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
$^3$TEC, Guangdong Power Grid Co., Guangzhou, China

Abstract. This article is the second part of a two articles series about the definition of higher-order polymorphic functions in a type system with recursive types and set-theoretic type connectives (unions, intersections, and negations).

In the first part, presented in a companion paper, we defined and studied the syntax, semantics, and evaluation of the explicitly-typed version of a 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 for function applications, and a type reconstruction system that allows the programmer to omit explicit type annotations for function definitions.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Polymorphism

Keywords Types, XML, intersection types, type constraints.

1. Introduction

Many XML processing languages, such as XDuce, C-Duce, XQuery, OcamDIuse, 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— was lifted by Castagna and Xu [4]. In Part 1 of this work, presented in the previous edition of POPL [3], we described how to take full advantage of Castagna and Xu’s system by defining a calculus with higher-order polymorphic functions and recursive types with union, intersection, and negation connectives. The approach is general and goes well beyond the sole application to XML processing languages. As a matter of fact, the motivating example we gave in Part 1 [3] does not involve XML, but looks like a rather classic display of functional programming specimens:

\[
\begin{align*}
\text{map} & : (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
\text{map f l} & = \text{case l of} \\
& \quad | [] \rightarrow [] \\
& \quad | (x : xs) \rightarrow (f \ x : \text{map f xs})
\end{align*}
\]

\[
\begin{align*}
\text{even} & : (\text{Int} \rightarrow \text{Bool}) \land ((\gamma \rightarrow \text{Int}) \rightarrow (\alpha \rightarrow \text{Int})) \\
\text{even x} & = \text{case x of} \\
& \quad | \text{Int} \rightarrow (x \mod 2 == 0) \\
& \quad | _ \rightarrow x
\end{align*}
\]

The first function is the classic map function defined in Haskell (we use Greek letters to denote type variables). The second would be an Haskell function were it not for two oddities: its type declaration contains type connectives (type intersection “\&“ and type difference “\setminus“); 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 the definition of even may seem weird, it follows a very common pattern used to manipulate functional data-structures. Two examples are Okasaki’s functional implementation of red-black trees (for which our system provides a far better typing) and the transformation of XML documents whose elements are modified or left unchanged according to their tag-type (see actual code in Section 3.3 later on and in Appendix A). Furthermore it is a perfect minimal example to illustrate all the aspects of our system.

In Part 1 [3] we showed that the system presented there is expressive enough to define the two functions above and to verify that they 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. However, the system in Part 1 [3] is not able to infer (without the help of the programmer) the type of the partial application of map to even, which must be equivalent to

\[
\begin{align*}
\text{map even} & : ([\text{Int}] \rightarrow [\text{Bool}]) \land \\
& ((\gamma \rightarrow \text{Int}) \rightarrow (\alpha \rightarrow \text{Int})) \land \\
& ((\gamma \land \text{Int}) \rightarrow (\gamma \land \text{Int}))
\end{align*}
\]

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).

Typing map even is difficult because it demands to infer several different instantiations\(^1\) of the type of map and then take their intersection. This is why the calculus in [3] includes explicit type substitutions: the programmer must explicitly provide the type-substitutions used to instantiate the types of the terms that form an application, a requirement that makes the system of [3] not usable in practice, yet. In this paper we remove this limitation by defining a sound and complete inference system that deduces the type-substitutions that a programmer should insert in a program of [3] to make it well typed. In other words, we define “local

\(^1\)For map even we need to infer just two instantiations, namely, \((\gamma \land \text{Int}) / \alpha, (\gamma \land \text{Int}) / \beta\) and \((\gamma \land \text{Int}) / \alpha, (\gamma \land \text{Int}) \lor \text{Bool} / \beta\). The type in (1) is redundant since the first type of the intersection is an instance (eg, for \(\gamma = \text{Int}\)) of the third. We included it just for the sake of the presentation.
type inference for [3], namely, we solve the problem of checking whether there exist some type-substitutions that make the types of a function and of its arguments compatible and, if so, of inferring the type of the application as we did for (i). In particular, we show that local type inference for [3] reduces to the problem of finding two sets of type substitutions \( \{ \sigma_i \mid i \in I \} \) and \( \{ \tau_j \mid j \in J \} \) such that for two given types \( t \) and \( s \) the relation \( \bigwedge_{i \in I} s \sigma_i \leq \bigwedge_{j \in J} \tau_j \) holds, and we give a sound and complete algorithm for this problem. We also show how the same algorithm can be used to perform type reconstruction and infer types more precisely than those inferred by the type systems of the ML family. All detailed proofs and complete definitions can be found in the Appendix.

