_2 = \{((\beta \setminus t_2) \vee \alpha') \wedge (\beta \vee t_1)/_{\alpha}\}$, which is valid since it maps the type variable α into a well-formed type. Ill-formed recursive types are generated when there exists a chain $\alpha_0 = \alpha_1\ B_1\ t_1, \ldots, \alpha_i = \alpha_{i+1}\ B_{i+1}\ t_{i+1}, \ldots, \alpha_n = \alpha_0\ B_{n+1}\ t_{n+1}$ (where $B_i \in \{\wedge, \vee\}$ for all i, and $n \geq 0$) in the equation system built from the normalized constraint-set. This chain implies the equation $\alpha_0 = \alpha_0\ B\ t'$ for some $B \in \{\wedge, \vee\}$ and t', and the corresponding solution for α_0 will be an ill-formed recursive type. To avoid this issue, we give an arbitrary ordering on type variables occurring in the constraint-set C such that different type variables have different orders. Then we always select the normalized constraint (α, c, t) such that the order of α is smaller than all the orders of the top-level type variables in t. As a result, the transformed equation system does not contain any problematic chain like the one above.

Definition B.5 (Ordering). *Let V be a set of type variables. An* ordering *O on V is an injective map from V to $\mathbb{N}$.*

We formalize normalization as a judgement $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, which states that under the environment Σ (which, informally, contains the types that have already been processed at this point), C is normalized to $\mathscr{S}$. The judgement is derived according the rules of Figure 7. These rules describe the same algorithm as the function norm given in Figure 3 (*ie*, $\Sigma \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathsf{norm}(t, \Sigma)$ is provable in the system of Figure 7) but extended to handle also product types. We just switched to a deduction systems since it eases the formal treatment.

If the constraint-set is empty, then clearly any substitution is a solution, and, the result of the normalization is simply the singleton containing the empty set (rule (NEMPTY)). Otherwise, each constraint is normalized separately, and the normalization of the constraint-set is the intersection of the normalizations of each constraint (rule (NJOIN)). By using rules (NSYM), (NZERO), and (NDNF) repeatedly, we transform any constraint into the constraint of the form $(\tau, \leq, \mathbb{0})$ where τ is disjunctive normal form: the first rule reverses $(t', \geq, t)$ into $(t, \leq, t')$, the second rule moves the type t' from the right of $\leq$ to the left, yielding $(t \wedge \neg t', \leq, \mathbb{0})$, and finally the last rule puts $t \wedge \neg t'$ in disjunctive normal form. Such a type τ is the type to be normalized. If τ is a union of single normal forms, the rule (NUNION) splits the union of single normal forms into constraints featuring each of the single normal forms. Then the results of each constraint normalization are joined by the rule (NJOIN).

The following rules handle constraints of the form $(\tau, \leq, \mathbb{0})$, where τ is a single normal form. If there are some top-level type variables, the rule (NTLV) generates a normalized constraint for the top-level type variable whose order is the smallest. Otherwise, there are no top-level type variables. If τ has already been normalized (i.e., it belongs to Σ), then it is not processed again (rule (NHYP)). Otherwise, we memoize it and then process it using the predicate for single normal forms $\Sigma \vdash^{*}_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ (rule (NASSUM)). Note that switching from $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ to $\Sigma \vdash^{*}_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ prevents the incorrect use of (NHYP) just after (NASSUM), which would wrongly say that any type is normalized without doing any computation.

Finally, the last four rules state how to normalize constraints of the form $(\tau, \leq, \mathbb{0})$ where τ is a single normal form and contains no top-level type variables. Thereby τ should be an intersection of atoms with the same constructor. If τ is an intersection of basic types, normalizing is equivalent to checking whether τ is empty or not: if it is (rule (NBASIC-T)), we return the singleton containing the empty set (any substitution is a solution), otherwise there is no solution and we return the empty set (rule (NBASIC-F)). When τ is an intersection of products, the rule (NPROD) decomposes τ into several candidate types (following

$$\frac{}{\Sigma \vdash_{\mathscr{N}} \emptyset \rightsquigarrow \{\emptyset\}}\text{(NEMPTY)} \qquad \frac{\Sigma \vdash_{\mathscr{N}} \{(t_i\ c_i\ t'_i)\} \rightsquigarrow S_i}{\Sigma \vdash_{\mathscr{N}} \{(t_i\ c_i\ t'_i) \mid i \in I\} \rightsquigarrow \bigsqcap_{i\in I} S_i}\text{(NJOIN)}$$

$$\frac{\Sigma \vdash_{\mathscr{N}} \{(t \leq t')\} \rightsquigarrow \mathscr{S}}{\Sigma \vdash_{\mathscr{N}} \{(t' \geq t)\} \rightsquigarrow \mathscr{S}}\text{(NSYM)} \qquad \frac{\Sigma \vdash_{\mathscr{N}} \{(t \wedge \neg t' \leq \mathbb{0})\} \rightsquigarrow \mathscr{S} \qquad t' \neq \mathbb{0}}{\Sigma \vdash_{\mathscr{N}} \{(t \leq t')\} \rightsquigarrow \mathscr{S}}\text{(NZERO)}$$

$$\frac{\Sigma \vdash_{\mathscr{N}} \{(\mathsf{dnf}(t) \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}{\Sigma \vdash_{\mathscr{N}} \{(t \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\text{(NDNF)} \qquad \frac{\Sigma \vdash_{\mathscr{N}} \{(\tau_i \leq \mathbb{0}) \mid i \in I\} \rightsquigarrow \mathscr{S}}{\Sigma \vdash_{\mathscr{N}} \{(\bigvee_{i\in I} \tau_i \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\text{(NUNION)}$$

$$\frac{\mathsf{tlv}(\tau_0) = \emptyset \qquad \alpha' \wr_O P \cup N \qquad \mathscr{S} = \begin{cases} \{\{(\alpha', \leq, \neg t_{\alpha'})\}\} & \alpha' \in P \\ \{\{(\alpha', \geq, t_{\alpha'})\}\} & \alpha' \in N \end{cases}}{\Sigma \vdash_{\mathscr{N}} \{(\bigwedge_{\alpha\in P} \alpha \wedge \bigwedge_{\alpha\in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\text{(NTLV)}$$

$$\frac{\tau_0 \in \Sigma \qquad \mathsf{tlv}(\tau_0) = \emptyset}{\Sigma \vdash_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \{\emptyset\}}\text{(NHYP)} \qquad \frac{\Sigma \cup \{\tau_0\} \vdash^*_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S} \qquad \tau_0 \notin \Sigma \qquad \mathsf{tlv}(\tau_0) = \emptyset}{\Sigma \vdash_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\text{(NASSUM)}$$

$$\frac{\bigwedge_{i\in P} b_i \wedge \bigwedge_{j\in N} \neg b_j \leq \mathbb{0}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i\in P} b_i \wedge \bigwedge_{j\in N} \neg b_j \leq \mathbb{0})\} \rightsquigarrow \{\emptyset\}}\text{(NBASIC-T)} \qquad \frac{\bigwedge_{i\in P} b_i \wedge \bigwedge_{j\in N} \neg b_j \not\leq \mathbb{0}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i\in P} b_i \wedge \bigwedge_{j\in N} \neg b_j \leq \mathbb{0})\} \rightsquigarrow \emptyset}\text{(NBASIC-F)}$$

$$\frac{\forall N' \subseteq N. \begin{cases} \Sigma \vdash_{\mathscr{N}} \{\bigwedge_{i\in P} t^1_i \wedge \bigwedge_{j\in N'} \neg t^1_j \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^1_{N'} \\ \Sigma \vdash_{\mathscr{N}} \{\bigwedge_{i\in P} t^2_i \wedge \bigwedge_{j\in N\setminus N'} \neg t^2_j \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^2_{N'} \end{cases}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i\in P} (t^1_i \times t^2_i) \wedge \bigwedge_{j\in N} \neg(t^1_j \times t^2_j) \leq \mathbb{0})\} \rightsquigarrow \bigsqcap_{N'\subseteq N} (\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})}\text{(NPROD)}$$

$$\frac{\exists j \in N\ \forall P' \subseteq P. \begin{cases} \Sigma \vdash_{\mathscr{N}} \{t^1_j \wedge \bigwedge_{i\in P'} \neg t^1_i \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^1_{P'} \\ \begin{cases} \Sigma \vdash_{\mathscr{N}} \{\bigwedge_{i\in P\setminus P'} t^2_i \wedge \neg t^2_j \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^2_{P'} & P' \neq P \\ \mathscr{S}^2_{N'} = \emptyset & \text{otherwise} \end{cases} \end{cases}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i\in P} (t^1_i \to t^2_i) \wedge \bigwedge_{j\in N} \neg(t^1_j \to t^2_j) \leq \mathbb{0})\} \rightsquigarrow \bigsqcup_{j\in N} \bigsqcap_{P'\subseteq P} (\mathscr{S}^1_{P'} \sqcup \mathscr{S}^2_{P'})}\text{(NARROW)}$$

where t and t with scripts are types, c_i belongs to $\{\leq, \geq\}$, τ_0 and τ_i are single normal forms, $\alpha \wr_O P \cup N$ denotes α has the smallest order in $P \cup N$ under the ordering O, and $t_{\alpha'}$ is the type obtained from $\bigwedge_{\alpha\in P} \alpha \wedge \bigwedge_{\alpha\in N} \neg\alpha \wedge \tau_0$ by eliminating α'.

Figure 7. Normalization rules

Lemma 3.11 in [4]), which are to be further normalized. The case when τ is an intersection of arrows (rule (NARROW)) is treated similarly. Note that, in the last two rules, we switch from $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ back to $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ in the premises to ensure termination.

If $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, then $\mathscr{S}$ is the result of the normalization of C. We now prove soundness, completeness, and termination of the constraint normalization algorithm.

To prove soundness, we use a family of subtyping relations $\leq_n$ that layer $\leq$[9] (*i.e.*, such that $\bigcup_{n\in\mathscr{N}} \leq_n = \leq$) and a family of satisfaction predicates $\Vdash_n$ that layer $\Vdash$ (*i.e.*, such that $\bigcup_{n\in\mathscr{N}} \Vdash_n = \Vdash$), which are defined as follows.

Definition B.6. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support $(\llbracket\_\rrbracket, \mathscr{D})$. We define the family $(\leq_n)_{n\in\mathscr{N}}$ of subtyping relations as*

$$t \leq_n s \overset{\text{def}}{\iff} \forall \eta.\ \llbracket t \rrbracket_n \eta \subseteq \llbracket s \rrbracket_n \eta$$

where $\llbracket\_\rrbracket_n$ is the rank n interpretation of a type, defined as

$$\llbracket t \rrbracket_n \eta = \{d \in \llbracket t \rrbracket \eta \mid \mathsf{height}(d) \leq n\}$$

[9] See [4] for the definitions of the notions of models, interpretations, and assignments

and height(d) *is the height of an element* d *in* $\mathscr{D}$*, defined as*

$$\begin{array}{rl} \mathsf{height}(c) & = 1 \\ \mathsf{height}((d, d')) & = max(\mathsf{height}(d), \mathsf{height}(d')) + 1 \\ \mathsf{height}(\{(d_1, d'_1), \ldots, (d_n, d'_n)\}) & = \begin{cases} 1 & n = 0 \\ max(\mathsf{height}(d_i), \mathsf{height}(d'_i), \ldots) + 1 & n > 0 \end{cases} \end{array}$$

Lemma B.7. *Let* $\leq$ *be the subtyping relation induced by a well-founded convex model with infinite support. Then*

(1) $t \leq_0 s$ *for all* $t, s \in \mathscr{T}$.
(2) $t \leq s \iff \forall n.\, t \leq_n s$.
(3)

$$\bigwedge_{i\in I}(t_i \times s_i) \leq_{n+1} \bigvee_{j\in J}(t_j \times s_j) \iff \forall J' \subseteq J\,.\, \begin{cases} \bigwedge_{i\in I} t_i \leq_n \bigvee_{j\in J'} t_j \\ \vee \\ \bigwedge_{i\in I} s_i \leq_n \bigvee_{j\in J\setminus J'} s_j \end{cases}$$

(4)

$$\bigwedge_{i\in I}(t_i \to s_i) \leq_{n+1} \bigvee_{j\in J}(t_j \to s_j) \iff \exists j_0{\in}J\,.\forall I'{\subseteq}I\,.\, \begin{cases} t_{j_0} \leq_n \bigvee_{i\in I'} t_i \\ \vee \\ \begin{cases} I \neq I' \\ \wedge \\ \bigwedge_{i\in I\setminus I'} s_i \leq_n s_{j_0} \end{cases} \end{cases}$$

Proof. (1) straightforward.
(2) straightforward.
(3) the result follows by Lemma 3.11 in [4] and Definition B.6.
(4) the result follows by Lemma 3.12 in [4] and Definition B.6.

□

Definition B.8. *Given a constraint-set* C *and a type substitution* σ*, we define the rank* n *satisfaction predicate* $\Vdash_n$ *as*

$$\sigma \Vdash_n C \overset{\text{def}}{\iff} \forall (t, \leq, s) \in C.\, t \leq_n s \text{ and } \forall (t, \geq, s) \in C.\, s \leq_n t$$

Lemma B.9. *Let* $\leq$ *be the subtyping relation induced by a well-founded convex model with infinite support. Then*

(1) $\sigma \Vdash_0 C$ *for all* σ *and* C.
(2) $\sigma \Vdash C \iff \forall n.\, \sigma \Vdash_n C$.

Proof. Consequence of Lemma B.7. □

Given a set Σ of types, we write $C(\Sigma)$ for the constraint-set $\{(t, \leq, \mathbb{0}) \mid t \in \Sigma\}$.

Lemma B.10 (Soundness). *Let* C *be a constraint-set. If* $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$*, then for all normalized constraint-set* $C' \in \mathscr{S}$ *and all substitution* σ*, we have* $\sigma \Vdash C' \Rightarrow \sigma \Vdash C$.

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all $C' \in \mathscr{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_n C$.
(2) Assume $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all $C' \in \mathscr{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_{n+1} C$.

Before proving these statements, we explain how the first property implies the lemma. Suppose $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, $C' \in \mathscr{S}$ and $\sigma \Vdash C'$. It is easy to check that $\sigma \Vdash_n C(\emptyset)$ holds for all n. From $\sigma \Vdash C'$, we deduce $\sigma \Vdash_n C'$ for all n (by Lemma B.9). By Property (1), we have $\sigma \Vdash_n C$ for all n, and we have then the required result by Lemma B.9.

We prove these two properties simultaneously by induction on the derivations of $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ and $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$.

(NEMPTY): straightforward.
(NJOIN): according to Definition B.4, if there exists $C_i \in \mathscr{S}_i$ such that $C_i = \emptyset$, then $\bigsqcap_{i\in I} \mathscr{S}_i = \emptyset$, and the result follows immediately. Otherwise, we have $C' = \bigcup_{i\in I} C_i$, where $C_i \in \mathscr{S}_i$. As $\sigma \Vdash_n C'$, then clearly $\sigma \Vdash_n C_i$. By induction, we have $\sigma \Vdash_n \{(t_i\ c_i\ t'_i)\}$. Therefore, we get $\sigma \Vdash_n \{(t_i\ c_i\ t'_i) \mid i \in I\}$.
(NSYM): by induction, we have $\sigma \Vdash_n \{(t \leq t')\}$. Then clearly $\sigma \Vdash_n \{(t' \geq t)\}$.
(NZERO): by induction, we have $\sigma \Vdash_n \{(t \wedge \neg t' \leq \mathbb{0})\}$. According to set-theory, we have $\sigma \Vdash_n \{(t \leq t')\}$.
(NDNF): similar to the case of (NZERO).
(NUNION): similar to the case of (NZERO).

(NTLV): assume α' has the smallest order in $P \cup N$. If $\alpha' \in P$, then we have $C' = (\alpha', \leq, \neg t_{\alpha'})$. From $\sigma \Vdash_n C'$, we deduce $\sigma(\alpha') \leq_n \neg t_{\alpha'}\sigma$. According to set-theory, we have $\sigma(\alpha') \wedge t_{\alpha'}\sigma \leq_n \mathbb{0}$, that is, $\sigma \Vdash_n \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.

(NHYP): since we have $\tau_0 \in \Sigma$ and $\sigma \Vdash_n C(\Sigma)$, then $\sigma \Vdash_n \{(\tau_0 \leq \mathbb{0})\}$ holds.

(NASSUM): if $n = 0$, then $\sigma \Vdash_0 \{(\tau_0 \leq \mathbb{0})\}$ holds. Suppose $n > 0$. From $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_k C'$, it is easy to prove that $\sigma \Vdash_k C(\Sigma)$ (*) and $\sigma \Vdash_k C'$ (**) hold for all $0 \leq k \leq n$. We now prove that $\sigma \Vdash_k \{(\tau_0 \leq \mathbb{0}\})$ (***) holds for all $1 \leq k \leq n$. By definition of $\Vdash_0$, we have $\sigma \Vdash_0 C(\Sigma \cup \{\tau_0\})$ and $\sigma \Vdash_0 C'$. Consequently, by the induction hypothesis (item (2)), we have $\sigma \Vdash_1 \{\tau_0 \leq \mathbb{0}\}$. From this and (*), we deduce $\sigma \Vdash_1 C(\Sigma \cup \{\tau_0\})$. Because we also have $\sigma \Vdash_1 C'$ (by (**)), we can use the induction hypothesis (item (2)) again to deduce $\sigma \Vdash_2 \{(\tau_0 \leq \mathbb{0}\})$. Hence, we can prove (***) by induction on $1 \leq k \leq n$. In particular, we have $\sigma \Vdash_n \{(\tau_0 \leq \mathbb{0}\})$, which is the required result.

(NBASIC): straightforward.

(NPROD): If $\prod_{N' \subseteq N}(\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})$ is $\emptyset$, then the result follows straightforwardly. Otherwise, we have $C' = \bigcup_{N' \subseteq N} C_{N'}$, where $C_{N'} \in (\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})$. Since $\sigma \Vdash_n C'$, we have $\sigma \Vdash_n C_{N'}$ for all subset $N' \subseteq N$. Moreover, following Definition B.4, either $C_{N'} \in \mathscr{S}^1_{N'}$ or $C_{N'} \in \mathscr{S}^2_{N'}$. By induction, we have either $\sigma \Vdash_n \{\bigwedge_{i \in P} t^1_i \wedge \bigwedge_{j \in N'} \neg t^1_j \leq \mathbb{0}\}$ or $\sigma \Vdash_n \{\bigwedge_{i \in P} t^2_i \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j \leq \mathbb{0}\}$. That is, for all subset $N' \subseteq N$, we have

$$\bigwedge_{i \in P} t^1_i\sigma \wedge \bigwedge_{j \in N'} \neg t^1_j\sigma \leq_n \mathbb{0} \text{ or } \bigwedge_{i \in P} t^2_i\sigma \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j\sigma \leq_n \mathbb{0}$$

Applying Lemma B.7, we have

$$\bigwedge_{i \in P} (t^1_i \times t^2_i)\sigma \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j)\sigma \leq_{n+1} \mathbb{0}$$

Thus, $\sigma \Vdash_{n+1} \{(\bigwedge_{i \in P}(t^1_i \times t^2_i) \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j) \leq \mathbb{0})\}$.

(NARROW): similar to the case of (NPROD).

□

Given a normalized constraint-set C and a set X of type variables, we define the restriction $C|_X$ of C by X to be $\{(\alpha, c, t) \in C \mid \alpha \in X\}$.

Lemma B.11. *Let t be a type and $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathscr{S}$. Then for all normalized constraint-set $C \in \mathscr{S}$, all substitution σ and all n, if $\sigma \Vdash_n C|_{\mathsf{tlv}(t)}$ and $\sigma \Vdash_{n-1} C \setminus C|_{\mathsf{tlv}(t)}$, then $\sigma \Vdash_n \{(t, \leq, \mathbb{0})\}$.*

Proof. By applying the rules (NDNF) and (NUNION), the constraint-set $\{(t, \leq, \mathbb{0})\}$ is normalized into a new constraint-set C', consisting of the constraints of the form $(\tau, \leq, \mathbb{0})$, where τ is a single normal form. That is, $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \{C'\}$. Let $C'_1 = \{(\tau, \leq, \mathbb{0}) \in C' \mid \mathsf{tlv}(\tau) \neq \emptyset\}$ and $C'_2 = C' \setminus C'_1$. It is easy to deduce that all the constraints in $C \setminus C|_{\mathsf{tlv}(t)}$ are generated from C'_2 and must pass at least one instance of $\vdash^*_{\mathscr{N}}$ (*i.e.*, being decomposed at least once). Since $\sigma \Vdash_{n-1} C \setminus C|_{\mathsf{tlv}(t)}$, then according to the statement (2) in the proof of Lemma B.10, we have $\sigma \Vdash_n C'_2$. Moreover, from $\sigma \Vdash_n C|_{\mathsf{tlv}(t)}$, we have $\sigma \Vdash_n C'_1$. Thus, $\sigma \Vdash_n C'$ and *a fortiori* $\sigma \Vdash_n \{(t, \leq, \mathbb{0})\}$. □

Lemma B.12 (Completeness). *Let C be a constraint-set such that $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all substitution σ, if $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.*

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all σ, if $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.
(2) Assume $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all σ, if $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.

The result is then a direct consequence of the first item (indeed, we have $\sigma \Vdash C(\emptyset)$ for all σ). We prove the two items simultaneously by induction on the derivations of $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ and $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$.

(NEMPTY): straightforward.

(NJOIN): as $\sigma \Vdash \{(t_i\ c_i\ t'_i) \mid i \in I\}$, we have in particular $\sigma \Vdash \{(t_i\ c_i\ t'_i)\}$ for all i. By induction, there exists $C_i \in \mathscr{S}_i$ such that $\sigma \Vdash C_i$. So $\sigma \Vdash \bigcup_{i \in I} C_i$. Moreover, according to Definition B.4, $\bigcup_{i \in I} C_i$ must be in $\prod_{i \in I} \mathscr{S}_i$. Therefore, the result follows.

(NSYM): if $\sigma \Vdash \{(t' \geq t)\}$, then $\sigma \Vdash \{(t \leq t')\}$. By induction, the result follows.

(NZERO): since $\sigma \Vdash \{(t \leq t')\}$, according to set-theory, $\sigma \Vdash \{(t \wedge \neg t' \leq \mathbb{0})\}$. By induction, the result follows.

(NDNF): similar to the case of (NZERO).

(NUNION): similar to the case of (NZERO).

(NTLV): assume α' has the smallest order in $P \cup N$. If $\alpha' \in P$, then according to set-theory, we have $\alpha'\sigma \leq \neg(\bigwedge_{\alpha \in (P \setminus \{\alpha'\})} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0)$, that is $\sigma \Vdash \{(\alpha' \leq \neg t_{\alpha'})\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.

(NHYP): it is clear that $\sigma \Vdash \emptyset$.

(NASSUM): as $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash \{(\tau_0 \leq \mathbb{0})\}$, we have $\sigma \Vdash C(\Sigma \cup \{\tau_0\})$. By induction, the result follows.

(NBASIC): straightforward.

(NPROD): as

$$\sigma \Vdash \{(\bigwedge_{i \in P} (t_i^1 \times t_i^2) \wedge \bigwedge_{j \in N} \neg(t_j^1 \times t_j^2) \leq \mathbb{0})\}$$

we have

$$\bigwedge_{i \in P} (t_i^1 \times t_i^2)\sigma \wedge \bigwedge_{j \in N} \neg(t_j^1 \times t_j^2)\sigma \leq \mathbb{0}$$

Applying Lemma 3.11 in [4], for all subset $N' \subseteq N$, we have

$$\bigwedge_{i \in P} t_i^1\sigma \wedge \bigwedge_{j \in N'} \neg t_j^1\sigma \leq \mathbb{0} \text{ or } \bigwedge_{i \in P} t_i^2\sigma \wedge \bigwedge_{j \in N \setminus N'} \neg t_j^2\sigma \leq \mathbb{0}$$

that is,

$$\sigma \Vdash \{(\bigwedge_{i \in P} t_i^1 \wedge \bigwedge_{j \in N'} \neg t_j^1 \leq \mathbb{0})\} \text{ or } \sigma \Vdash \{(\bigwedge_{i \in P} t_i^2 \wedge \bigwedge_{j \in N \setminus N'} \neg t_j^2 \leq \mathbb{0})\}$$

By induction, either there exists $C_{N'}^1 \in \mathscr{S}_{N'}^1$ such that $\sigma \Vdash C_{N'}^1$ or there exists $C_{N'}^2 \in \mathscr{S}_{N'}^2$ such that $\sigma \Vdash C_{N'}^2$. According to Definition B.4, we have $C_{N'}^1, C_{N'}^2 \in \mathscr{S}_{N'}^1 \sqcup \mathscr{S}_{N'}^2$. Thus there exists $C'_{N'} \in \mathscr{S}_{N'}^1 \sqcup \mathscr{S}_{N'}^2$ such that $\sigma \Vdash C'_{N'}$. Therefore $\sigma \Vdash \bigcup_{N' \subseteq N} C'_{N'}$. Moreover, according to Definition B.4 again, $\bigcup_{N' \subseteq N} C'_{N'} \in \prod_{N' \subseteq N} (\mathscr{S}_{N'}^1 \sqcup \mathscr{S}_{N'}^2)$. Hence, the result follows.

(NARROW): similar to the case (NPROD) except we use Lemma 3.12 in [4].

□

We now prove termination of the algorithm.

Definition B.13 (Plinth). *A* plinth $\beth \subset \mathscr{T}$ *is a set of types with the following properties:*

- $\beth$ *is finite;*
- $\beth$ *contains* $\mathbb{1}$, $\mathbb{0}$ *and is closed under Boolean connectives* $(\wedge, \vee, \neg)$*;*
- *for all types* $(t_1 \times t_2)$ *or* $(t_1 \rightarrow t_2)$ *in* $\beth$*, we have* $t_1 \in \beth$ *and* $t_2 \in \beth$*.*

As stated in [11], every finite set of types is included in a plinth. Indeed, we already know that for a regular type t the set of its subtrees S is finite. The definition of the plinth ensures that the closure of S under Boolean connective is also finite. Moreover, if t belongs to a plinth $\beth$, then the set of its subtrees is contained in $\beth$. This is used to show the termination of algorithms working on types.

Lemma B.14 (Termination). *Let C be a finite constraint-set. Then the normalization of C terminates.*

Proof. Let T be the set of type occurring in C. As C is finite, T is finite as well. Let $\beth$ be a plinth such that $T \subseteq \beth$. Then when we normalize a constraint $(t \leq \mathbb{0})$ during the process of $\emptyset \vdash_{\mathscr{N}} C$, t would belong to $\beth$. We prove the lemma by induction on $(|\beth \setminus \Sigma|, U, |C|)$ lexicographically ordered, where Σ is the set of types we have normalized, U is the number of unions $\vee$ occurring in the constraint-set C plus the number of constraints $(t, \geq, s)$ and the number of constraint $(t, \leq, s)$ where $s \neq \mathbb{0}$ or t is not in disjunctive normal form, and C is the constraint-set to be normalized.

(NEMPTY): it terminates immediately.

(NJOIN): $|C|$ decreases.

(NSYM): U decreases.

(NZERO): U decreases.

(NDNF): U decreases.

(NUNION): although $|C|$ increases, U decreases.

(NTLV): it terminates immediately.

(NHYP): it terminates immediately.

(NASSUM): as $\tau_0 \in \beth$ and $\tau_0 \notin \Sigma$, the number $|\beth \setminus \Sigma|$ decreases.

(NBASIC): it terminates immediately.

(NPROD): although $(|\beth \setminus \Sigma|, U, |C|)$ may not change, the next rule to apply must be one of (NEMPTY), (NJOIN), (NSYM), (NZERO), (NDNF), (NUNION), (NTLV), (NHYP) or (NASSUM). Therefore, either the normalization terminates or the triple decreases in the next step.

(NARROW): similar to Case (NPROD).

□

Lemma B.15 (Finiteness). *Let C be a constraint-set and $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. Then $\mathscr{S}$ is finite.*

Proof. It is easy to prove that each normalizing rule generates a finite set of finite sets of normalized constraints. □

Definition B.16. *Let C be a normalized constraint-set and O an ordering on $\mathsf{var}(C)$. We say C is* well-ordered *if for all normalized constraint $(\alpha, c, t_\alpha) \in C$ and for all $\beta \in \mathsf{tlv}(t_\alpha)$, $O(\alpha) < O(\beta)$ holds.*

Lemma B.17. *Let C be a constraint-set and $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. Then for all normalized constraint-set $C' \in \mathscr{S}$, C' is well-ordered.*

Proof. The only way to generate normalized constraints is Rule (NTLV), where we have selected the normalized constraint for the type variable α whose order is minimum as the representative one, that is, $\forall \beta \in \mathsf{tlv}(t_\alpha) \,.\, O(\alpha) < O(\beta)$. Therefore, the result follows. □

Definition B.18. *A general renaming ρ is a special type substitution that maps each type variable to another (fresh) type variable.*

Lemma B.19. *Let t, s be two types and $[\rho_i]_{i\in I}$, $[\rho_j]_{j\in J}$ two sets of general renamings. Then if $\emptyset \vdash_{\mathscr{N}} \{(s \wedge t, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, then $\emptyset \vdash_{\mathscr{N}} \{((\bigwedge_{j\in J} s\rho_j) \wedge (\bigwedge_{i\in I} t\rho_i), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$.*

Proof. By induction on the number of (NPROD) and (NARROW) used in the derivation of $\emptyset \vdash_{\mathscr{N}} \{(s\wedge\neg t, \leq, \mathbb{0})\}$ and by cases on the disjunctive normal form τ of $s\wedge\neg t$. The failure of the normalization of $(s\wedge t, \leq, \mathbb{0})$ is essentially due to (NBASIC-F), (NPROD) and (NARROW), where there are no top-level type variables to make the type empty.
The case of arrows is a little complicated, as we need to consider more than two types: one type for the negative parts and two types for the positive parts from t and s respectively. Indeed, what we prove is the following stronger statement:

$$\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{k\in K} t_k, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \Longrightarrow \emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{k\in K} (\bigwedge_{i_k\in I_k} t_k\rho_{i_k}), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $|K| \geq 2$ and ρ_{i_k}'s are general renamings. For simplicity, we only consider $|K| = 2$, as it is easy to extend to the case of $|K| > 2$.

<u>Case 1</u>: $\tau = \tau_{b_s} \wedge \tau_{b_t}$ and $\tau \not\simeq \mathbb{0}$, where τ_{b_s} (τ_{b_t} resp.) is an intersection of basic types from s (t resp.). Then the expansion of τ is

$$(\bigwedge_{j\in J} \tau_{b_s}\rho_j) \wedge (\bigwedge_{i\in I} \tau_{b_t}\rho_i) \simeq \tau_{b_s} \wedge \tau_{b_t} \not\simeq \mathbb{0}$$

So $\emptyset \vdash_{\mathscr{N}} \{((\bigwedge_{j\in J} \tau_{b_s}\rho_j) \wedge (\bigwedge_{i\in I} \tau_{b_t}\rho_i), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$.

<u>Case 2</u>: $\tau = \bigwedge_{p_s\in P_s}(w_{p_s}\times v_{p_s}) \wedge \bigwedge_{n_s\in N_s} \neg(w_{n_s}\times v_{n_s}) \wedge \bigwedge_{p_t\in P_t}(w_{p_t}\times v_{p_t}) \wedge \bigwedge_{n_t\in N_t} \neg(w_{n_t}\times v_{n_t})$, where P_s, N_s are from s and P_t, N_t are from t. Since $\emptyset \vdash_{\mathscr{N}} \{\tau, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, by the rule (NPROD), there exist two sets $N'_s \subseteq N_s$ and $N'_t \subseteq N_t$ such that

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{p_s\in P_s} w_{p_s} \wedge \bigwedge_{n_s\in N'_s} \neg w_{n_s} \wedge \bigwedge_{p_t\in P_t} w_{p_t} \wedge \bigwedge_{n_t\in N'_t} \neg w_{n_t}, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{p_s\in P_s} v_{p_s} \wedge \bigwedge_{n_s\in N_s\setminus N'_s} \neg v_{n_s} \wedge \bigwedge_{p_t\in P_t} v_{p_t} \wedge \bigwedge_{n_t\in N_t\setminus N'_t} \neg v_{n_t}, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By induction, we have

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{j\in J}(\bigwedge_{p_s\in P_s} w_{p_s} \wedge \bigwedge_{n_s\in N'_s} \neg w_{n_s})\rho_j \wedge \bigwedge_{i\in I}(\bigwedge_{p_t\in P_t} w_{p_t} \wedge \bigwedge_{n_t\in N'_t} \neg w_{n_t})\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{j\in J}(\bigwedge_{p_s\in P_s} v_{p_s} \wedge \bigwedge_{n_s\in N_s\setminus N'_s} \neg v_{n_s})\rho_j \wedge \bigwedge_{i\in I}(\bigwedge_{p_t\in P_t} v_{p_t} \wedge \bigwedge_{n_t\in N_t\setminus N'_t} \neg v_{n_t})\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

Then by the rule (NPROD) again, we get

$$\emptyset \vdash_{\mathscr{N}} \{\bigwedge_{j\in J}(\tau_s)\rho_j \wedge \bigwedge_{i\in I}(\tau_t)\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $\tau_s = \bigwedge_{p_s\in P_s}(w_{p_s}\times v_{p_s}) \wedge \bigwedge_{n_s\in N_s} \neg(w_{n_s}\times v_{n_s})$ and $\tau_t = \bigwedge_{p_t\in P_t}(w_{p_t}\times v_{p_t}) \wedge \bigwedge_{n_t\in N_t} \neg(w_{n_t}\times v_{n_t})$.

Case 3: $\tau = \bigwedge_{p_s \in P_s}(w_{p_s} \to v_{p_s}) \wedge \bigwedge_{n_s \in N_s} \neg(w_{n_s} \to v_{n_s}) \wedge \bigwedge_{p_t \in P_t}(w_{p_t} \to v_{p_t}) \wedge \bigwedge_{n_t \in N_t} \neg(w_{n_t} \to v_{n_t})$, where P_s, N_s are from s and P_t, N_t are from t. Since $\emptyset \vdash_{\mathscr{N}} \{\tau, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, by the rule (NARROW), for all $w \to v \in N_s \cup N_t$, there exist a set $P'_s \subseteq P_s$ and a set $P'_t \subseteq P_t$ such that

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{ \bigwedge_{p_s \in P'_s} \neg w_{p_s} \wedge \bigwedge_{p_t \in P'_t} \neg w_{p_t} \wedge w, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P'_s = P_s \wedge P'_t = P_t \text{ or } \emptyset \vdash_{\mathscr{N}} \{ \bigwedge_{p_s \in P_s \setminus P'_s} v_{p_s} \wedge \bigwedge_{p_t \in P_t \setminus P'_t} v_{p_t} \wedge \neg v, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By induction, for all $\rho \in [\rho_i]_{i \in I} \cup [\rho_j]_{j \in J}$, we have

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{j \in J}(\bigwedge_{p_s \in P'_s} \neg w_{p_s})\rho_j \wedge \bigwedge_{i \in I}(\bigwedge_{p_t \in P'_t} \neg w_{p_t})\rho_i \wedge w\rho, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ \begin{cases} P'_s = P_s \wedge P'_t = P_t \\ \text{or} \\ \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{j \in J}(\bigwedge_{p_s \in P_s \setminus P'_s} v_{p_s})\rho_j \wedge \bigwedge_{i \in I}(\bigwedge_{p_t \in P_t \setminus P'_t} v_{p_t})\rho_i \wedge \neg v\rho, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases} \end{cases}$$

Then by the rule (NARROW) again, we get

$$\emptyset \vdash_{\mathscr{N}} \{\bigwedge_{j \in J}(\tau_s)\rho_j \wedge \bigwedge_{i \in I}(\tau_t)\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $\tau_s = \bigwedge_{p_s \in P_s}(w_{p_s} \to v_{p_s}) \wedge \bigwedge_{n_s \in N_s} \neg(w_{n_s} \to v_{n_s})$ and $\tau_t = \bigwedge_{p_t \in P_t}(w_{p_t} \to v_{p_t}) \wedge \bigwedge_{n_t \in N_t} \neg(w_{n_t} \to v_{n_t})$.