The system is fully implemented and, at the moment of writing, in alpha-test. It will be distributed in the next public release of the CDuce language [2]. In the meanwhile, the current version can be tested by compiling the master branch of the CDuce git repository: git clone https://git.cduce.org/cduce (we recommend to check the bugtracker for current issues).

Next section outlines the various problems to be faced in this research and succinctly describes the system of [3]. The reader acquainted with the work in [3] can skip directly to Section 2.1.

2. Overview

The aim of this research is the definition an XML processing functional language with high-order polymorphic functions, that is, in the specific, a polymorphic version of the language CDuce [2]. CDuce is a strongly-typed programming language that eases the manipulation of data in XML format. Issued from academic research it is used in production, available on different platforms, and included in all major Linux distributions. 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 basic, arrow, and product type constructors and the intersection, union, and negation type connectives. In this work we omit for brevity recursive functions and product types constructors and expressions (our results can be easily extended to them as sketched in Section 5 and detailed in the appendixes) and add type variables. So in the rest of this work we study a calculus whose types and expressions are described by the next two following definitions.

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

\[
t ::= b \mid t \to t \mid t \land t \mid t \lor t \mid \neg t \mid 0 \mid 1 \mid \alpha
\]

and such that every infinite branch contains infinitely many occurrences of \( \to \) or \( \land \) constructor. We use \( T \) to denote the set of all types.

In the definition, \( b \) ranges over basic types (eg. \( \text{Int} \), \( \text{Bool} \)), \( \alpha \) ranges over type variables, and \( 0 \) and \( 1 \) respectively denote the empty (that types no value) and top (that types all values) types. Coinduction accounts for recursive types and the condition on infinite branches bars out ill-formed types such as \( t \equiv t \lor t \) (which does not carry any information about the set denoted by the type) or \( t \equiv \neg t \) (which cannot represent any set). It also ensures that the binary relation \( \triangleright \subseteq T^2 \) defined by \( t_1 \triangleright t_2 \triangleright t_3 \) \( \Leftrightarrow t_1 \land t_2 \triangleright t_3 \land \neg t : t_1 \triangleright t_2 \triangleright t_3 \) is Noetherian. This gives an induction principle on \( T \) that we will use without any further explicit reference to the relation. We use \( \var(t) \) to denote the set of type variables occurring in a type \( t \).

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 acceptance used in Scala where, like in C# and Java, type parameters for polymorphic/generic method calls can be omitted. In our specific problem, we will omit —and, thus, infer— the annotations that specify how the types of a function and of its argument can be made compatible. As explained in Section 6 it is more general than Pierce and Turner’s local type inference for arguments types [20].

A type \( t \) is said to be ground or closed if and only if \( \var(t) \) is empty. The subtyping relation for these types is the one defined by Castagna and Xu [4]. For this work it suffices to consider that ground types are interpreted as sets of values (i.e., either constants or λ-abstractions) that have that type, and that 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, \( s \rightarrow t \) contains all λ-abstractions that when applied to a value of type \( s \), if the computation terminates, then they return a result of type \( t \) (eg. \( 0 \rightarrow 1 \) is the set of all functions\(^3\) and \( 1 \rightarrow 0 \) is the set of functions that diverge on every argument). Type connectives (ie, union, intersection, negation) are interpreted as the corresponding set-theoretic operators (eg, \( s \lor t \) is the union of the values of the two types). 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 \( \sigma \) (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 \( \equiv \)). Finally, notice that in this system \( s \leq t \) if and only if \( s \land \neg t \leq 0 \).

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

\[
e ::= c \mid x \mid e \mid \lambda x.e \mid e \lor e \mid e \land e
\]

and such that in every expression \( e \equiv ? \) \( e_1 : e_2 \) the type \( t \) is closed.