Case 4: $\tau = (\bigvee_{k_s \in K_s} \tau_{k_s}) \wedge (\bigvee_{k_t \in K_t} \tau_{k_t})$, where τ_{k_s} and τ_{k_t} are single normal forms. As $\emptyset \vdash_{\mathscr{N}} \{(\tau, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, there must exist at least one $k_s \in K_s$ and at least one $k_t \in K_t$ such that $\emptyset \vdash_{\mathscr{N}} \{(\tau_{k_s} \wedge \tau_{k_t}, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$. By Cases $(1) - (3)$, the result follows.

□

The type tallying problem is parameterized with a set Δ of type variables that cannot be instantiated, but so far, we have not considered these monomorphic variables in the normalization procedure. Taking Δ into account affects only the (NTLV) rule, where a normalized constraint is built by singling out a variable α. Since the type substitution σ we want to construct must not touch the type variables in Δ (*i.e.*, $\sigma \sharp \Delta$), we cannot choose a variable α in Δ. To avoid this, we order the variables in C so that those belonging to Δ are always greater than those not in Δ. If, by choosing the minimum top-level variable α, we obtain $\alpha \in \Delta$, it means that all the top-level type variables are contained in Δ. According to Lemmas C.3 and C.11 in the companion paper [3], we can then safely eliminate these type variables. So taking Δ into account, we amend the (NTLV) rule as follows.

$$\frac{\mathsf{tlv}(\tau_0) = \emptyset \qquad \alpha' \,\sharp_O P \cup N \qquad \mathscr{S} = \begin{cases} \{\{(\alpha', \leq, \neg t_{\alpha'})\}\} & \alpha' \in P \setminus \Delta \\ \{\{(\alpha', \geq, t_{\alpha'})\}\} & \alpha' \in N \setminus \Delta \\ \Sigma \vdash_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} & \alpha' \in \Delta \end{cases}}{\Sigma \vdash_{\mathscr{N}} \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}} \text{(NTLV)}$$

Furthermore, it is easy to prove the soundness, completeness, and termination of the algorithm extended with Δ.

B.1.2 Constraint merging

A normalized constraint-set may contain several constraints for a same type variable, which can eventually be merged together. For instance, the constraints $\alpha \geq t_1$ and $\alpha \geq t_2$ can be replaced by $\alpha \geq t_1 \vee t_2$, and the constraints $\alpha \leq t_1$ and $\alpha \leq t_2$ can be replaced by $\alpha \leq t_1 \wedge t_2$. That is to say, we can merge all the lower bounds (resp. upper bounds) of a type variable into only one by unions (resp. intersections).

After repeated uses of the merging rules, a set C contains at most one lower bound constraint and at most one upper bound constraint for each type variable. If both lower and upper bounds exist for a given α, that is, $\alpha \geq t_1$ and $\alpha \leq t_2$ belong to C, then the substitution we want to construct from C must satisfy the constraint $(t_1, \leq, t_2)$ as well. For that, we first normalize the constraint $(t_1, \leq, t_2)$, yielding a set of constraint-sets $\mathscr{S}$, and then saturate C with any normalized constraint-set $C' \in \mathscr{S}$. Formally, we describe the saturation process as the saturation rule $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$, where Σ_p is a set of type pairs (if $(t_1, t_2) \in \Sigma_p$, then the constraint $t_1 \leq t_2$ has already been treated at this point), C_Σ is a normalized constraint-set (which collects the treated original constraints, like $(\alpha, \geq, t_1)$ and $(\alpha, \leq, t_2)$, that generate the additional constraints), C is the normalized constraint-set we want to saturate, and $\mathscr{S}$ is a set of sets of normalized constraints (the result of the saturation of C joined with C_Σ). The saturation rules are given in Figure 9, which describe the same algorithm as *Step 2* of the function merge given in Subsection 3.2.1.

If $\alpha \geq t_1$ and $\alpha \leq t_2$ belongs to the constraint-set C that is being saturated, and $t_1 \leq t_2$ has already been processed (i.e., $(t_1, t_2) \in \Sigma_p$), then the rule (SHYP) simply extends C_Σ (the result of the saturation so far) with $\{\alpha \geq t_1, \alpha \leq t_2\}$. Otherwise, the rule (SASSUM) first normalizes the fresh constraint $\{t_1 \leq t_2\}$,

$$\frac{\forall i \in I \,.\, (\alpha \geq t_i) \in C \qquad |I| \geq 2}{\vdash_{\mathscr{M}} C \rightsquigarrow (C \setminus \{(\alpha \geq t_i) \mid i \in I\}) \cup \{(\alpha \geq \bigvee_{i \in I} t_i)\}}(\text{MLB})$$

$$\frac{\forall i \in I \,.\, (\alpha \leq t_i) \in C \qquad |I| \geq 2}{\vdash_{\mathscr{M}} C \rightsquigarrow (C \setminus \{(\alpha \leq t_i) \mid i \in I\}) \cup \{(\alpha \leq \bigwedge_{i \in I} t_i)\}}(\text{MUB})$$

Figure 8. Merging rules

$$\frac{\Sigma_p, C_\Sigma \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S} \qquad (t_1, t_2) \in \Sigma_p}{\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \rightsquigarrow \mathscr{S}}(\text{SHYP})$$

$$\frac{\begin{array}{l}(t_1, t_2) \notin \Sigma_p \quad \emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S} \\ \mathscr{S}' = \{\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma\} \sqcap \mathscr{S} \\ \forall C' \in \mathscr{S}' \,.\, \Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C' \rightsquigarrow \mathscr{S}_{C'}\end{array}}{\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \rightsquigarrow \bigsqcup_{C' \in \mathscr{S}'} \mathscr{S}_{C'}}(\text{SASSUM})$$

$$\frac{\forall \alpha, t_1, t_2 \nexists \{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C}{\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \{C \cup C_\Sigma\}}(\text{SDONE})$$

where $\Sigma_p, C_\Sigma \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$ means that there exists C' such that $\vdash_{\mathscr{M}} C \rightsquigarrow C'$ and $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C' \rightsquigarrow \mathscr{S}$.

Figure 9. Saturation rules

yielding a set of normalized constraint-sets $\mathscr{S}$. It then saturates (joins) C and C_Σ with each constraint-set $C_{\mathscr{S}} \in \mathscr{S}$, the union of which gives a new set $\mathscr{S}'$ of normalized constrain-sets. Each C' in $\mathscr{S}'$ may contain several constraints for the same type variable, so they have to be merged and saturated themselves. Finally, if C does not contain any couple $\alpha \geq t_1$ and $\alpha \leq t_2$ for a given α, the process is over and the rule (SDONE) simply returns $C \cup C_\Sigma$.

If $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$, then the result of the merging of C is $\mathscr{S}$.

Lemma B.20 (**Soundness**). *Let C be a normalized constraint-set. If $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$, then for all normalized constraint-set $C' \in \mathscr{S}$ and all substitution σ, we have $\sigma \Vdash C' \Rightarrow \sigma \Vdash C$.*

Proof. We prove the following statements.

- Assume $\vdash_{\mathscr{M}} C \rightsquigarrow C'$. For all σ, if $\sigma \Vdash C'$, then $\sigma \Vdash C$.
- Assume $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$. For all σ and $C_0 \in \mathscr{S}$, if $\sigma \Vdash C_0$, then $\sigma \Vdash C_\Sigma \cup C$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$.

(SHYP)**:** by induction, we have $\sigma \Vdash (C_\Sigma \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\}) \cup C$, that is $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$.

(SASSUM)**:** according to Definition B.4, $C_0 \in \mathscr{S}_{C'}$ for some $C' \in \mathscr{S}'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C' \rightsquigarrow \mathscr{S}_{C'}$, we have $\sigma \Vdash C'$. Moreover, the equation $\mathscr{S}' = \{\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma\} \sqcap \mathscr{S}$ gives us $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma \subseteq C'$. Therefore, we have $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$.

(SDONE)**:** straightforward.

□

Lemma B.21 (**Completeness**). *Let C be a normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. Then for all substitution σ, if $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.*

Proof. We prove the following statements.

- Assume $\vdash_{\mathscr{M}} C \rightsquigarrow C'$. For all σ, if $\sigma \Vdash C$, then $\sigma \Vdash C'$.
- Assume $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$. For all σ, if $\sigma \Vdash C_\Sigma \cup C$, then there exists $C_0 \in \mathscr{S}$ such that $\sigma \Vdash C_0$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$.

(SHYP): the result follows by induction.
(SASSUM): as $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$, we have $\sigma \Vdash \{(t_1 \leq t_2)\}$. As $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$, applying Lemma B.12, there exists $C'_0 \in \mathscr{S}$ such that $\sigma \Vdash C'_0$. Let $C' = C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C) \cup C'_0$. Clearly we have $\sigma \Vdash C'$ and $C' \in S'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C' \rightsquigarrow \mathscr{S}_{C'}$, there exists $C_0 \in \mathscr{S}_{C'}$ such that $\sigma \Vdash C_0$. Moreover, it is clear that $C_0 \in \bigsqcup_{C' \in S'} \mathscr{S}_{C'}$. Therefore, the result follows.
(SDONE): straightforward.

□

Lemma B.22 (Termination). *Let C be a finite normalized constraint-set. Then $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C$ terminates.*

Proof. Let T be the set of types occurring in C. As C is finite, T is finite as well. Let $\sqsupset$ be a plinth such that $T \subseteq \sqsupset$. Then when we saturate a fresh constraint $(t_1, \leq, t_2)$ during the process of $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C$, (t_1, t_2) would belong to $\sqsupset \times \sqsupset$. According to Lemma B.14, we know that $\emptyset \vdash_{\mathscr{N}} \{(t_1, \leq, t_2)\}$ terminates. Moreover,the termination of the merging of the lower bounds or the upper bounds of a same type variable is straightforward. Finally, we have to prove termination of the saturation process. The proof proceeds by induction on $(|(\sqsupset \times \sqsupset)| - |\Sigma_p|, |C|)$ lexicographically ordered:

$(Shyp)$: $|C|$ decreases.
$(Sassum)$: as $(t_1, t_2) \notin \Sigma_p$ and $t_1, t_2 \in \sqsupset$, $|(\sqsupset \times \sqsupset)| - |\Sigma_p|$ decreases.
$(Sdone)$: it terminates immediately.

□

Definition B.23 (Sub-constraint). *Let $C_1, C_2 \subseteq \mathscr{C}$ be two normalized constraint-sets. We say C_1 is a* sub-constraint *of C_2, denoted as $C_1 \lessdot C_2$, if for all $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ such that $t' \; c \; t$, where $c \in \{\leq, \geq\}$.*

Lemma B.24. *Let $C_1, C_2 \subseteq \mathscr{C}$ be two normalized constraint-sets and $C_1 \lessdot C_2$. Then for all substitution σ, if $\sigma \Vdash C_2$, then $\sigma \Vdash C_1$.*

Proof. Considering any constraint $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ and $t' \; c \; t$, where $c \in \{\leq, \geq\}$. Since $\sigma \Vdash C_2$, then $\sigma(\alpha) \; c \; t'\sigma$. Moreover, as $t' \; c \; t$, we have $t'\sigma \; c \; t\sigma$. Thus $\sigma(\alpha) \; c \; t\sigma$. □

Definition B.25. *Let $C \subseteq \mathscr{C}$ be a normalized constraint-set. We say C is* saturated *if for each type variable $\alpha \in \mathsf{dom}(C)$,*

(1) *there exists at most one form $(\alpha \geq t_1) \in C$,*
(2) *there exists at most one form $(\alpha \leq t_2) \in C$,*
(3) *if $(\alpha \geq t_1), (\alpha \leq t_2) \in C$, then $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$ and there exists $C' \in \mathscr{S}$ such that C' is a sub-constraint of C (*i.e., *$C' \lessdot C$).*

Lemma B.26. *Let C be a finite normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. Then for all normalized constraint set $C' \in \mathscr{S}$, C' is saturated.*

Proof. We prove a stronger statement: assume $\Sigma_p, C_\Sigma \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. If

(i) for all $(t_1, t_2) \in \Sigma_p$ there exists $C' \in (\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\})$ such that $C' \lessdot C_\Sigma \cup C$ and
(ii) for all $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C_\Sigma$ the pair (t_1, t_2) is in Σ_p,

then C_0 is saturated for all $C_0 \in \mathscr{S}$.

The proof of conditions (1) and (2) for a saturated constraint-set is straightforward for all $C_0 \in \mathscr{S}$. The proof of the condition (3) proceeds by induction on the derivation $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$ and a case analysis on the last rule used in the derivation.

(SHYP): as $(t_1, t_2) \in \Sigma_p$, the conditions (i) and (ii) hold for the premise. By induction, the result follows.
(SASSUME): take any premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{S}} C'' \rightsquigarrow \mathscr{S}_{C'}$, where $C' \in \mathscr{S}'$ and $\vdash_{\mathscr{M}} C' \rightsquigarrow C''$. For any $(s_1, s_2) \in \Sigma_p$, the condition (i) gives us that there exists $C_0 \in (\emptyset \vdash_{\mathscr{N}} \{(s_1 \leq s_2)\})$ such that $C_0 \lessdot C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$. Since $\mathscr{S}' = C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C) \sqcap \mathscr{S}$, we have $C_0 \lessdot C''$. Moreover, consider (t_1, t_2). As $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$, there exists $C_0 \in \mathscr{S}$ such that $C_0 \lessdot C''$. Thus the condition (i) holds for the premise. Moreover, the condition (ii) holds straightforwardly for premise. By induction, the result follows.
(SDONE): the result follows by the conditions (i) and (ii).

□

Lemma B.27 (**Finiteness**). *Let C be a constraint-set and $\emptyset, \emptyset \vdash_{\mathscr{MS}} C \rightsquigarrow \mathscr{S}$. Then $\mathscr{S}$ is finite.*

Proof. It follows by Lemma B.15. □

Lemma B.28. *Let C be a well-ordered normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathscr{MS}} C \rightsquigarrow \mathscr{S}$. Then for all normalized constraint-set $C' \in \mathscr{S}$, C' is well-ordered.*

Proof. The merging of the lower bounds (or the upper bounds) of a same type variable preserves the orders. The result of saturation is well-ordered by Lemma B.17. □

Normalization and merging may produce redundant constraint-sets. For example, consider the constraint-set $\{(\alpha \times \beta), \leq, (\texttt{Int} \times \texttt{Bool})\}$. Applying the rule (NPROD), the normalization of this set is

$$\{\{(\alpha, \leq, \mathbb{0})\}, \{(\beta, \leq, \mathbb{0})\}, \{(\alpha, \leq, \mathbb{0}), (\beta, \leq, \mathbb{0})\}, \{(\alpha, \leq, \texttt{Int}), (\beta, \leq, \texttt{Bool})\}\}.$$

Clearly each constraint-set is a saturated one. Note that $\{(\alpha, \leq, \mathbb{0}), (\beta, \leq, \mathbb{0})\}$ is redundant, since any solution of this constraint-set is a solution of $\{(\alpha, \leq, \mathbb{0})\}$ and $\{(\beta, \leq, \mathbb{0})\}$. Therefore it is safe to eliminate it. Generally, for any two different normalized constraint sets $C_1, C_2 \in \mathscr{S}$, if $C_1 \lessdot C_2$, then according to Lemma B.24, any solution of C_2 is a solution of C_1. Therefore, C_2 can be eliminated from $\mathscr{S}$.

Definition B.29. *Let $\mathscr{S}$ be a set of normalized constraint-sets. We say that $\mathscr{S}$ is* minimal *if for any two different normalized constraint-sets $C_1, C_2 \in \mathscr{S}$, neither $C_1 \lessdot C_2$ nor $C_2 \lessdot C_1$. Moreover, we say $\mathscr{S} \simeq \mathscr{S}'$ if for all substitution σ such that $\exists C \in \mathscr{S}.\, \sigma \Vdash C \iff \exists C' \in \mathscr{S}'.\, \sigma \Vdash C'$.*

Lemma B.30. *Let C be a well-ordered normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathscr{MS}} C \rightsquigarrow \mathscr{S}$. Then there exists a minimal set $\mathscr{S}_0$ such that $\mathscr{S}_0 \simeq \mathscr{S}$.*

Proof. By eliminating the redundant constraint-sets in $\mathscr{S}$. □

B.1.3 Constraint solving

From constraints to equations. Given a well-ordered saturated constraint-set, we transform it into an equivalent equation system. This shows that the type tallying problem is essentially a unification problem.

Definition B.31 (**Equation system**). *An* equation system *E is a set of equations of the form $\alpha = t$ such that there exists at most one equation in E for every type variable α. We define the domain of E, written $\mathsf{dom}(E)$, as the set $\{\alpha \mid \exists t\,.\, \alpha = t \in E\}$.*

Definition B.32 (**Equation system solution**). *Let E be an equation system. A solution to E is a substitution σ such that*

$$\forall \alpha = t \in E\,.\, \sigma(\alpha) \simeq t\sigma \text{ holds}$$

If σ is a solution to E, we write $\sigma \Vdash E$.

From a normalized constraint-set C, we obtain some explicit conditions for the substitution σ we want to construct from C. For instance, from the constraint $\alpha \leq t$ (resp. $\alpha \geq t$), we know that the type substituted for α must be a subtype of t (resp. a super type of t).

We assume that each type variable $\alpha \in \mathsf{dom}(C)$ has a lower bound t_1 and a upper bound t_2 using, if necessary, the fact that $\mathbb{0} \leq \alpha \leq \mathbb{1}$. Formally, we rewrite C as follows:

$$\begin{cases} t_1 \leq \alpha \leq \mathbb{1} & \text{if } \alpha \geq t_1 \in C \text{ and } \nexists t.\ \alpha \leq t \in C \\ \mathbb{0} \leq \alpha \leq t_2 & \text{if } \alpha \leq t_2 \in C \text{ and } \nexists t.\ \alpha \geq t \in C \\ t_1 \leq \alpha \leq t_2 & \text{if } \alpha \geq t_1, \alpha \leq t_2 \in C \end{cases}$$

We then transform each constraint $t_1 \leq \alpha \leq t_2$ in C into an equation $\alpha = (t_1 \vee \alpha') \wedge t_2$[10], where α' is a fresh type variable. The type $(t_1 \vee \alpha') \wedge t_2$ ranges from t_1 to t_2, so the equation $\alpha = (t_1 \vee \alpha') \wedge t_2$ expresses the constraint that $t_1 \leq \alpha \leq t_2$, as wished. We prove the soundness and completeness of this transformation.

To prove soundness, we define the rank n satisfaction predicate $\Vdash_n$ for equation systems, which is similar to the one for constraint-sets.

Lemma B.33 (**Soundness**). *Let $C \subseteq \mathscr{C}$ be a well-ordered saturated normalized constraint-set and E its transformed equation system. Then for all substitution σ, if $\sigma \Vdash E$ then $\sigma \Vdash C$.*

Proof. Without loss of generality, we assume that each type variable $\alpha \in \mathsf{dom}(C)$ has a lower bound and an upper bound, that is $t_1 \leq \alpha \leq t_2 \in C$. We write $O(C_1) < O(C_2)$ if $O(\alpha) < O(\beta)$ for all $\alpha \in \mathsf{dom}(C_1)$ and all $\beta \in \mathsf{dom}(C_2)$. We first prove a stronger statement:

[10] Or, equivalently, $\alpha = t_1 \vee (\alpha' \wedge t_2)$. Besides, in practice, if only $\alpha \geq t_1$ ($\alpha \leq t_2$ resp.) and all the occurrences of α in the co-domain of the function type are positive (negative resp.), we can use $\alpha = t_1$ ($\alpha = t_2$ resp.) instead, and the completeness is ensured by subsumption.

(*) for all σ, n and $C_\Sigma \subseteq C$, if $\sigma \Vdash_n E$, $\sigma \Vdash_n C_\Sigma$, $\sigma \Vdash_{n-1} C \setminus C_\Sigma$, and $O(C \setminus C_\Sigma) < O(C_\Sigma)$, then $\sigma \Vdash_n C \setminus C_\Sigma$.

Here C_Σ denotes the set of constraints that have been checked. The proof proceeds by induction on $|C \setminus C_\Sigma|$.

$C \setminus C_\Sigma = \emptyset$: straightforward.

$C \setminus C_\Sigma \neq \emptyset$: take the constraint $(t_1 \leq \alpha \leq t_2) \in C \setminus C_\Sigma$ such that $O(\alpha)$ is the maximum in $\mathsf{dom}(C \setminus C_\Sigma)$. Clearly, there exists a corresponding equation $\alpha = (t_1 \vee \alpha') \wedge t_2 \in E$. As $\sigma \Vdash_n E$, we have $\sigma(\alpha) \simeq_n ((t_1 \vee \alpha') \wedge t_2)\sigma$. Then,

$$\begin{array}{rcl}\sigma(\alpha) \wedge \neg t_2\sigma & \simeq_n & ((t_1 \vee \alpha') \wedge t_2)\sigma \wedge \neg t_2\sigma \\ & \simeq_n & \mathbb{0}\end{array}$$

Therefore, $\sigma(\alpha) \leq_n t_2\sigma$.
Consider the constraint $(t_1, \leq, \alpha)$. We have

$$\begin{array}{rcl}t_1\sigma \wedge \neg\sigma(\alpha) & \simeq_n & t_1\sigma \wedge \neg((t_1 \vee \alpha') \wedge t_2)\sigma \\ & \simeq_n & t_1\sigma \wedge \neg t_2\sigma\end{array}$$

What remains to do is to check the subtyping relation $t_1\sigma \wedge \neg t_2\sigma \leq_n \mathbb{0}$, that is, to check that the judgement $\sigma \Vdash_n \{(t_1 \leq t_2)\}$ holds. Since the whole constraint-set C is saturated, according to Definition B.25, we have $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$ and there exists $C' \in \mathscr{S}$ such that $C' \lessdot C$, that is $C' \lessdot C_\Sigma \cup C \setminus C_\Sigma$. Moreover, as C is well-ordered, $O(\{\alpha\}) < O(\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2))$ and thus $O(C \setminus C_\Sigma) < O(\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2))$. Therefore, we can deduce that $C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)} \lessdot C_\Sigma$ and $C' \setminus C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)} \lessdot C \setminus C_\Sigma$. From the premise and Lemma B.24, we have $\sigma \Vdash_n C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)}$ and $\sigma \Vdash_{n-1} C' \setminus C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)}$. Then, by Lemma B.11, we get $\sigma \Vdash_n \{(t_1 \leq t_2)\}$.
Finally, consider the constraint-set $C \setminus (C_\Sigma \cup \{(t_1 \leq \alpha \leq t_2)\})$. By induction, we have $\sigma \Vdash_n C \setminus (C_\Sigma \cup \{(t_1 \leq \alpha \leq t_2)\})$. Thus the result follows.

Finally, we explain how to prove the lemma with the statement (*). Take $C_\Sigma = \emptyset$. Since $\sigma \Vdash E$, we have $\sigma \Vdash_n E$ for all n. Trivially, we have $\sigma \Vdash_0 C$. This can be used to prove $\sigma \Vdash_1 C$. Since $\sigma \Vdash_1 E$, by (*), we get $\sigma \Vdash_1 C$, which will be used to prove $\sigma \Vdash_2 C$. Consequently, we can get $\sigma \Vdash_n C$ for all n, which clearly implies the lemma. □

Lemma B.34 (Completeness). *Let $C \subseteq \mathscr{C}$ be a saturated normalized constraint-set and E its transformed equation system. Then for all substitution σ, if $\sigma \Vdash C$ then there exists σ' such that $\sigma' \sharp \sigma$ and $\sigma \cup \sigma' \Vdash E$.*

Proof. Let $\sigma' = \{^{\sigma(\alpha)}/_{\alpha'} \mid \alpha \in \mathsf{dom}(C)\}$. Consider each equation $\alpha = (t_1 \vee \alpha') \wedge t_2 \in E$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. As $\sigma \Vdash C$, then $t_1\sigma \leq \sigma(\alpha)$ and $\sigma(\alpha) \leq t_2\sigma$. Thus

$$\begin{array}{rcll}((t_1 \vee \alpha') \wedge t_2)(\sigma \cup \sigma') & = & (t_1(\sigma \cup \sigma') \vee \alpha'(\sigma \cup \sigma')) \wedge t_2(\sigma \cup \sigma') & \\ & = & (t_1\sigma \vee \sigma(\alpha)) \wedge t_2\sigma & \\ & \simeq & \sigma(\alpha) \wedge t_2\sigma & (t_1\sigma \leq \sigma(\alpha)) \\ & \simeq & \sigma(\alpha) & (\sigma(\alpha) \leq t_2\sigma) \\ & = & (\sigma \cup \sigma')(\alpha) &\end{array}$$

□

Definition B.35. *Let E be an equation system and O an ordering on $\mathsf{dom}(E)$. We say that E is* well ordered *if for all $\alpha = t_\alpha \in E$, we have $O(\alpha) < O(\beta)$ for all $\beta \in \mathsf{tlv}(t_\alpha) \cap \mathsf{dom}(E)$.*

Lemma B.36. *Let C be a well-ordered saturated normalized constraint-set and E its transformed equation system. Then E is well ordered.*

Proof. Clearly, $\mathsf{dom}(E) = \mathsf{dom}(C)$. Consider an equation $\alpha = (t_1 \vee \alpha') \wedge t_2$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. By Definition B.16, for all $\beta \in (\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2)) \cap \mathsf{dom}(C)$. $O(\alpha) < O(\beta)$. Moreover, α' is a fresh type variable in C, that is $\alpha' \notin \mathsf{dom}(C)$. And then $\alpha' \notin \mathsf{dom}(E)$. Therefore, $\mathsf{tlv}((t_1 \vee \alpha') \wedge t_2) \cap \mathsf{dom}(E) = (\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2)) \cap \mathsf{dom}(C)$. Thus the result follows. □

Solution of Equation Systems. We now extract a solution (i.e., a substitution) from the equation system we build from C. In an equation $\alpha = t_\alpha$, α may also appear in the type t_α; such an equality reminds the definition of a recursive type. As a first step, we introduce a recursion operator μ in all the equations of the system, transforming $\alpha = t_\alpha$ into $\alpha = \mu x_\alpha.\ t_\alpha\{^{x_\alpha}/_\alpha\}$. This ensures that type variables do not appear in the right-hand side of the equalities, making the whole solving process easier. If some recursion operators are in fact not needed in the solution (*i.e.*, we have $\alpha = \mu x_\alpha . t_\alpha$ with $x_\alpha \notin \mathsf{fv}(t_\alpha)$), then we can simply eliminate them.

If the equation system contains only one equation, then this equation is immediately a substitution. Otherwise, consider the equation system $\{\alpha = \mu x_\alpha.\ t_\alpha\} \cup E$, where E contains only equations closed with the recursion operator μ as explained above. The next step is to substitute the content expression $\mu x_\alpha.\ t_\alpha$ for all the occurrences of α in equations in E. In detail, let $\beta = \mu x_\beta.\ t_\beta \in E$. Since t_α may contain some occurrences of β and these occurrences are clearly bounded by μx_β, we in fact replace the equation

$\beta = \mu x_\beta.\, t_\beta$ with $\beta = \mu x_\beta.\, t_\beta\{\mu x_\alpha.\, t_\alpha/_\alpha\}\{x_\beta/_\beta\}$, yielding a new equation system E'. Finally, assume that the equation system E' (which has fewer equations) has a solution σ'. Then the substitution $\{t_\alpha \sigma'/_\alpha\} \oplus \sigma'$ is a solution to the original equation system $\{\alpha = \mu x_\alpha.\, t_\alpha\} \cup E$. The solving algorithm Unify() is given in Figure 10.

Require: an equation system E
Ensure: a substitution σ

let $e2mu\ (\alpha, t_\alpha) \ = \ (\alpha,\ \mu x_\alpha.\, t_\alpha\{x_\alpha/_\alpha\})$ **in**
let $subst\ (\alpha, t_\alpha)\ (\beta, t_\beta) \ = \ (\beta,\ t_\beta\{t_\alpha/_\alpha\}\{x_\beta/_\beta\})$ **in**
let rec $mu2sub\ E \ =$
  match E **with**
  $|[\,] \ \to \ [\,]$
  $|(\alpha, t_\alpha) :: E' \ \to$
    let $E'' \ = \ List.map\ (subst\ (\alpha, t_\alpha))\ E'$ **in**
    let $\sigma' \ = \ mu2sub\ E''$ **in** $\{t_\alpha \sigma'/_\alpha\} \oplus \sigma'$
in
let $e2sub\ E \ =$
  let $E' \ = \ List.map\ e2mu\ E$ **in**
  $mu2sub\ E'$

Figure 10. Equation system solving algorithm Unify()

Definition B.37 (General solution). *Let E be an equation system. A* general solution *to E is a substitution σ from* $\mathsf{dom}(E)$ *to $\mathscr{T}$ such that*

$$\forall \alpha \in \mathsf{dom}(\sigma)\ .\ \mathit{var}(\sigma(\alpha)) \cap \mathsf{dom}(\sigma) = \emptyset$$

and

$$\forall \alpha \ = \ t \in E\ .\ \sigma(\alpha) \ \simeq \ t\sigma \text{ holds}$$

Lemma B.38. *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then $\forall \alpha \in \mathsf{dom}(\sigma).\ \mathit{var}(\sigma(\alpha)) \cap \mathsf{dom}(\sigma) = \emptyset$ and $\mathsf{dom}(\sigma) = \mathsf{dom}(E)$.*

Proof. The algorithm Unify() consists of two steps: (i) transform types into recursive types and (ii) extract the substitution. After the first step, for each equation $(\alpha = t_\alpha) \in E$, we have $\alpha \notin \mathsf{var}(t_\alpha)$. Consider the second step. Let $\mathsf{var}(E) = \bigcup_{(\alpha = t_\alpha) \in E} \mathsf{var}(t_\alpha)$ and $\overline{S} = \mathscr{V} \setminus S$, where S is a set of type variables. We prove a stronger statement:

$$\forall \alpha \in \mathsf{dom}(\sigma).\ \mathsf{var}(\sigma(\alpha)) \cap (\mathsf{dom}(\sigma) \cup \overline{\mathsf{var}(E)}) = \emptyset \text{ and } \mathsf{dom}(\sigma) = \mathsf{dom}(E).$$

The proof proceeds by induction on E:

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' \ = \ \{(\beta \ = \ t_\beta\{t_\alpha/_\alpha\}\{x_\beta/_\beta\}) \mid (\beta \ = \ t_\beta) \ \in \ E'\}$. Then there exists a substitution σ'' such that $\sigma'' \ = \ \mathsf{Unify}(E'')$ and $\sigma \ = \ \{t_\alpha \sigma''/_\alpha\} \oplus \sigma''$. By induction, we have $\forall \beta \in \mathsf{dom}(\sigma'').\ \mathsf{var}(\sigma''(\beta)) \cap (\mathsf{dom}(\sigma'') \cup \overline{\mathsf{var}(E'')}) = \emptyset$ and $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. As $\alpha \notin \mathsf{dom}(E'')$, we have $\alpha \notin \mathsf{dom}(\sigma'')$ and then $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\} = \mathsf{dom}(E)$.
Moreover, $\alpha \notin \mathsf{var}(E'')$, then $\mathsf{dom}(\sigma) \subset \mathsf{dom}(\sigma'') \cup \overline{\mathsf{var}(E'')}$. Thus, for all $\beta \in \mathsf{dom}(\sigma'')$, we have $\mathsf{var}(\sigma''(\beta)) \cap \mathsf{dom}(\sigma) = \emptyset$. Consider $t_\alpha \sigma''$. It is clear that $\mathsf{var}(t_\alpha \sigma'') \cap \mathsf{dom}(\sigma) = \emptyset$. Besides, the algorithm does not introduce any fresh variable, then for all $\beta \in \mathsf{dom}(\sigma)$, we have $\mathsf{var}(t_\beta) \cap \overline{\mathsf{var}(E)} = \emptyset$. Therefore, the result follows.

□

Lemma B.39 (Soundness). *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then $\sigma \Vdash E$.*

Proof. By induction on E.

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' \ = \ \{(\beta \ = \ t_\beta\{t_\alpha/_\alpha\}\{x_\beta/_\beta\}) \mid (\beta \ = \ t_\beta) \ \in \ E'\}$. Then there exists a substitution σ'' such that $\sigma'' \ = \ \mathsf{Unify}(E'')$ and $\sigma \ = \ \{t_\alpha \sigma''/_\alpha\} \oplus \sigma''$. By induction, we have

$\sigma'' \Vdash E''$. According to Lemma B.38, we have $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. So $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\}$. Considering any equation $(\beta = t_\beta) \in E$ where $\beta \in \mathsf{dom}(\sigma'')$. Then

$$\begin{array}{rll} \sigma(\beta) = & \sigma''(\beta) & \text{(apply } \sigma) \\ \simeq & t_\beta\{^{t_\alpha}/_\alpha\}\{^{x_\beta}/_\beta\}\sigma'' & \text{(as } \sigma'' \Vdash E'') \\ = & t_\beta\{^{t_\alpha\{^{x_\beta}/_\beta\}}/_\alpha, {}^{x_\beta}/_\beta\}\sigma'' & \\ = & t_\beta\{^{t_\alpha\{^{x_\beta}/_\beta\}\sigma''}/_\alpha, {}^{x_\beta\sigma''}/_\beta\} \oplus \sigma'' & \\ = & t_\beta\{^{t_\alpha(\{^{x_\beta\sigma''}/_\beta\} \oplus \sigma'')}/_\alpha, {}^{x_\beta\sigma''}/_\beta\} \oplus \sigma'' & \\ \simeq & t_\beta\{^{t_\alpha(\{^{t_\beta\sigma''}/_\beta\} \oplus \sigma'')}/_\alpha, {}^{t_\beta\sigma''}/_\beta\} \oplus \sigma'' & \text{(expand } x_\beta) \\ \simeq & t_\beta\{^{t_\alpha(\{^{\beta\sigma''}/_\beta\} \oplus \sigma'')}/_\alpha, {}^{\beta\sigma''}/_\beta\} \oplus \sigma'' & \text{(as } \sigma'' \Vdash E'') \\ = & t_\beta\{^{t_\alpha\sigma''}/_\alpha\} \oplus \sigma'' & \\ = & t_\beta\sigma & \end{array}$$

Finally, consider the equation $(\alpha = t_\alpha)$. As

$$\begin{array}{rll} \sigma(\alpha) = & t_\alpha\sigma'' & \text{(apply } \sigma) \\ = & t_\alpha\{^{\beta\sigma''}/_\beta \mid \beta \in \mathsf{dom}(\sigma'')\} & \text{(expand } \sigma'') \\ = & t_\alpha\{^{\beta\sigma}/_\beta \mid \beta \in \mathsf{dom}(\sigma'')\} & \text{(as } \beta\sigma = \beta\sigma'') \\ = & t_\alpha\{^{\beta\sigma}/_\beta \mid \beta \in \mathsf{dom}(\sigma'') \cup \{\alpha\}\} & \text{(as } \alpha \notin \mathsf{var}(t_\alpha)) \\ = & t_\alpha\{^{\beta\sigma}/_\beta \mid \beta \in \mathsf{dom}(\sigma)\} & \text{(as } \mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\}) \\ = & t_\alpha\sigma & \end{array}$$

Thus, the result follows.