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

As we said at the beginning of the section, the functional core of CDuce [2] has exactly the same types and expressions as the above except for two single differences: (i) its types do not contain type variables and (ii) it includes product types and recursive functions, which we omitted here for brevity. The reasons why in CDuce (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 to which the reader can refer. The novelty of this research with respect to CDuce, 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 \( \lambda x.\to x.x \), while the classic “auto-application” term is written as \( \lambda (\alpha \to \beta) : \alpha \to \beta \rightarrow x.x.x \). The intended meaning of using a type variable, such as \( \alpha \), 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 \( \lambda x.\to x.x \) has not only types \( \alpha \rightarrow \alpha \) but by subsumption also, for instance, the types \( \alpha \rightarrow \beta \) (the type of all functions, which is a super-type of \( \alpha \rightarrow \alpha \)) and \( \text{Int} \) (the type of all non integer values), and by instantiation the types \( \text{Int} \rightarrow \text{Int} \), \( \text{Bool} \rightarrow \text{Bool} \), etc. The addition of type variables and instantiation makes the calculus a full-fledged intersection
type system (see Section 3.5 in [3]): for instance, by combining intersections, instantiation, and subtyping, it is possible to deduce that \(\lambda^{\alpha\rightarrow\beta}.x.x\) has type \((\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool}) \land \lnot \text{Int}\).

The key problem to be solved, then, is to define an explicit-
typed \(\lambda\)-calcus with intersection types and a type-case expression. This is technically quite challenging because of three main reasons: (i) type instantiation must be explicit, (ii) it may require the use of sets of type-substitutions, and (iii) it cannot always be immediately propagated to the body of a function. A detailed description of these reasons can be found in [3] but, in a nutshell:

(i) instantiation must be explicit because of the presence of a type-case: we check the type of a function by checking its type annotation, thus any type-substitution of variables of type annotation must be explicitly propagated. That is, to apply \(\lambda^{\alpha\rightarrow\beta}.x.x\) to 42 we must first apply the type-substitution \([\text{Int}^{\alpha\rightarrow\beta}]\) to it, yielding \(\lambda^{\text{Int} \rightarrow \text{Int}}.x.x\), and only then we can apply the function to 42.

(ii) sets of type-substitutions are needed because of intersection types. A function that expects arguments of type must be previously instantiated by a (we must first apply the type-substitution reasons can be found in [3] but, in a nutshell: This is technically quite challenging because of three main reasons: for instance, by combining the daffy identity function to an integer we must instantiate its type system (see Section 3.5 in [3]): for instance, by combining

\[ (\lambda^{\alpha\rightarrow\beta}.x.x) \frac{x}{42} = \gamma, \]

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 \(\lambda\)-abstraction \(\lambda^{\alpha\rightarrow\beta}.x.x.e\) we call the type \(\lambda^{\alpha\rightarrow\beta}.x.x.e\) the interface of the function and the set of type-substitutions \([\sigma]_j\) \(\in J\) the decoration of the function. We write \(\lambda^{\alpha\rightarrow\beta}.x.x.e\) 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 \(\lambda\)-abstractions. Let \(e\) be an expression: we use \(v(e)\) and \(b(v(e))\) respectively to denote the sets of free expression variables and bound expression variables of the expression \(e\); we use \(v(e)\) 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 \(\lambda^{\alpha\rightarrow\beta}.e\) (\(\lambda^{\alpha\rightarrow\beta}.y.y\)), \(\alpha\) occurring in the interface of the inner abstraction is “bound” by the decoration \([\sigma]_j\), and the whole expression is \(\alpha\)-equivalent to \(\lambda^{\beta\rightarrow\gamma} y.(\alpha^{\gamma\rightarrow\gamma} y.y)\). If a type variable is bound by an outer abstraction, it cannot be instantiated; such a type variable is called monomorphic. We assume that polymorphic type variables are pairwise distinct and distinct from any monomorphic type 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 [3] in terms of a relabeling operation “\(\alpha\rightarrow\beta\)”, which takes an expression \(e\) and a set of type-substitutions \([\sigma]_j\) \(\in J\) and pushes \([\sigma]_j\) down to all outermost \(\lambda\)-abstractions occurring in \(e\) (and collects and composes with the sets of type-substitutions it meets). Precisely, \(\alpha \in [\sigma]_j\) \(\in J\) is defined for \(\lambda\)-abstractions and applications of type-substitutions as

\[ (\lambda^{\alpha\rightarrow\beta}.x.x.e) (\alpha^{\gamma\rightarrow\gamma} y.y) \alpha \in [\sigma]_j \in J \iff \lambda^{\alpha\rightarrow\beta}.x.x.e \in [\sigma]_j \in J \]