□

Lemma B.40. *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then σ is a general solution to E.*

Proof. Immediate consequence of Lemmas B.38 and B.39. □

Clearly, given an equation system E, the algorithm $\mathsf{Unify}(E)$ terminates with a substitution σ.

Lemma B.41 (Termination). *Given an equation system E, the algorithm $\mathsf{Unify}(E)$ terminates.*

Proof. By induction on the number of equations in E. □

Definition B.42. *Let σ, σ' be two substitutions. We say $\sigma \simeq \sigma'$ if and only if $\forall\alpha.\ \sigma(\alpha) \simeq \sigma'(\alpha)$.*

Lemma B.43 (Completeness). *Let E be an equation system. For all substitution σ, if $\sigma \Vdash E$, then there exist σ_0 and σ' such that $\sigma_0 = \mathsf{Unify}(E)$ and $\sigma \simeq \sigma' \circ \sigma_0$.*

Proof. According to Lemma B.41, there exists σ_0 such that $\sigma_0 = \mathsf{Unify}(E)$. For any $\alpha \notin \mathsf{dom}(\sigma_0)$, clearly we have $\alpha\sigma_0\sigma = \alpha\sigma$ and then $\alpha\sigma_0\sigma \simeq \alpha\sigma$. What remains to prove is that if $\sigma \Vdash E$ and $\sigma_0 = \mathsf{Unify}(E)$ then $\forall\alpha \in \mathsf{dom}(\sigma_0).\ \alpha\sigma_0\sigma \simeq \alpha\sigma$. The proof proceeds by induction on E:

$E = \emptyset$: straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$: let $E'' = \{(\beta = t_\beta\{^{t_\alpha}/_\alpha\}\{^{x_\beta}/_\beta\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma_0 = \{^{t_\alpha\sigma''}/_\alpha\} \oplus \sigma''$. Considering each equation $(\beta = t_\beta\{^{t_\alpha}/_\alpha\}\{^{x_\beta}/_\beta\}) \in E''$, we have

$$\begin{array}{rll} t_\beta\{^{t_\alpha}/_\alpha\}\{^{x_\beta}/_\beta\}\sigma = & t_\beta\{^{t_\alpha\{^{x_\beta}/_\beta\}}/_\alpha, {}^{x_\beta}/_{x_\beta}\}\sigma & \\ = & t_\beta\{^{t_\alpha\{^{x_\beta}/_\beta\}\sigma}/_\alpha, {}^{x_\beta\sigma}/_\beta\} \oplus \sigma & \\ = & t_\beta\{^{t_\alpha(\{^{x_\beta\sigma}/_\beta\} \oplus \sigma)}/_\alpha, {}^{x_\beta\sigma}/_\beta\} \oplus \sigma & \\ \simeq & t_\beta\{^{t_\alpha(\{^{t_\beta\sigma}/_\beta\} \oplus \sigma)}/_\alpha, {}^{t_\beta\sigma}/_\beta\} \oplus \sigma & \text{(expand } x_\beta) \\ \simeq & t_\beta\{^{t_\alpha(\{^{\beta\sigma}/_\beta\} \oplus \sigma)}/_\alpha, {}^{\beta\sigma}/_\beta\} \oplus \sigma & \text{(as } \sigma \Vdash E) \\ = & t_\beta\{^{t_\alpha\sigma}/_\alpha\} \oplus \sigma & \\ \simeq & t_\beta\{^{\alpha\sigma}/_\alpha\} \oplus \sigma & \\ = & t_\beta\sigma & \\ \simeq & \beta\sigma & \end{array}$$

Therefore, $\sigma \Vdash E''$. By induction on E'', we have $\forall\beta \in \mathsf{dom}(\sigma'').\ \beta\sigma''\sigma \simeq \beta\sigma$. According to Lemma B.38, $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. As $\alpha \notin \mathsf{dom}(E'')$, then $\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma'') \cup \{\alpha\}$. Therefore for any $\beta \in \mathsf{dom}(\sigma'') \cap \mathsf{dom}(\sigma_0)$, $\beta\sigma_0\sigma \simeq \beta\sigma''\sigma \simeq \beta\sigma$. Finally, considering α, we have

$$\begin{array}{rll} \alpha\sigma_0\sigma = & t_\alpha\sigma''\sigma & \text{(apply } \sigma_0) \\ = & t_\alpha\{^{\beta\sigma''}/_\beta \mid \beta \in \mathsf{dom}(\sigma'')\}\sigma & \text{(expand } \sigma'') \\ = & t_\alpha\{^{\beta\sigma''\sigma}/_\beta \mid \beta \in \mathsf{dom}(\sigma'')\} \oplus \sigma & \\ \simeq & t_\alpha\{^{\beta\sigma}/_\beta \mid \beta \in \mathsf{dom}(\sigma'')\} \oplus \sigma & \text{(as } \forall\beta \in \sigma''.\ \beta\sigma \simeq \beta\sigma''\sigma) \\ = & t_\alpha\sigma & \\ \simeq & \alpha\sigma & \text{(as } \sigma \Vdash E) \end{array}$$

Therefore, the result follows.

□

In our calculus, a type is well-formed if and only if the recursion traverses a constructor. In other words, the recursive variable should not appear at the top level of the recursive content. For example, the type $\mu x.\ x \vee t$ is not well-formed. To make the substitutions usable, we should avoid these substitutions with ill-formed types. Fortunately, this can been done by giving an ordering on the domain of an equation system to make sure that the equation system is well-ordered.

Lemma B.44. *Let E be a well-ordered equation system. If $\sigma = \mathsf{Unify}(E)$, then for all $\alpha \in \mathsf{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.*

Proof. Assume that there exists an ill-formed $\sigma(\alpha)$. That is, $\sigma(\alpha) = \mu x.\ t$ where x occurs at the top level of t. According to the algorithm $\mathsf{Unify}()$, there exists a sequence of equations $(\alpha =)\alpha_0 = t_{\alpha_0}, \alpha_1 = t_{\alpha_1}, \ldots, \alpha_n = t_{\alpha_n}$ such that $\alpha_i \in \mathsf{tlv}(t_{\alpha_{i-1}})$ and $\alpha_0 \in \mathsf{tlv}(t_{\alpha_n})$ where $i \in \{1, \ldots, n\}$ and $n \geq 0$. According to Definition B.35, $O(\alpha_{i-1}) < O(\alpha_i)$ and $O(\alpha_n) < O(\alpha_0)$. Therefore, we have $O(\alpha_0) < O(\alpha_1) < \ldots < O(\alpha_n) < O(\alpha_0)$, which is impossible. Thus the result follows. □

As mentioned above, there may be some useless recursion constructor μ. They can be eliminated by checking whether the recursive variable appears in the content expression or not. Moreover, if a recursive type is empty (which can be checked with the subtyping algorithm), then it can be replaced by $\mathbb{0}$.

B.1.4 The complete algorithm

To conclude, we now describe the solving procedure $\mathsf{Sol}_\Delta(C)$ for the type tallying problem C. We first normalize C into a finite set $\mathscr{S}$ of well-ordered normalized constraint-sets (`Step 1`). If $\mathscr{S}$ is empty, then there are no solutions to C. Otherwise, each constraint-set $C_i \in \mathscr{S}$ is merged and saturated into a finite set $\mathscr{S}_{C_i}$ of well-order saturated normalized constraint-sets (`Step 2`). Then all these sets are collected into another set $\mathscr{S}'$ (*i.e.*, $\mathscr{S}' = \bigsqcup_{C_i \in \mathscr{S}} \mathscr{S}_{C_i}$). If $\mathscr{S}'$ is empty, then there are no solutions to C. Otherwise, for each constraint-set $C'_i \in \mathscr{S}'$, we transform C'_i into an equation system E_i and then construct a general solution σ_i from E_i (`Step 3`). Finally, we collect all the solutions σ_i, yielding a set Θ of solutions to C. We write $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$ if $\mathsf{Sol}_\Delta(C)$ terminates with Θ, and we call Θ the solution of the type tallying problem C.

Theorem B.45 (Soundness). *Let C be a constraint-set. If $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, $\sigma \Vdash C$.*

Proof. Consequence of Lemmas B.10, B.17, B.20, B.26, B.28, B.33 and B.39. □

Theorem B.46 (Completeness). *Let C be a constraint-set and $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then for all substitution σ, if $\sigma \Vdash C$, then there exists $\sigma' \in \Theta$ and σ'' such that $\sigma \approx \sigma'' \circ \sigma'$.*

Proof. Consequence of Lemmas B.12, B.21, B.34 and B.43. □

Theorem B.47 (Termination). *Let C be a constraint-set. Then $\mathsf{Sol}_\Delta(C)$ terminates.*

Proof. Consequence of Lemmas B.14, B.22 and B.41. □

Lemma B.48. *Let C be a constraint-set and $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then*

(1) *Θ is finite.*
(2) *for all $\sigma \in \Theta$ and for all $\alpha \in \mathsf{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.*

Proof.(1): Consequence of Lemmas B.15 and B.27.
(2): Consequence of Lemmas B.17, B.28, B.36 and B.44.

□

B.2 Type-Substitution Inference Algorithm

In Section A, we presented a sound and complete inference system, which is parametric in the decision procedures for $\sqsubseteq_\Delta$, $\amalg^i_\Delta()$, and $\bullet_\Delta$. In this section we tackle the problem of computing these operators. We focus on the application problem $\bullet_\Delta$, since the other two can be solved similarly. Recall that to compute $t \bullet_\Delta s$, we have to find two sets of substitutions $[\sigma_i]_{i \in I}$ and $[\sigma_j]_{j \in J}$ such that $\forall h \in I \cup J.\ \sigma_h \sharp \Delta$ and

$$\bigwedge_{i \in I} t\sigma_i \leq \mathbb{0} \to \mathbb{1} \tag{18}$$

$$\bigwedge_{j \in J} s\sigma_j \leq \mathsf{dom}(\bigwedge_{i \in I} t\sigma_i) \tag{19}$$

This problem is more general than the other two problems. If we are able to decide inequation (19), it means that we are able to decide $s' \sqsubseteq_\Delta t'$ for any s' and t', just by considering t' ground. Therefore we can decide $\sqsubseteq_\Delta$. We can also decide $[\sigma_i]_{i \in I} \Vdash s \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$ for all s, and therefore compute $\amalg^i_\Delta(s)$.

Let the cardinalities of I and J be p and q respectively. We first show that for fixed p and q, we can reduce the application problem to a type tallying problem. Note that if we increase p, the type on the right of Inequality (19) is larger, and if we increase q the type on the left is smaller. Namely, the larger p and q are, the higher the chances that the inequality holds. Therefore, we can search for cardinalities that make the inequality hold by starting from $p = q = 1$, and then by increasing p and q in a dove-tail order until we get a solution. This gives us a semi-decision procedure for the general application problem. In order to ensure termination, we give some heuristics based on the shapes of s and t to set upper bounds for p and q.

B.2.1 Application problem with fixed cardinalities

We explain how to reduce the application problem with fixed cardinalities for I and J to a type tallying problem. Without loss of generality, we can split each substitution σ_k $(k \in I \cup J)$ into two substitutions: a renaming substitution ρ_k that maps each variable in the domain of σ_k into a fresh variable and a second substitution σ'_k such that $\sigma_k = \sigma'_k \circ \rho_k$. The two inequalities then can be rewritten as

$$\bigwedge_{i\in I}(t\rho_i)\sigma'_i \leq \mathbb{0} \to \mathbb{1}$$

$$\bigwedge_{j\in J}(s\rho_j)\sigma'_j \leq \mathsf{dom}(\bigwedge_{i\in I}(t\rho_i)\sigma'_i)$$

The domains of the substitutions σ'_k are pairwise distinct, since they are composed by fresh type variables. We can therefore merge the σ'_k into one substitution $\sigma = \bigcup_{k\in I\cup J}\sigma'_k$. We can then further rewrite the two inequalities as

$$(\bigwedge_{i\in I}(t\rho_i))\sigma \leq \mathbb{0} \to \mathbb{1}$$

$$(\bigwedge_{j\in J}(s\rho_j))\sigma \leq \mathsf{dom}((\bigwedge_{i\in I}(t\rho_i))\sigma)$$

which are equivalent to

$$t'\sigma \leq \mathbb{0} \to \mathbb{1}$$

$$s'\sigma \leq \mathsf{dom}(t'\sigma)$$

where $t' = (\bigwedge_{i\in I} t\rho_i)$ and $s' = (\bigwedge_{j\in J} s\rho_j)$. As $t'\sigma \leq \mathbb{0} \to \mathbb{1}$, then $t'\sigma$ must be a function type. Then according to Lemmas C.12 and C.13 in the companion paper [3], we can reduce these two inequalities to the constraint set[11]:

$$C = \{(t', \leq, \mathbb{0} \to \mathbb{1}), (t', \leq, s' \to \gamma)\}$$

where γ is a fresh type variable. We have reduced the original application problem $t \bullet_\Delta s$ to solving C, which can be done as explained in Section B.1. We write $\mathsf{AppFix}_\Delta(t, s)$ for the algorithm of the application problem with fixed cardinalities $t \bullet_\Delta s$ and $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$ if $\mathsf{AppFix}_\Delta(t, s)$ terminates with Θ.

Lemma B.49. *Let t, s be two types and γ a type variable such that $\gamma \notin \mathit{var}(t) \cup \mathit{var}(s)$. Then for all substitution σ, if $t\sigma \leq s\sigma \to \gamma\sigma$, then $s\sigma \leq \mathsf{dom}(t\sigma)$ and $\sigma(\gamma) \geq t\sigma \cdot s\sigma$.*

Proof. Consider any substitution σ. As $t\sigma \leq s\sigma \to \gamma\sigma$, by Lemma C.12 in the companion paper [3], we have $s\sigma \leq \mathsf{dom}(t\sigma)$. Then by Lemma C.13 in the companion paper [3], we get $\sigma(\gamma) \geq t\sigma \cdot s\sigma$. □

Lemma B.50. *Let t, s be two types and γ a type variable such that $\gamma \notin \mathit{var}(t) \cup \mathit{var}(s)$. Then for all substitution σ, if $s\sigma \leq \mathsf{dom}(t\sigma)$ and $\gamma \notin \mathsf{dom}(\sigma)$, then there exists σ' such that $\sigma' \sharp \sigma$ and $t(\sigma \cup \sigma') \leq (s \to \gamma)(\sigma \cup \sigma')$.*

Proof. Consider any substitution σ. As $s\sigma \leq \mathsf{dom}(t\sigma)$, by Lemma C.13 in the companion paper [3], the type $(t\sigma) \cdot (s\sigma)$ exists and $t\sigma \leq s\sigma \to ((t\sigma) \cdot (s\sigma))$. Let $\sigma' = \{^{(t\sigma)\cdot(s\sigma)}/_{\gamma}\}$. Then

$$\begin{array}{rcl} t(\sigma \cup \sigma') & = & t\sigma \\ & \leq & s\sigma \to ((t\sigma) \cdot (s\sigma)) \\ & = & s\sigma \to \gamma\sigma' \\ & = & (s \to \gamma)(\sigma \cup \sigma') \end{array}$$

□

Note that the solution of the γ introduced in the constraint $(t, \leq, s \to \gamma)$ represents a result type for the application of t to s. In particular, completeness for the tallying problem ensures that each solution will assign to γ (which occurs in a covariant position) the minimum type for that solution. So the minimum solutions for γ are in $t \bullet_\Delta s$ (see the substitution $\sigma'(\gamma) = (t\sigma) \cdot (s\sigma)$ in the proof of Lemma B.50).

Theorem B.51 (Soundness). *Let t and s be two types. If $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, we have $t\sigma \leq \mathbb{0} \to \mathbb{1}$ and $s\sigma \leq \mathsf{dom}(t\sigma)$.*

Proof. Consequence of Lemmas B.49 and B.45. □

[11] The first constraint $(t', \leq, \mathbb{0} \to \mathbb{1})$ can be eliminated since it is implied by the second one.