Thus obtaining a “lazy” instantiation of function bodies, which delays the propagation of a set of substitutions to the function body until the precise type of the function argument is known. This is obtained by decorating \(\lambda\)-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 \((\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool})\) we first “ lazily” instantiate it as follows:

\[ (\lambda^{\alpha\rightarrow\beta}.x.x\,\text{Bool}) (\alpha^{\gamma\rightarrow\gamma} y.y) x. \]
belonging depends on the type of the argument and keeps only those type-substitutions that make the type of the argument 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_iσ_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, \((λ^\text{Int}_{i=0} λ^\text{Int}_{j=0}.x.\text{Int}→\text{Int})(x→\text{Int}\times\text{Int})(x)\) 42—, then it reduces to \((λ^\text{Int}_{i=0} λ^\text{Int}_{j=0}.\text{Int}→\text{Int})(y.42)42\), (which is observationally equivalent to \((λ^\text{Int}_{i=0} λ^\text{Int}_{j=0}.\text{Int}→\text{Int})(y.42)42\) since the reduction discards the \{\text{Bool}_{i}\}\) 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 the \(\text{α}\) subscript in \(\text{Γ}⇒\) and by 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 [3] 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 interesting for this second part, that is, (ALG-ABSTR), (ALG-INST), and (ALG-APPL). First of all notice the presence of 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; this set must contain all the type variables occurring in \(\text{Γ}\). Rule (ALG-ABSTR) checks that \(\text{Γ}⇒\text{e}:\text{t}\). Rule (ALG-INST) for applications checks that the type of the function is a functional type (ie, the combination of) its interface and its decoration, that is, \(\text{Γ}⇒\text{σ}[\text{e}]:\text{t}\). 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 \(σ_i\)—the function body \(e\) relabeled with the single type-substitution \(σ_j\) under consideration (ie, \(e[\text{σ}_j]\)), has (a subtype of) the corresponding output type.

Rule (ALG-INST) for e[σ_j]∈I.J the type obtained by applying the type of the set-substitutions to the type of e, provided that the type-substitutions do not instantiate monomorphic variables (ie, for all j∈I. dom(σ_j)∩Δ=∅, noted as σ_j∉Δ).

Rule (ALG-APPL) for applications checks that the type of the function is a functional type (ie, \(t\leq0\rightarrow\text{t}\)). Then it checks that the type of the argument is a subtype of the domain of \(t\) (denoted by \(\text{dom}(t)\)). Finally, it infers for the application the type \(t\geq0\equiv\min\{u\mid t\leq0\rightarrow u\}\), that is, the smallest result type that can be obtained by subsuming \(t\) to an arrow type with domain \(s\).

Even if \(t\leq0\rightarrow\text{t}\), 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 \(\text{dom}(t)\) is not immediate. Formally, if \(t\leq0\rightarrow\text{t}\), then \(t\geq0\equiv\min\{u\mid t\leq0\rightarrow u\}\), that is, the smallest result type that can be obtained by subsuming \(t\) to an arrow type with domain \(s\).

\(^4\)For every type \(t\) such that \(t\leq0\rightarrow\text{t}\) and type \(s\) such that \(s\leq\text{dom}(t)\), the type \(t\rightarrow s\) exists and can be effectively computed.

\[\begin{align*}
\text{(ALG-CONST)} & \quad \Delta;\Gamma⇒\text{e}:\text{t} \\
\text{(ALG-VAR)} & \quad \Delta;\Gamma⇒\text{x}:\text{Γ}(\text{x}) \\
\text{(ALG-ABSTR)} & \quad \Delta;\Delta∪\Delta;Γ⇒\text{e}:\text{t} \\
\text{(ALG-INST)} & \quad \Delta;\Gamma⇒\text{σ}[\text{e}]:\text{t}\quad\text{ψ} \\
\text{(ALG-APPL)} & \quad \Delta;\Gamma⇒\text{e}:\text{t} \\
\text{(ALG-INST)} & \quad \Delta;\Gamma⇒\text{e}[\text{σ}]:\text{t}\quad\text{ψ} \\
\text{(ALG-APPL)} & \quad \Delta;\Gamma⇒\text{e}:\text{t} \\
\text{(ALG-FST)} & \quad \Delta;\Gamma⇒\text{e}:\text{t} \\
\text{(ALG-CASE-both)} & \quad \Delta;\Gamma⇒\text{e}:\text{t} \\
\end{align*}\]
that in the implicitly-typed calculus (3) even can be defined as
\[
\lambda [\text{Int} \rightarrow \text{Bool}] \bullet ([\text{Int} \rightarrow \alpha] \bullet x) \; x \in \text{Int} \; ? \; f \in \text{Int} \; x \mod 2 = 0 : x \quad (11)
\]
(\(s\))\(t\) is syntactic sugar for \(s \land \neg t\) while — with the products and recursive function definitions given in the appendix — map is
\[
\mu m_{\langle \alpha \rightarrow \beta \rangle} \mu m_{\langle \alpha \rightarrow \beta \rangle} f = \lambda [\text{Alpha}] \bullet \text{let}\; \text{nil} = \text{NIL} \; \text{in} \; \text{nil} : (f(\pi_1), m f(\pi_2)) \quad (12)
\]
where the type \text{nil} tested in the type case denotes the singleton type that contains just the constant \text{nil}, and \([\alpha]\) denotes the regular type that is the least solution of \(X = (\alpha) \times \text{nil}\).

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) \text{map} to (the expression (11) defining) \text{even} we need to instantiate \text{map}, that is, to perform local type inference. The type inference system of Section 3 infers the following set of type-substitutions \([\langle (\gamma) \text{Int} \rangle / \alpha, \langle (\gamma) \text{Int} \rangle / \beta, \langle (\gamma) \text{Int} \rangle / \beta \rangle\) and textually inserts it between the two terms (so that the type-substitutions apply to the type variables of \text{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 Part 1 [3] and executed as efficiently as if it were a monomorphic expression. Finally, in Section 4 we show that we could allow the programmer to omit the type declaration for \text{map} —ie, \text{map} : \langle \alpha \rightarrow \beta \rangle \rightarrow \langle \alpha \rangle \rightarrow \langle \beta \rangle— since it is possible to reuse the algorithms developed in Section 3 to reconstruct for \text{map} a type slightly more precise than the one above.

Contributions: The overall contribution of this work (Parts 1 and 2) 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 local type inference. The technical contributions of this Part 2 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 type-substitution \(\sigma\) 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 system of the explicitly-typed calculus of [3].

– the definition of a sound and complete algorithm for local type inference for the calculus. The algorithm yields a semi-decision procedure for the typeability of a \(\lambda\)-calculus with intersection and recursive types and with explicitly-typed \(\lambda\)-abstractions.

– the definition of a type reconstruction algorithm that uses the machinery developed for local type inference and improves reconstruction defined for ML languages.

We also provide two different implementations: a prototype implementation of the calculus presented here and the polymorphic extension of the compiler of CDuce, a production-grade language.

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 the grammar in (7). To define the type-substitution inference system we proceed in two steps. First, we define a syntax-directed deduction system for the implicitly-typed calculus by modifying the one in Figure 1: whenever the old system checks a subtyping relation, the new system tries to guess some explicit type-substitutions to insert in that position. Second, we show how to compute the operations used by the deduction system defined in the first step. 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 a single exception: we will not try to insert type-substitutions in decorations, that is, we will consider only expressions in the explicitly-typed calculus in which all decorations are absent (ie, they are a singleton set that contains only the empty type-substitution). 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 \(\lambda^{\neg \neg\neg} x.3\) we want to infer that it is ill-typed (as, say, Haskell does); but if we allowed to infer decorations, then the expression could be typed by inserting a decoration as in \(\lambda [\text{Int}](\gamma) \rightarrow \beta\). Likewise, if the programmer specified the signature \text{map} : \langle (\alpha \rightarrow \beta) \rangle \rightarrow \beta\), we expect the system to answer that the definition of \text{map} does not conform this signature, rather than it conforms the signature by substituting \(\langle (\alpha \rightarrow \beta) \rangle \rightarrow \beta\) for \(\gamma\) (alternatively, we must omit the signature altogether and let the system infer it: see Section 4 on reconstruction).