Theorem B.52 (**Completeness**). *Let t and s be two types and* $\mathsf{AppFix}_\Delta(t,s) \rightsquigarrow \Theta$*. For all substitution σ, if $t\sigma \leq \mathbb{0} \to \mathbb{1}$ and $s\sigma \leq \mathsf{dom}(t\sigma)$, then there exists $\sigma' \in \Theta$ and σ'' such that $\sigma \simeq \sigma'' \circ \sigma'$.*

Proof. Consequence of Lemmas B.50 and B.46. □

B.2.2 General application problem

Now we take the cardinalities of I and J into account to solve the general application problem. We start with I and J both of cardinality 1 and explore all the possible combinations of the cardinalities of I and J by, say, a dove-tail order until we get a solution. More precisely, the algorithm consists of two steps:

Step A: we generate a constraint set as explained in Section B.2.1 and apply the tallying solving algorithm described in Section B.1, yielding either a solution or a failure.

Step B: if all attempts to solve the constraint sets have failed at `Step 1` of the tallying solving algorithm given at the beginning of Section B.1.1, then fail (the expression is not typable). If they all failed but at least one did not fail in `Step 1`, then increment the cardinalities I and J to their successor in the dove-tail order and start from `Step A` again. Otherwise all substitutions found by the algorithm are solutions of the application problem.

Notice that the algorithm returns a failure only if the solving of the constraint-set fails at `Step 1` of the algorithm for the tallying problem. The reason is that up to `Step 1` all the constraints at issue are on distinct occurrences of type variables: if they fail there is no possible expansion that can make the constraint-set satisfiable (see Lemma B.53). For example, the function `map` can not be applied to any integer, as the normalization of $\{(\texttt{Int}, \leq, \alpha \to \beta)\}$ is empty (and even for any expansion of $\alpha \to \beta$). In `Step 2` instead constraints of different occurrences of a same variable are merged. Thus even if the constraints fail it may be the case that they will be satisfied by expanding different occurrences of a same variable into different variables. Therefore an expansion is tried. For example, consider the application of a function of type $((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})) \to t$ to an argument of type $\alpha \to \alpha$. We start with the constraint

$$(\alpha \to \alpha, \leq, (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})).$$

The tallying algorithm first normalizes it into the set

$$\{(\alpha, \leq, \texttt{Int}), (\alpha, \geq, \texttt{Int}), (\alpha, \leq, \texttt{Bool}), (\alpha, \geq, \texttt{Bool})\}\ (\texttt{Step 1}).$$

But it fails at `Step 2` as neither $\texttt{Int} \leq \texttt{Bool}$ nor $\texttt{Bool} \leq \texttt{Int}$ hold. However, if we expand $\alpha \to \alpha$, the constraint to be solved becomes

$$((\alpha_1 \to \alpha_1) \wedge (\alpha_2 \to \alpha_2), \leq, (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}))$$

and one of the constraint-set of its normalization is

$$\{(\alpha_1, \leq, \texttt{Int}), (\alpha_1, \geq, \texttt{Int}), (\alpha_2, \leq, \texttt{Bool}), (\alpha_2, \geq, \texttt{Bool})\}$$

The conflict between `Int` and `Bool` disappears and we can find a solution to the expanded constraint.

Note that we keep trying expansion without giving any bound on the cardinalities I and J, so the procedure may not terminate, which makes it only a semi-algorithm. The following lemma justifies why we do not try to expand if normalization (i.e., Step 1 of the tallying algorithm) fails.

Lemma B.53. *Let t, s be two types, γ a fresh type variable and $[\rho_i]_{i\in I}$, $[\rho_j]_{j\in J}$ two sets of general renamings. If $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1}), (t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$, then $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1}), (\bigwedge_{i\in I} t\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$.*

Proof. As $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1}), (t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$, then either $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$ or $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$. If the first one holds, then according to Lemma B.19, we have $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$, and *a fortiori*

$$\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1}), (\bigwedge_{i\in I} t\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$$

Assume that $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$. Without loss of generality, we consider the disjunctive normal form τ of t:

$$\tau = \bigvee_{k_b \in K_b} \tau_{k_b} \vee \bigvee_{k_p \in K_p} \tau_{k_p} \vee \bigvee_{k_a \in K_a} \tau_{k_a}$$

where τ_{k_b} (τ_{k_p} and τ_{k_a} resp.) is an intersection of basic types (products and arrows resp.) and type variables. Then there must exist $k \in K_b \cup K_p \cup K_a$ such that $\emptyset \vdash_{\mathscr{N}} \{(\tau_k, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$. If $k \in K_b \cup K_p$, then the constraint $(\tau_k, \leq, s \to \gamma)$ is equivalent to $(\tau_k, \leq, \mathbb{0})$. By Lemma B.19, we get $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, that is, $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$. So the result follows.

Otherwise, it must be that $k \in K_a$ and $\tau_k = \bigwedge_{p\in P}(w_p \to v_p) \wedge \bigwedge_{n\in N} \neg(w_n \to v_n)$. We claim that $\emptyset \vdash_{\mathscr{N}} \{(\tau_k, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$ (otherwise, $\emptyset \vdash_{\mathscr{N}} \{(\tau_k, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$ does not hold). Applying Lemma B.19 again, we get $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$. Moreover, following the rule (NARROW), there exists a set

$P' \subseteq P$ such that

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{ \bigwedge_{p \in P'} \neg w_p \wedge s, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P' = P \text{ or } \emptyset \vdash_{\mathscr{N}} \{ \bigwedge_{p \in P \setminus P'} v_p \wedge \neg\gamma, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

Applying B.19, we get

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{i \in I}(\bigwedge_{p \in P'} \neg w_p)\rho_i \wedge \bigwedge_{j \in J} s\rho_j, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P' = P \text{ or } \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{i \in I}(\bigwedge_{p \in P \setminus P'} v_p)\rho_i \wedge \neg\gamma, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By the rule (NARROW), we have

$$\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i \in I}(\bigwedge_{p \in P} (w_p \to v_p))\rho_i, \leq, (\bigwedge_{j \in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$$

Therefore, we have $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i \in I} \tau_k \rho_i, \leq, (\bigwedge_{j \in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$. So the result follows.

□

Let $\mathsf{App}_\Delta(t, s)$ denote the semi-algorithm for the general application problem.

Theorem B.54. *Let t, s be two types and γ the special fresh type variable introduced in $(\bigwedge_{i \in I} t\sigma_i, \leq, (\bigwedge_{j \in J} s\sigma_j) \to \gamma)$. If $\mathsf{App}_\Delta(t, s)$ terminates with Θ, then*

(1) **(Soundness)** *if $\Theta \neq \emptyset$, then for each $\sigma \in \Theta$, $\sigma(\gamma) \in t \bullet_\Delta s$.*
(2) **(Weak completeness)** *if $\Theta = \emptyset$, then $t \bullet_\Delta s = \emptyset$.*

Proof. (1)**:** consequence of Theorem B.51 and Lemma B.49.
(2)**:** consequence of Lemma B.53.

□

Consider the application `map even`, whose types are

$$\begin{array}{rcl} \texttt{map} & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \texttt{even} & :: & (\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int})) \end{array}$$

We start with the constraint-set

$$C_1 = \{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq ((\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int}))) \to \gamma\}$$

where γ is a fresh type variable (and where we α-converted the type of `map`). Then the algorithm $\mathsf{Sol}_\Delta(C_1)$ generates a set of eight constraint-sets at `Step 2`:

$$\begin{array}{l} \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \texttt{Bool}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \alpha \setminus \texttt{Int}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \texttt{Bool} \vee (\alpha \setminus \texttt{Int})\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \texttt{Bool} \wedge (\alpha \setminus \texttt{Int})\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \texttt{Int}, \beta_1 \geq \texttt{Bool}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \setminus \texttt{Int}, \beta_1 \geq \alpha \setminus \texttt{Int}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \texttt{Int} \vee \alpha, \beta_1 \geq \texttt{Bool} \vee (\alpha \setminus \texttt{Int})\} \end{array}$$

Clearly, the solutions to the 2nd-5th constraint-sets are included in those to the first constraint-set. For the other four constraint-sets, by minimum instantiation, we can get four solutions for γ (*i.e.*, the result types of `map even`): $[\,] \to [\,]$, or $[\texttt{Int}] \to [\texttt{Bool}]$, or $[\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]$, or $[\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]$. Of these solutions only the last two are minimal (the first type is an instance of the third one and the second is an instance of the fourth one) and since both are valid we can take their intersection, yielding the (minimum) solution

$$([\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]) \wedge ([\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]) \qquad (20)$$

Alternatively, we can dully follow the algorithm, perform an iteration, expand the type of the function, yielding the constraint-set

$$\begin{array}{l} \{((\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1]) \wedge ((\alpha_2 \to \beta_2) \to [\alpha_2] \to [\beta_2]) \\ \quad \leq ((\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int}))) \to \gamma\} \end{array}$$

from which we get the type (20) directly.

As stated in Section B.1, we chose an arbitrary ordering on type variables, which affects the generated substitutions and then the resulting types. Assume that σ_1 and σ_2 are two type substitutions generated by different orders. Thanks to the completeness of the tallying problem, there exist σ_1' and σ_2' such that $\sigma_2 \simeq \sigma_1' \circ \sigma_1$ and $\sigma_1 \simeq \sigma_2' \circ \sigma_2$. Therefore, the result types corresponding to σ_1 and σ_2 are equivalent under $\sqsubseteq_\Delta$, that is $\sigma_1(\gamma) \sqsubseteq_\Delta \sigma_2(\gamma)$ and $\sigma_2(\gamma) \sqsubseteq_\Delta \sigma_1(\gamma)$. However, this does not imply that $\sigma_1(\gamma) \simeq \sigma_2(\gamma)$. For example, $\alpha \sqsubseteq_\Delta \mathbb{0}$ and $\mathbb{0} \sqsubseteq_\Delta \alpha$, but $\alpha \not\simeq \mathbb{0}$. Moreover, some result types are easier to understand or more precise than some others. Which one is better is a language design and implementation problem[12]. For example, consider the map even again. The type (20) is obtained under the ordering $o(\alpha_1) < o(\beta_1) < o(\alpha)$. While under the ordering $o(\alpha) < o(\alpha_1) < o(\beta_1)$, we would instead get

$$([\beta \setminus \texttt{Int}] \to [\beta]) \wedge ([\texttt{Int} \vee \texttt{Bool} \vee \beta] \to [\texttt{Bool} \vee \beta]) \tag{21}$$

It is clear that (20) $\sqsubseteq_\emptyset$ (21) and (21) $\sqsubseteq_\emptyset$ (20). However, compared with (20), (21) is less precise and less comprehensible, if we look at the type $[\texttt{Int} \vee \texttt{Bool} \vee \beta] \to [\texttt{Bool} \vee \beta]$: (1) there is a Bool in the domain which is useless here and (2) we know that Int cannot appear in the returned list, but this is not expressed in the type.

There is a final word on completeness, which states that for every solution of the application problem, our algorithm finds a solution that is more general. However this solution is not necessarily the first one found by the algorithm: even if we find a solution, continuing with a further expansion may yield a more general solution. We have just seen that, in the case of map even, the good solution is the second one, although this solution could have already been deduced by intersecting the first minimal solutions we found. Another simple example is the case of the application of a function of type $(\alpha \times \beta) \to (\beta \times \alpha)$ to an argument of type $(\texttt{Int}\times\texttt{Bool}) \vee (\texttt{Bool}\times\texttt{Int})$. For this application our algorithm returns after one iteration the type $(\texttt{Int}\vee\texttt{Bool}) \times (\texttt{Int}\vee\texttt{Bool})$ (since it unifies α with β) while one further iteration allows the system to deduce the more precise type $(\texttt{Int}\times\texttt{Bool})\vee(\texttt{Bool}\times\texttt{Int})$. Of course this raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist? This is a problem we did not tackle in this work, but if the answer to the previous question were negative then it would be easy to prove the existence of a principal type: since at each iteration there are only finitely many solutions, then the principal type would be the intersection of the minimal solutions of the last iteration (how to decide that an iteration is the last one is yet another problem).

B.2.3 Heuristics to stop type-substitution inference

We only have a semi-algorithm for $t \bullet_\Delta s$ because, as long as we do not find a solution, we may increase the cardinalities of I and J (where I and J are defined as in the previous sections) indefinitely. In this section, we propose two heuristic numbers p and q for the cardinalities of I and J that are established according to the form of s and t. These heuristic numbers set the upper limit for the procedure: if no solution is found when the cardinalities of I and J have reached these heuristic numbers, then the procedure stops returning failure. This yields a terminating algorithm for $t \bullet_\Delta s$ which is clearly sound but, in our case, not complete. Whether it is possible to define these boundaries so that they ensure termination *and* completeness is still an open issue.

Through some examples, we first analyze the reasons why one needs to expand the function type t and/or the argument type s: the intuition is that type connectives are what makes the expansions necessary. Then based on this analysis, we give some heuristic numbers for the copies of types that are needed by the expansions. These heuristics follow some simple (but, we believe, reasonable) guidelines. First, when the substitutions found for a given p and q yield a useless type (*e.g.*, "$\mathbb{0} \to \mathbb{0}$" the type of a function that cannot be applied to any value), it seems sensible to expand the types (*i.e.*, increase p or q), in order to find more informative substitutions. Second, if iterating the process does not give a more precise type (in the sense of $\sqsubseteq$), then it seems sensible to stop. Last, when the process continuously yields more and more precise types, we choose to stop when the type is "good enough" for the programmer. In particular we choose to avoid to introduce too many new fresh variables that make the type arbitrarily more precise but at the same time less "programmer friendly". We illustrate these behaviours for three strategies: increasing p (that is, expanding the domain of the function), increasing q (that is, expanding the type of the argument) or lastly increasing both p and q at the same time.

Expansion of t. A simple reason to expand t is the presence of (top-level) unions in s. Generally, it is better to have as many copies of t as there are disjunctions in s. Consider the example,

$$\begin{array}{l} t = (\alpha \to \alpha) \to (\alpha \to \alpha) \\ s = (\texttt{Int} \to \texttt{Int}) \vee (\texttt{Bool} \to \texttt{Bool}) \end{array} \tag{22}$$

If we do not expand t (*ie*, if p is 1), then the result type computed for the application of t to s is $\mathbb{0} \to \mathbb{0}$. However, this result type cannot be applied hereafter, since its domain is $\mathbb{0}$, and is therefore useless (more precisely, it can be applied only to expressions that are provably diverging). When p is 2, we get an extra result type, $(\texttt{Int} \to \texttt{Int}) \vee (\texttt{Bool} \to \texttt{Bool})$, which is obtained by instantiating t twice, by Int and Bool respectively. Carrying on expanding t does not give more precise result types, as we always select only two copies of t to match the two summands in s, according to the decomposition rule for arrows [4].

A different example that shows that the cardinality of the summands in the union type of the argument is a good heuristic choice for p is the following one:

$$\begin{array}{l} t = (\alpha \times \beta) \to (\beta \times \alpha) \\ s = (\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int}) \end{array} \tag{23}$$

Without expansion, the result type is $((\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Bool} \vee \texttt{Int}))$ (α unifies Int and Bool). If we expand t, there exists a more precise result type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$, each summand of which

[12] In the current implementation we assume that the type variables in the function type always have smaller orders than those in the argument type.

Table 1. Heuristic number $H_p(s)$ for the copies of t

Shape of s	**Number** $H_p(s)$
$\bigvee_{i\in I} s_i$	$\Sigma_{i\in I} H_p(s_i)$
$\bigwedge_{i\in P} b_i \wedge \bigwedge_{i\in N} \neg b_i \wedge \bigwedge_{i\in P_1} \alpha_i \wedge \bigwedge_{i\in N_1} \neg\alpha_i$	1
$\bigwedge_{i\in P}(s_i^1 \times s_i^2) \wedge \bigwedge_{i\in N} \neg(s_i^1 \times s_i^2)$	$\Sigma_{N'\subseteq N} H_p(s^1_{N'} \times s^2_{N'})$
$(s_1 \times s_2)$	$H_p(s_1) * H_p(s_2)$
$\bigwedge_{i\in P}(s_i^1 \to s_i^2) \wedge \bigwedge_{i\in N} \neg(s_i^1 \to s_i^2)$	1

where $(s^1_{N'} \times s^2_{N'}) = (\bigwedge_{i\in P} s_i^1 \wedge \bigwedge_{i\in N'} \neg s_i^1 \times \bigwedge_{i\in P} s_i^2 \wedge \bigwedge_{i\in N\setminus N'} \neg s_i^2)$.

corresponds to a different summand in s. Besides, due to the decomposition rule for product types [4], there also exist some other result types which involve type variables, like $((\texttt{Int} \vee \texttt{Bool}) \times \alpha) \vee ((\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Int} \vee \texttt{Bool}) \setminus \alpha)$. Further expanding t makes more product decompositions possible, which may in turn generate new result types. However, the type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$ is informative enough, and so we set the heuristic number to 2, that is, the number of summands in s.

We may have to expand t also because of intersection. First, suppose s is an intersection of basic types; it can be viewed as a single basic type. Consider the example

$$t = \alpha \to (\alpha \times \alpha) \text{ and } s = \texttt{Int} \tag{24}$$

Without expansion, the result type is $\gamma_1 = (\texttt{Int} \times \texttt{Int})$. With two copies of t, besides γ_1, we get another result type $\gamma_2 = (\beta \times \beta) \vee (\texttt{Int} \setminus \beta \times \texttt{Int} \setminus \beta)$, which is more general than γ_1 (*eg*, $\gamma_1 = \gamma_2\{\mathbb{0}/\beta\}$). Generally, with k copies, we get k result types of the form

$$\gamma_k = (\beta_1 \times \beta_1) \vee \ldots \vee (\beta_{k-1} \times \beta_{k-1}) \vee (\texttt{Int} \setminus (\bigvee_{i=1..k-1} \beta_i) \times \texttt{Int} \setminus (\bigvee_{i=1..k-1} \beta_i))$$

It is clear that $\gamma_{k+1} \sqsubseteq_\emptyset \gamma_k$. Moreover, it is easy to find two substitutions $[\sigma_1, \sigma_2]$ such that $[\sigma_1, \sigma_2] \Vdash \gamma_k \sqsubseteq_\emptyset \gamma_{k+1}$ ($k \geq 2$). Therefore, γ_2 is the minimum (with respect to $\sqsubseteq_\emptyset$) of $\{\gamma_k, k \geq 1\}$, so expanding t more than once is useless (we do not get a type more precise than γ_2). However, we think the programmer expects $(\texttt{Int} \times \texttt{Int})$ as a result type instead of γ_2. So we take the heuristic number here as 1.