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 an explicitly-typed expression that is well typed 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 that occur 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 \(i.e., those not in \(\Delta\)\) 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 seeing that whenever a subtyping relation is specified, the right-hand side type cannot be instantiated: either because it is a ground type (rules (ALG-CASE-8)) 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 later in this section.

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, \(\Delta\) a set of type variables, and \(\{\sigma_i\}_{i \in I}\) a set of type-substitutions. We define:

\[
\sigma_i \in I \; \vdash\; s \sqsubseteq \Delta \; t \quad \iff \quad \exists \sigma_i \in I \; s \sqsubseteq \Delta, \forall i \in I, \sigma_i \notin \Delta \quad (13)
\]

\[
s \sqsubseteq \Delta \; t \quad \iff \quad \exists \{\sigma_i\}_{i \in I} \; \text{such that } [\sigma_i]_{i \in I} \; \vdash\; s \sqsubseteq \Delta \quad (14)
\]

Intuitively, it suffices to replace \(s\) by \(t\) for \(\sqsubseteq\Delta\) and \(\sqsubset\Delta\) in the algorithmic rules of Figure 1 (where \(\Delta\) 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 (we subscribed the turnstile symbol by \(\vdash\) to stress that it is the \(\vdash\) inference system for type-substitutions) 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 \( \lambda \)-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 (\( s_i' \subseteq \Delta \cup \Delta' \). Notice that we do not require the sets of type-substitutions that make \( s_i' \subseteq \Delta \cup \Delta' \) satisfy 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 (i.e., to intersecting them point-wise: see Definition B.9 and Corollary B.12 — henceforth, references starting with letters refer to appendices).

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 and the other for the argument type) but also because the set of type-substitutions for the function type must enforce two distinct constraints: the type resulting from applying the set of type-substitutions to the type of the function must be a subtype of \( s \to t \), and its domain must be compatible with \( (i.e., a super-type of\) the type inferred for the argument. In order to solve all these constraints we collapse them into a single definition which is the algorithmic counterpart of the set of types used in Section 2 to define the operation \( t \cdot s \) occurring in the rule (ALG-APPL). Precisely, we define \( t \bullet s \) as the set of types for which there exist two sets of type-substitutions (for variables not in \( \Delta \)) that make \( s \) compatible with the domain of \( t \):

\[
\begin{align*}
  t \bullet s \overset{def}{=} & \left\{ u \mid \begin{array}{c}
  \forall \sigma \in C \quad (\sigma \vdash_{\Delta} 0 \to u) \\
  \forall \sigma \in C \quad (\sigma \vdash_{\Delta} s \subseteq \text{dom}(t[\sigma])_{(j, i)}) \\
  u = t[\sigma]_{(j, i)} \cdot s[\sigma]_{(j, i)}
  \end{array} \right\}
\end{align*}
\]

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 B.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 \( t \bullet s \). 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 \( \lambda \)-abstractions are empty. Both of these properties are stated in terms of the \textbf{erase(\cdot)} 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 \( \Delta \Gamma \vdash a : t \), then there exists an explicitly-typed expression \( e \) such that \( \text{erase}(e) = a \) and \( \Delta \Gamma \vdash e : t \).

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 we defined in Part I [3] and evaluated. Notice that \( \subseteq_{\Delta} \) 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 fact, adding to the system in Figure 2 a subsumption rule that uses the relation \( \subseteq_{\Delta} \) that:

\[
\text{(SUBSUMPTION)} \quad \Delta \Gamma \vdash a : t_1 \quad t_1 \subseteq_{\Delta} t_2 \\
\Delta \Gamma \vdash a : t_2
\]

is sound. This means that the set of solutions is upward closed with respect to \( \subseteq_{\Delta} \) 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 \( \Delta \Gamma \vdash e : t \), then there exists a type \( t' \) such that \( \Delta \Gamma \vdash e : t' \) and \( t' \subseteq_{\Delta} t \).

The inference system is syntax directed and describes an algorithm that is parametric in the decision procedures for \( \subseteq_{\Delta} \) 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 \( \Delta \) a finite set of type variables. A type-substitution \( \sigma \) is a solution for the tallying problem of \( C \) and \( \Delta \) (noted \( \sigma \vdash_{\Delta} C \)) if \( \sigma \not\vdash_{\Delta} C \) and for all \( (s, t) \in C \), \( s \not\subseteq t \) holds.

Thus a constraint-set corresponds to the logical conjunction of the constraints that compose it, and the tallying problem searches for a type-substitution that satisfies this conjunction. 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 \( \vdash_{\Delta} \) 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 \( s \not\subseteq_{\Delta} t \) for given \( s \) and \( t \). By definition this corresponds to finding a set of \( n \) type-substitutions \( [\sigma_i]_{i \leq n} \) such that \( \bigwedge_{i \leq n} (s \not\subseteq t) \). We can split each type-substitution \( \sigma_i \) in two: a renaming type-substitution \( \rho_i \) that maps each variable of \( s \) not in \( \Delta \) into a fresh type variable, and a type substitution \( \sigma'_i \) such that \( \sigma_i = \sigma'_i \circ \rho_i \). Thus the inequality becomes \( \bigwedge_{i \leq n} (s \rho_i)\sigma'_i \leq t \). The domains of \( \sigma'_i \) are
by construction pairwise disjoint (they are formed of distinct fresh variables) and disjoint from the variables in $t$; thus we can merge them into a single substitution $\sigma = \bigcup_{i \in I} \sigma_i$ and apply it to $t$ with no effect, yielding the inequality $(\bigwedge_{i \in I} \sigma_i) \sigma \leq \sigma$. Let $u_n = \bigwedge_{\sigma \leq \sigma_i} \sigma_i$, we have just transformed the problem of proving the relation $s \subseteq \Delta t$ into the problem of finding an $n$ for which there exists a solution to the tallying problem for $\{u_n \leq t\}$ and $\Delta$. The way to proceed to find $n$ is explained in Section 3.2.3.

The (INF-APPL) rule deserves a special treatment 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 $\Delta$ such that both $\bigwedge_{i \in I} \sigma_i \leq 0 \rightarrow t$ and $\bigwedge_{j \in J} \sigma_j \leq \text{dom}(\bigwedge_{i \in I} \sigma_i)$ 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 $\subseteq$—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 disjoint 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 of whose form is either

$$\bigwedge_{p \in P} b_p \land \bigwedge_{n \in N} \neg b_n \land \bigwedge_{q \in P'} \neg q_n \land \bigwedge_{r \in N'} \neg q_r \quad (13)$$

or

$$\bigwedge_{p \in P} (s_p \rightarrow t_p) \land \bigwedge_{n \in N} (\neg s_n \land \neg t_n) \land \bigwedge_{q \in P'} \neg q_n \land \bigwedge_{r \in N'} \neg q_r \quad (14)$$

When either $P'$ or $N'$ is non empty, we call the variables $a_q$'s and $a_r$'s the top-level variables of the normal form.

### 3.2.3 Solution of the tallying problem.

In order to solve the tallying problem for given $\Delta$ and $C$, we first fix some total order $\preceq$—any will do—on the type variables occurring in $C$ and not in $\Delta$ (from now on, when speaking of type variables we will mean type variables not in $\Delta$), order that will be used to ensure that all inferred types satisfy the contracitivity condition of Definition 2.1. Next, we produce sets of constraint-sets (as a single constraint-set corresponds to logical conjunction, so a set of constraint-sets corresponds to disjunction of the corresponding conjunctions) 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 by producing soluble sets of equations equivalent to the original problem and then solving these equations; 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 $\mathcal{H}_1, \mathcal{H}_2 \subseteq \mathcal{P}(\mathcal{T} \times \mathcal{T})$ be two sets of constraint-sets. We define

$$\mathcal{H}_1 \cap \mathcal{H}_2 \equiv \{ C_1 \cup C_2 | C_1 \in \mathcal{H}_1, C_2 \in \mathcal{H}_2 \}$$

$$\mathcal{H}_1 \cup \mathcal{H}_2 \equiv \mathcal{H}_1 \cup \mathcal{H}_2$$

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, $t = \bigwedge_{p \in P} t_p \land \bigwedge_{n \in N} \neg t_n \land \bigwedge_{q \in P'} t_q \land \bigwedge_{r \in N'} \neg t_r$ and $k \in P' \cup N'$, we define single$(\alpha_k, t)$ as the constraint equivalent to $t \leq 0$ in which $\alpha_k$ is “singled-out”, that is,

$$\bigwedge_{p \in P} t_p \land \bigwedge_{n \in N} \neg t_n \land \bigwedge_{q \in P'} t_q \land \bigwedge_{r \in N'} \neg t_r \leq \alpha_k$$