An intersection of product types is equivalent to $\bigvee_{i\in I}(s_1^i \times s_2^i)$, so we consider just a single product type (and then use union for the general case). For instance,

$$\begin{array}{l} t = ((\alpha \to \alpha) \times (\beta \to \beta)) \to ((\beta \to \beta) \times (\alpha \to \alpha)) \\ s = (((\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd})) \times (\texttt{Bool} \to \texttt{Bool})) \end{array} \tag{25}$$

For the application to succeed, we have a constraint generated for each component of the product type, namely $(\alpha \to \alpha \geq (\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd}))$ and $(\beta \to \beta \geq \texttt{Bool} \to \texttt{Bool})$. As with Example (22), it is better to expand $\alpha \to \alpha$ once for the first constraint, while there is no need to expand $\beta \to \beta$ for the second one. As a result, we expand the whole type t once, and get the result type $((\texttt{Bool} \to \texttt{Bool}) \times ((\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd})))$ as expected. Generally, if the heuristic numbers of the components of a product type are respectively p_1 and p_2, we take $p_1 * p_2$ as the heuristic number for the whole product.

Finally, suppose s is an intersection of arrows, like for example `map even`.

$$\begin{array}{l} t = (\alpha \to \beta) \to [\alpha] \to [\beta] \\ s = (\texttt{Int} \to \texttt{Bool}) \wedge ((\gamma \setminus \texttt{Int}) \to (\gamma \setminus \texttt{Int})) \end{array} \tag{26}$$

When $p = 1$, the constraint to solve is $(\alpha \to \beta \geq s)$. As stated in Subsection B.2.2, we get four possible result types: $[\,] \to [\,]$, $[\texttt{Int}] \to [\texttt{Bool}]$, $[\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]$, or $[\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]$, and we can build the minimum one by taking the intersection of them. If we continue expanding t, any result type we obtain is an intersection of some of the result types we have deduced for $p = 1$. Indeed, assume we expand t so that we get p copies of t. Then we would have to solve either $(\bigvee_{i=1..p} \alpha_i \to \beta_i \geq s)$ or $(\bigwedge_{i=1..p} \alpha_i \to \beta_i \geq s)$. For the first constraint to hold, by the decomposition rule of arrows, there exists i_0 such that $s \leq \alpha_{i_0} \to \beta_{i_0}$, which is the same constraint as for $p = 1$. The second constraint implies $s \leq \alpha_i \to \beta_i$ for all i; we recognize again the same constraint as for $p = 1$ (except that we intersect p copies of it). Consequently, expanding does not give us more information, and it is enough to take $p = 1$ as the heuristic number for this case.

Following the discussion above, we propose in Table 1 a heuristic number $H_p(s)$ that, according to the shape of s, sets an upper bound to the number of copies of t. We assume that s is in normal form. This definition can be easily extended to recursive types by memoization.

The next example shows that performing the expansion of t with $H_p(s)$ copies may not be enough to get a result type, confirming that this number is a heuristic that does not ensure completeness. Let

$$\begin{array}{l} t = ((\texttt{true} \times (\texttt{Int} \to \alpha)) \to t_1) \wedge ((\texttt{false} \times (\alpha \to \texttt{Bool})) \to t_2) \\ s = (\texttt{Bool} \times (\texttt{Int} \to \texttt{Bool})) \end{array} \tag{27}$$

Here $\mathsf{dom}(t)$ is $(\texttt{true} \times (\texttt{Int} \to \alpha)) \vee (\texttt{false} \times (\alpha \to \texttt{Bool}))$. The type s cannot be completely contained in either summand of $\mathsf{dom}(t)$, but it can be contained in $\mathsf{dom}(t)$. Indeed, the first summand requires the substitution of α to be a supertype of `Bool` while the second one requires it to be a subtype of `Int`. As `Bool`

Table 2. Heuristic number $H_q(\mathsf{dom}(t))$ for the copies of s

Shape of $\mathsf{dom}(t)$	**Number** $H_q(\mathsf{dom}(t))$
$\bigvee_{i\in I} t_i$	$\prod_{i\in I} H_q(t_i)+1$
$\bigwedge_{i\in P} b_i \wedge \bigwedge_{i\in N} \neg b_i \wedge \bigwedge_{i\in P_1} \alpha_i \wedge \bigwedge_{i\in N_1} \neg\alpha_i$	1
$\bigwedge_{i\in P}(t_i^1\times t_i^2)\wedge\bigwedge_{i\in N}\neg(t_i^1\times t_i^2)$	$\prod_{N'\subseteq N} H_q(t^1_{N'}\times t^2_{N'})$
$(t_1\times t_2)$	$H_q(t_1)+H_q(t_2)$
$\bigwedge_{i\in P}(t_i^1\to t_i^2)\wedge\bigwedge_{i\in N}\neg(t_i^1\to t_i^2)$	$\lvert P\rvert * (H_q(t_i^1)+H_q(t_i^2))$

where $(t^1_{N'}\times t^2_{N'}) = (\bigwedge_{i\in P} t_i^1 \wedge \bigwedge_{i\in N'}\neg t_i^1 \times \bigwedge_{i\in P} t_i^2 \wedge \bigwedge_{i\in N\setminus N'}\neg t_i^2)$,

is not a subtype of `Int`, to make the application possible, we have to expand the function type at least once. However, according to Table 1, the heuristic number in this case is 1 (*ie*, no expansions).

Expansion of s**.** For simplicity, we assume that $\mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) = \bigvee_{i\in I}\mathsf{dom}(t)\sigma_i$, so that the tallying problem for the application becomes $\bigwedge_{j\in J} s\sigma'_j \leq \bigvee_{i\in I}\mathsf{dom}(t)\sigma_i$. We now give some heuristic numbers for $|J|$ depending on $\mathsf{dom}(t)$.

First, consider the following example where $\mathsf{dom}(t)$ is a union:

$$\begin{array}{rl}\mathsf{dom}(t) = & (\mathtt{Int}\to((\mathtt{Bool}\to\mathtt{Bool})\wedge(\mathtt{Int}\to\mathtt{Int})))\\ & \vee(\mathtt{Bool}\to((\mathtt{Bool}\to\mathtt{Bool})\wedge(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Real}\to\mathtt{Real})))\\ s = & (\mathtt{Int}\to(\alpha\to\alpha))\vee(\mathtt{Bool}\to(\beta\to\beta))\end{array} \qquad (28)$$

For the application to succeed, we need to expand $\mathtt{Int}\to(\alpha\to\alpha)$ with two copies (so that we can make two distinct instantiations $\alpha=\mathtt{Bool}$ and $\alpha=\mathtt{Int}$) and $\mathtt{Bool}\to(\beta\to\beta)$ with three copies (for three instantiations $\beta=\mathtt{Bool}$, $\beta=\mathtt{Int}$, and $\beta=\mathtt{Real}$), corresponding to the first and the second summand in $\mathsf{dom}(t)$ respectively. Since the expansion distributes the union over the intersections, we need to get six copies of s. In detail, we need the following six substitutions: $\{\alpha=\mathtt{Bool},\beta=\mathtt{Bool}\}$, $\{\alpha=\mathtt{Bool},\beta=\mathtt{Int}\}$, $\{\alpha=\mathtt{Bool},\beta=\mathtt{Real}\}$, $\{\alpha=\mathtt{Int},\beta=\mathtt{Bool}\}$, $\{\alpha=\mathtt{Int},\beta=\mathtt{Int}\}$, and $\{\alpha=\mathtt{Int},\beta=\mathtt{Real}\}$, which are the Cartesian products of the substitutions for α and β.

If $\mathsf{dom}(t)$ is an intersection of basic types, we use 1 for the heuristic number. If it is an intersection of product types, we can rewrite it as a union of products and we only need to consider the case of just a single product type. For instance,

$$\begin{array}{rl}\mathsf{dom}(t) = & ((\mathtt{Int}\to\mathtt{Int})\times(\mathtt{Bool}\to\mathtt{Bool}))\\ s = & ((\alpha\to\alpha)\times(\alpha\to\alpha))\end{array} \qquad (29)$$

It is easy to infer that the substitution required by the left component needs α to be `Int`, while the one required by the right component needs α to be `Bool`. Thus, we need to expand s at least once. Assume that $s=(s_1\times s_2)$ and we need q_i copies of s_i with the type substitutions: $\sigma_1^i,\ldots,\sigma_{q_i}^i$. Generally, we can expand the whole product type so that we get $s_1\times s_2$ copies as follows:

$$\begin{array}{rl} & \bigwedge_{j=1..q_1}(s_1\times s_2)\sigma_j^1 \wedge \bigwedge_{j=1..q_2}(s_1\times s_2)\sigma_j^2\\ = & ((\bigwedge_{j=1..q_1} s_1\sigma_j^1\wedge\bigwedge_{j=1..q_2} s_1\sigma_j^2)\times(\bigwedge_{j=1..q_1} s_2\sigma_j^1\wedge\bigwedge_{j=1..q_2} s_2\sigma_j^2))\end{array}$$

Clearly, this expansion type is a subtype of $(\bigwedge_{j=1..q_1} s_1\sigma_j^1 \times \bigwedge_{j=1..q_2} s_2\sigma_j^2)$ and so the type tallying succeeds.

Next, consider the case where $\mathsf{dom}(t)$ is an intersection of arrows:

$$\begin{array}{rl}\mathsf{dom}(t) = & (\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})\\ s = & \alpha\to\alpha\end{array} \qquad (30)$$

Without expansion, we need $(\alpha\to\alpha)\leq(\mathtt{Int}\to\mathtt{Int})$ and $(\alpha\to\alpha)\leq(\mathtt{Bool}\to\mathtt{Bool})$, which reduce to $\alpha=\mathtt{Int}$ and $\alpha=\mathtt{Bool}$; this is impossible. Thus, we have to expand s once, for the two conjunctions in $\mathsf{dom}(t)$.

Note that we may also have to expand s because of unions or intersections occurring under arrows. For example,

$$\begin{array}{rl}\mathsf{dom}(t) = & t'\to((\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}))\\ s = & t'\to(\alpha\to\alpha)\end{array} \qquad (31)$$

As in Example (30), expanding once the type $\alpha\to\alpha$ (which is under an arrow in s) makes type tallying succeed. Because $(t'\to s_1)\wedge(t'\to s_2)\simeq t'\to(s_1\wedge s_2)$, we can in fact perform the expansion on s and then use subsumption to obtain the desired result. Likewise, we may have to expand s if $\mathsf{dom}(t)$ is an arrow type and contains an union in its domain. Therefore, we have to look into $\mathsf{dom}(t)$ and s deeply if they contain both arrow types.

Following these intuitions, we define in Table 2 a heuristic number $H_q(\mathsf{dom}(t))$ that, according to the sharp of $\mathsf{dom}(t)$, sets an upper bound to the number of copies of s.

Together. Up to now, we have considered the expansions of t and s separately. However, it might be the case that the expansions of t and s are interdependent, namely, the expansion of t causes the expansion of s and vice versa. Here we informally discuss the relationship between the two, and hint as why decidability is difficult to prove.

Let $\mathsf{dom}(t) = t_1 \vee t_2$, $s = s_1 \vee s_2$, and suppose the type tallying between $\mathsf{dom}(t)$ and s requires that $t_i\sigma_i \geq s_i$, where σ_1 and σ_2 are two conflicting type substitutions. Then we can simply expand $\mathsf{dom}(t)$ with σ_1 and σ_2, yielding $t_1\sigma_1 \vee t_2\sigma_1 \vee t_1\sigma_2 \vee t_2\sigma_2$. Clearly, this expansion type is a supertype of $t_1\sigma_1 \vee t_2\sigma_2$ and thus a supertype of s. Note that as t is on the bigger side of $\leq$, then the extra chunk of type brought by the expansion (*i.e.*, $t_2\sigma_1 \vee t_1\sigma_2$) does not matter. That is to say, the expansion of t would not cause the expansion of s.

However, the expansion of s could cause the expansion of t, and even a further expansion of s itself. Assume that $s = s_1 \vee s_2$ and s_i requires a different substitution σ_i (*i.e.*, $s_i\sigma_i \leq \mathsf{dom}(t)$ and σ_1 is in conflict with σ_2). If we expand s with σ_1 and σ_2, then we have

$$\begin{array}{ll} & (s_1 \vee s_2)\sigma_1 \wedge (s_1 \vee s_2)\sigma_2 \\ = & (s_1\sigma_1 \wedge s_1\sigma_2) \vee (s_1\sigma_1 \wedge s_2\sigma_2) \vee (s_2\sigma_1 \wedge s_1\sigma_2) \vee (s_2\sigma_1 \wedge s_2\sigma_2) \end{array}$$

It is clear that $s_1\sigma_1 \wedge s_1\sigma_2$, $s_1\sigma_1 \wedge s_2\sigma_2$ and $s_2\sigma_1 \wedge s_2\sigma_2$ are subtypes of $\mathsf{dom}(t)$. Consider the extra type $s_1\sigma_2 \wedge s_2\sigma_1$. If this extra type is empty (e.g., because s_1 and s_2 have different top-level constructors), or if it is a subtype of $\mathsf{dom}(t)$, then the type tallying succeeds. Otherwise, in some sense, we need to solve another type tallying between $s \wedge (s_2\sigma_1 \wedge s_1\sigma_2)$ and $\mathsf{dom}(t)$, which would cause the expansion of t or s. This is the main reason why we fail to prove the decidability of the application problem (that is, deciding $\bullet_\Delta$) so far.

To illustrate this phenomenon, consider the following example:

$$\begin{array}{rl} \mathsf{dom}(t) = & ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \vee (\texttt{Int} \to \texttt{Int})) \to ((\beta \to \beta) \vee (\texttt{Bool} \to \texttt{Bool})) \\ & \vee(\beta \times \beta) \\ s = & (\alpha \to (\texttt{Int} \to \texttt{Int})) \vee ((\texttt{Bool} \to \texttt{Bool}) \to \alpha) \vee (\texttt{Bool} \times \texttt{Bool}) \end{array} \tag{32}$$

Let us consider each summand in s respectively. A solution for the first summand is $\alpha \geq \texttt{Bool} \to \texttt{Bool}$, which corresponds to the first summand in $\mathsf{dom}(t)$. The second one requires $\alpha \leq \texttt{Int} \to \texttt{Int}$ and the third one $\beta \geq \texttt{Bool}$. Since $(\texttt{Bool} \to \texttt{Bool})$ is not subtype of $(\texttt{Int} \to \texttt{Int})$, we need to expand s once, that is,

$$\begin{array}{rl} s' = & s\{{}^{\texttt{Bool} \to \texttt{Bool}}/_{\alpha}\} \wedge s\{{}^{\texttt{Int} \to \texttt{Int}}/_{\alpha}\} \\ = & ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \wedge ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \wedge ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Bool} \to \texttt{Bool})) \wedge ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Bool} \to \texttt{Bool})) \wedge ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee(\texttt{Bool} \times \texttt{Bool}) \end{array}$$

Almost all the summands of s' are contained in $\mathsf{dom}(t)$ except the extra type

$$((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Bool} \to \texttt{Bool})) \wedge ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}))$$

Therefore, we need to consider another type tallying involving this extra type and $\mathsf{dom}(t)$. By doing so, we obtain $\beta = \texttt{Int}$; however we have inferred before that β should be a supertype of $\texttt{Bool}$. Consequently, we need to expand $\mathsf{dom}(t)$; the expansion of $\mathsf{dom}(t)$ with $\{{}^{\texttt{Bool}}/_{\beta}\}$ and $\{{}^{\texttt{Int}}/_{\beta}\}$ makes the type tallying succeed.

In day-to-day examples, the extra type brought by the expansion of s is always a subtype of (the expansion type of) $\mathsf{dom}(t)$, and we do not have to expand $\mathsf{dom}(t)$ or s again. The heuristic numbers we gave seem to be enough in practice.

C. Type reconstruction

We define an *implicit* calculus without interfaces, for which we define a reconstruction system.

Definition C.1. *An* implicit expression m *is an expression without any interfaces (or type substitutions). It is inductively generated by the following grammar:*

$$m ::= c \mid x \mid (m, m) \mid \pi_i(m) \mid m\,m \mid \lambda x.m \mid m{\in}t\,?\,m : m$$

The type reconstruction for expressions has the form $\Gamma \vdash_{\mathscr{R}} e : t \rightsquigarrow \mathscr{S}$, which states that under the typing environment Γ, e has type t if there exists at least one constraint-set C in the set of constraint-sets $\mathscr{S}$ such that C are satisfied. The type reconstruction rules are given in Figure 11.

Most of the rules, except the rules for type cases, are standard but differ from most of the type inference of other work in that they generate a set of constraint-sets rather than a single constraint-set. This is due to the type inference for type-cases. There are four possible cases for type-cases ((RECON-CASE)): (i) if no branch is selected, then the type t_0 inferred for the argument m_0 should be $\mathbb{0}$ (and the result type can be any type); (ii) if the first branch is selected, then the type t_0 should be a subtype of t and the result type α for the whole type-case should be a super-type of the type t_1 inferred for the first branch m_1; (iii) if the second branch is selected, then the type t_0 should be a subtype of $\neg t$ and the result type α should be a super-type of the type t_2 inferred for the second branch m_2; and (iv) both branches are selected, then the result type α should be a super-type of the union of t_1 and t_2 (note that the condition for t_0 is the one that does not satisfy (i), (ii) and (iii)). Therefore, there are four possible solutions for type-cases and thus four possible constraint-sets. Finally, the rule (RECON-CASE-VAR) deals with the type inference for the special binding type-case introduced in Appendix E in the companion paper [3].

$$\frac{}{\Gamma \vdash_{\mathscr{R}} c : b_c \rightsquigarrow \{\varnothing\}}(\text{RECON-CONST}) \qquad \frac{}{\Gamma \vdash_{\mathscr{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}}(\text{RECON-VAR})$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} (m_1, m_2) : t_1 \times t_2 \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2}(\text{RECON-PAIR})$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}}{\Gamma \vdash_{\mathscr{R}} \pi_i(m) : \alpha_i \rightsquigarrow \mathscr{S} \sqcap \{\{(t, \leq, \alpha_1 \times \alpha_2)\}\}}(\text{RECON-PROJ})$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} m_1\, m_2 : \alpha \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1, \leq, t_2 \to \alpha)\}\}}(\text{RECON-APPL})$$

$$\frac{\Gamma, (x : \alpha) \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}}{\Gamma \vdash_{\mathscr{R}} \lambda x.m : \alpha \to \beta \rightsquigarrow \mathscr{S} \sqcap \{\{(t, \leq, \beta)\}\}}(\text{RECON-ABSTR})$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathscr{R}} m_0 : t_0 \rightsquigarrow \mathscr{S}_0 \qquad (m_0 \notin \mathscr{X}) \\ \Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \\ \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2 \\ \mathscr{S} = \quad (\mathscr{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) \\ \quad \sqcup \quad (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0, \leq, t), (t_1, \leq, \alpha)\}\}) \\ \quad \sqcup \quad (\mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \neg t), (t_2, \leq, \alpha)\}\}) \\ \quad \sqcup \quad (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}) \end{array}}{\Gamma \vdash_{\mathscr{R}} (m_0 {\in} t\,?\, m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}(\text{RECON-CASE})$$

$$\frac{\begin{array}{l} \Gamma, (x : \Gamma(x) \wedge t) \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \\ \Gamma, (x : \Gamma(x) \wedge \neg t) \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2 \\ \mathscr{S} = \quad (\{\{(\Gamma(x), \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) \\ \quad \sqcup \quad (\mathscr{S}_1 \sqcap \{\{(\Gamma(x), \leq, t), (t_1, \leq, \alpha)\}\}) \\ \quad \sqcup \quad (\mathscr{S}_2 \sqcap \{\{(\Gamma(x), \leq, \neg t), (t_2, \leq, \alpha)\}\}) \\ \quad \sqcup \quad (\mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(\Gamma(x), \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}) \end{array}}{\Gamma \vdash_{\mathscr{R}} (x \,\epsilon\, t\,?\, m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}(\text{RECON-CASE-VAR})$$

where α, α_i and β in each rule are fresh type variables.

Figure 11. Type reconstruction rules

Let m be an implicit expression such that $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$. By inserting into m those types form of $\alpha \to \beta$ introduced by the derivation of $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$ for the λ-abstractions in m correspondingly, we obtain an explicit expression e for m, denoted as *insert*(m). In particular, for λ-abstraction $\lambda x.\, m$, we have

$$\mathit{insert}(\lambda x.\, m) = \lambda^{\alpha \to \beta} x.\mathit{insert}(m)$$

where $\alpha \to \beta$ is a fresh type introduced for $\lambda x.\, m$.

Theorem C.2 (**Soundness**). *Let m be an implicit expression such that $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$. Then for all $C \in \mathscr{S}$ and for all σ, if $\sigma \Vdash C$, then $\emptyset \,;\, \Gamma\sigma \vdash \mathit{insert}(m)@[\sigma] : t\sigma$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$. We proceed by a case analysis of the last rule used in the derivation.

(RECON-CONST): straightforward .
(RECON-VAR): straightforward.
(RECON-PAIR): consider the following derivation:

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} (m_1, m_2) : t_1 \times t_2 \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2}$$

Since $C \in \mathscr{S}_1 \sqcap \mathscr{S}_2$, according to Definition B.4, there exists $C_1 \in \mathscr{S}_1$ and $C_2 \in \mathscr{S}_2$ such that $C = C_1 \cup C_2$. Thus, we have $\sigma \Vdash C_1$ and $\sigma \Vdash C_2$. By induction, we have $\emptyset \,;\, \Gamma\sigma \vdash \mathit{insert}(m_1)@[\sigma] : t_1\sigma$ and $\emptyset \,;\, \Gamma\sigma \vdash \mathit{insert}(m_2)@[\sigma] : t_2\sigma$. By (*pair*), we get $\emptyset \,;\, \Gamma\sigma \vdash (\mathit{insert}(m_1)@[\sigma], \mathit{insert}(m_2)@[\sigma]) : (t_1\sigma \times t_2\sigma)$, that is $\emptyset \,;\, \Gamma\sigma \vdash \mathit{insert}((m_1, m_2))@[\sigma] : (t_1 \times t_2)\sigma$.

(RECON-PROJ): consider the following derivation:

$$\frac{\Gamma \vdash_{\mathscr{R}} m' : t' \rightsquigarrow \mathscr{S}'}{\Gamma \vdash_{\mathscr{R}} \pi_i(m') : \alpha_i \rightsquigarrow \mathscr{S}' \sqcap \{\{(t', \leq, \alpha_1 \times \alpha_2)\}\}}$$

According to Definition B.4, there exists $C' \in \mathscr{S}'$ such that $C = C' \cup \{(t', \leq, \alpha_1 \times \alpha_2)\}$. Thus, we have $\sigma \Vdash C'$ and $t'\sigma \leq (\alpha_1\sigma \times \alpha_2\sigma)$. By induction, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m')@[\sigma] : t'\sigma$. By subsumption, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m')@[\sigma] : (\alpha_1\sigma \times \alpha_2\sigma)$. Then by (*proj*), we get $\emptyset ⨾ \Gamma\sigma \vdash (\pi_i(\textit{insert}(m')@[\sigma])) : \alpha_i\sigma$, that is $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(\pi_i(m'))@[\sigma] : \alpha_i\sigma$.

(RECON-APPL): consider the following derivation:

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} m_1\, m_2 : \alpha \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1, \leq, t_2 \to \alpha)\}\}}$$

According to Definition B.4, there exists $C_1 \in \mathscr{S}_1$ and $C_2 \in \mathscr{S}_2$ such that $C = C_1 \cup C_2 \cup \{(t_1, \leq, t_2 \to \alpha)\}$. Thus, we have $\sigma \Vdash C_1$, $\sigma \Vdash C_2$ and $t_1\sigma \leq t_2\sigma \to \alpha\sigma$. By induction, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$ and $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_2\sigma$. By subsumption, we can get $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_2\sigma \to \alpha\sigma$. Then by (*appl*), we get $\emptyset ⨾ \Gamma\sigma \vdash (\textit{insert}(m_1)@[\sigma]\ \textit{insert}(m_2)@[\sigma]) : \alpha\sigma$, that is $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_1\, m_2)@[\sigma] : \alpha\sigma$.

(RECON-ABSTR): consider the following derivation:

$$\frac{\Gamma, (x : \alpha) \vdash_{\mathscr{R}} m' : t' \rightsquigarrow \mathscr{S}'}{\Gamma \vdash_{\mathscr{R}} \lambda x.m' : \alpha \to \beta \rightsquigarrow \mathscr{S}' \sqcap \{\{(t', \leq, \beta)\}\}}$$

According to Definition B.4, there exists $C' \in \mathscr{S}'$ such that $C = C' \cup \{(t', \leq, \beta)\}$. Thus, we have $\sigma \Vdash C'$ and $t'\sigma \leq \beta\sigma$. By induction, we have $\emptyset ⨾ \Gamma\sigma, (x : \alpha\sigma) \vdash \textit{insert}(m')@[\sigma] : t'\sigma$. By subsumption, we can get $\emptyset ⨾ \Gamma\sigma, (x : \alpha\sigma) \vdash \textit{insert}(m')@[\sigma] : \beta\sigma$. It is clear that there are no subterms form of $e[\sigma_j]_{j\in J}$ in $\textit{insert}(m')@[\sigma]$, so $\textit{insert}(m')@[\sigma] \sharp \mathsf{var}(\alpha\sigma \to \beta\sigma)$. Then according to weakening (*i.e.*, Lemma B.8 in the companion paper [3]), we have $\mathsf{var}(\alpha\sigma \to \beta\sigma) ⨾ \Gamma\sigma, (x : \alpha\sigma) \vdash \textit{insert}(m')@[\sigma] : \beta\sigma$. Finally, by (*abstr*), we get $\emptyset ⨾ \Gamma\sigma \vdash (\lambda^{\alpha\to\beta}_{[\sigma]} x.\ \textit{insert}(m')) : \alpha\sigma \to \beta\sigma$, that is $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(\lambda x.\ m')@[\sigma] : \alpha\sigma \to \beta\sigma$.

(RECON-CASE): consider the following derivation:

$$\frac{\begin{array}{lll} \Gamma \vdash_{\mathscr{R}} m_0 : t_0 \rightsquigarrow \mathscr{S}_0 & & (m_0 \notin \mathscr{X}) \\ \Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 & & \\ \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2 & & \\ \quad \mathscr{S} = & (\mathscr{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) & \\ \quad\quad \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0, \leq, t'), (t_1, \leq, \alpha)\}\}) & \\ \quad\quad \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \neg t'), (t_2, \leq, \alpha)\}\}) & \\ \quad\quad \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}) & \end{array}}{\Gamma \vdash_{\mathscr{R}} (m_0 {\in} t'\ ?\ m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}$$

Since $C \in \mathscr{S}$, according to Definition B.4, there are four possible cases for C: (i) $C \in \mathscr{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}$, (ii) $C \in \mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0, \leq, t'), (t_1, \leq, \alpha)\}\}$, (iii) $C \in \mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \neg t'), (t_2, \leq, \alpha)\}\}$, and (iv) $C \in \mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}$.

Case (i)**:** there exists $C_0 \in \mathscr{S}_0$ such that $\sigma \Vdash C_0$, $t_0\sigma \leq \mathbb{0}$ and $\mathbb{0} \leq \alpha\sigma$. By induction, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_0)@[\sigma] : t_0\sigma$. Since $t_0\sigma \leq \mathbb{0}$, we have $t_0\sigma \leq \neg t'$ and $t_0\sigma \leq t'$. Then applying the rule (*case*), we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m')@[\sigma]{\in}t'\ ?\ \textit{insert}(m_1)@[\sigma] : \textit{insert}(m_2)@[\sigma] : \mathbb{0}$, that is, $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m'{\in}t'\ ?\ m_1 : m_2)@[\sigma] : \mathbb{0}$. Finally, by subsumption, the result follows.

Case (ii)**:** there exists $C_0 \in \mathscr{S}_0$ and $C_1 \in \mathscr{S}_1$ such that $\sigma \Vdash C_0$, $\sigma \Vdash C_1$, $t_0\sigma \leq t'$ (t' is ground) and $t_1\sigma \leq \alpha\sigma$. By induction, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_0)@[\sigma] : t_0\sigma$ and $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$. If $t_0\sigma \leq \neg t'$, then $t_0\sigma \leq t' \wedge (\neg t') \simeq \mathbb{0}$ (*i.e.*, Case (i)), and thus the result follows by subsumption. Otherwise, we have $t_0\sigma \not\leq \neg t'$. Then applying the rule (*case*), we have

$$\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m')@[\sigma]{\in}t'\ ?\ \textit{insert}(m_1)@[\sigma] : \textit{insert}(m_2)@[\sigma] : t_1\sigma$$

that is, $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m'{\in}t'\ ?\ m_1 : m_2)@[\sigma] : t_1\sigma$. Finally, by subsumption, the result follows.

Case (iii)**:** similar to Case (ii).

Case (iv)**:** there exists $C_0 \in \mathscr{S}_0, C_1 \in \mathscr{S}_1$ and $C_2 \in \mathscr{S}_2$ such that $\sigma \Vdash C_0, \sigma \Vdash C_1$, $\sigma \Vdash C_2$ and $t_1\sigma \vee t_2\sigma \leq \alpha\sigma$. By induction, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_0)@[\sigma] : t_0\sigma$, $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$ and $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_2\sigma$. By subsumption, we have $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma \vee t_2\sigma$ and $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_1\sigma \vee t_2\sigma$. If $t_0\sigma \leq t'$ or $t_0\sigma \leq \neg t'$, then we are in Case $(i) - (iii)$, thus the result follows by subsumption. Otherwise, applying the rule (*case*), we have

$$\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m')@[\sigma]{\in}t'\ ?\ \textit{insert}(m_1)@[\sigma] : \textit{insert}(m_2)@[\sigma] : t_1\sigma \vee t_2\sigma$$

that is, $\emptyset ⨾ \Gamma\sigma \vdash \textit{insert}(m'{\in}t'\ ?\ m_1 : m_2)@[\sigma] : t_1\sigma \vee t_2\sigma$. Finally, by subsumption, the result follows.

(RECON-CASE-VAR): similar to (RECON-CASE).

□

Consider the implicit version of `map`, which can be defined as:

$$\mu m\ \lambda f\ .\ \lambda\ell\ .\ \ell \in \mathtt{nil}\ ?\ \mathtt{nil} : (f(\pi_1\ell), mf(\pi_2\ell))$$

The type inferred for map by the type reconstruction system is $\alpha_1 \to \alpha_2$ and the generated set $\mathscr{S}$ of constraint-sets is:

$$\begin{array}{ll}\{ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathbb{0}, \mathbb{0} \leq \alpha_5\}, \\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathtt{nil}, \mathtt{nil} \leq \alpha_5\}, \\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \neg\mathtt{nil}, (\alpha_6 \times \alpha_9) \leq \alpha_5\} \cup C, \\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathbb{1}, (\alpha_6 \times \alpha_9) \vee \mathtt{nil} \leq \alpha_5\} \cup C \quad \}\end{array}$$

where C is $\{\alpha_1 \leq \alpha_7 \to \alpha_6, \alpha_3 \setminus \mathtt{nil} \leq (\alpha_7 \times \alpha_8), \alpha_1 \to \alpha_2 \leq \alpha_1 \to \alpha_{10}, \alpha_3 \setminus \mathtt{nil} \leq (\alpha_{11} \times \alpha_{12}), \alpha_{10} \leq \alpha_{12} \to \alpha_9\}$. Then applying the tallying algorithm to the sets, we get the following types for map:

$$\begin{array}{l}\alpha_1 \to (\mathbb{0} \to \alpha_5) \\ \alpha_1 \to (\mathtt{nil} \to \mathtt{nil}) \\ \mathbb{0} \to ((\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge \alpha_{12}) \to (\alpha_6 \times \alpha_9)) \\ (\alpha_7 \to \alpha_6) \to (\mathbb{0} \to (\alpha_6 \times \alpha_9)) \\ (\alpha_7 \to \alpha_6) \to (\mathbb{0} \to [\alpha_6]) \\ \mathbb{0} \to ((\mathtt{nil} \vee (\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge \alpha_{12})) \to (\mathtt{nil} \vee (\alpha_6 \times \alpha_9))) \\ (\alpha_7 \to \alpha_6) \to (\mathtt{nil} \to (\mathtt{nil} \vee (\alpha_6 \times \alpha_9))) \\ (\alpha_7 \to \alpha_6) \to ((\mu x.\, \mathtt{nil} \vee (\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge x)) \to [\alpha_6])\end{array}$$

By replacing type variables that only occur positively by $\mathbb{0}$ and those only occurring negatively by $\mathbb{1}$, we obtain

$$\begin{array}{l}\mathbb{1} \to (\mathbb{0} \to \mathbb{0}) \\ \mathbb{1} \to (\mathtt{nil} \to \mathtt{nil}) \\ \mathbb{0} \to ((\mathbb{1} \times \mathbb{1}) \to \mathbb{0}) \\ (\mathbb{0} \to \mathbb{1}) \to (\mathbb{0} \to \mathbb{0}) \\ (\mathbb{1} \to \beta) \to (\mathbb{0} \to [\beta]) \\ \mathbb{0} \to ((\mathtt{nil} \vee (\mathbb{1} \times \mathbb{1})) \to \mathtt{nil}) \\ (\mathbb{0} \to \mathbb{1}) \to (\mathtt{nil} \to \mathtt{nil}) \\ (\alpha \to \beta) \to ([\alpha] \to [\beta])\end{array}$$

All the types, except the last two, are useless [13], as they provide no further information. Thus we deduce the following type for map:

$$((\alpha \to \beta) \to ([\alpha] \to [\beta])) \wedge ((\mathbb{0} \to \mathbb{1}) \to (\mathtt{nil} \to \mathtt{nil}))$$

which is more precise than $(\alpha \to \beta) \to ([\alpha] \to [\beta])$ since it states that the application of map to any function and the empty list returns the empty list.

[13] These useless types are generated from the fact that $\mathbb{0} \to t$ contains all the functions, or the fact that $(\mathbb{0} \times t)$ or $(t \times \mathbb{0})$ is a subtype of any type, or the fact that Case (i) in type-cases is useless in practice.