When $k \in N'$ and $\alpha_k \leq \bigvee_{p \in P'} t_p \lor \bigvee_{n \in N} \neg t_n \lor \bigvee_{q \in P'} t_q \lor \bigvee_{r \in N'} \neg t_r$ and $\alpha_k \leq \bigvee_{n \in N} \neg t_n \lor \bigvee_{q \in P'} t_q \lor \bigvee_{r \in N'} \neg t_r$ when $k \in P'$. Henceforth, to enhance readability we will often write $s \leq t$ for the constraint $(s, t)$, as we did above.

**Example.** We will show the various phases of the process by solving the tallying problem for the following constraint-set:

$$C = \{(\alpha \rightarrow \text{Bool}, \beta \rightarrow \beta), (\text{Int} \lor \text{Bool} \rightarrow \text{Int}, \alpha \rightarrow \beta)\}$$

and assume that $\alpha \preceq \beta$.

#### 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 $t \leq 0$. This function is parametric in a set $M$ of visited types (needed to handle coinduction) 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 of a normal form to 0 (beginning of line 6). If a summand contains a top-level variable, then the smallest (wrt $\preceq$) 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

5 Equivalence of $\leq 0$ and the two following constraints is easily derived from the De Morgan’s laws and the property $t_1 \leq t_2 \iff t_1 \land \neg t_2 \leq 0$. 

---

Figure 3. Constraint normalization
the constraint by calling the subtyping algorithm and, accordingly, it returns either the unsatisfiable set of constraint-sets (false) or the one that is always satisfied (true) (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 \( M \) of visited types. The regularity of types ensures that the algorithm always terminates (see Lemma C.14). Notice that, in line 2 the algorithm always singles out the smallest variable. Therefore, by construction, if \( \text{norm} \) generates a constraint \( (\alpha,t) \) or \( (t,\alpha) \), then every variable smaller than or equal to \( \alpha \) may occur in \( t \) only under an arrow (equivalently, every top-level variable of \( t \) is strictly larger than \( \alpha \)).

**Remark 3.1.** There is the special case of \( (\alpha,t) \) or \( (t,\alpha) \) 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 merge.

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

**Example (Cont’d).** The function \( \text{norm} \) works on single constraints (actually, on a type \( t \) representing the constraint \( t \leq 0 \)), so let us apply it on the first constraint of the example. We want to normalize the constraint \( \alpha \to \text{Bool} \leq \beta \to \beta \) and thus we apply **norm** to the type \( (\alpha \to \text{Bool} \land \neg (\beta \to \beta)) \). Now, this constraint has two distinct solutions: either (i) \( \beta \) is the empty set, in which case the larger type becomes \( \text{true} \to \text{true} \) that is the type of all functions (see Footnote 3) which contains every arrow type, in particular \( \alpha \to \text{Bool} \), or (ii) the types satisfy the usual covariant-contravariant rule for arrows, that is, \( \beta \leq \alpha \) and \( \text{Bool} \leq \beta \). Since there are two distinct solutions, then **norm** generates a set of two constraint-sets. Precisely \( \text{norm}((\alpha \to \text{Bool} \land \neg (\beta \to \beta)), \emptyset) \) returns \( \{\emptyset\} \). \( \{\beta, \emptyset\}, \{\alpha, \text{Bool}, \beta\} \} \). 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 \( P' = \text{false} \) since it ranges over the strict subsets of \( P \) which, in this case, is a singleton—so it requires \( s_0 \), \( \beta \) to be empty), while the second constraint-set is obtained by the union of the first two recursive calls (which require \( s_0 \leq s_p \) 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 where the variable is on the left-hand side and the other where it is on the right-hand side. 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 \( \alpha, i \leq \alpha, i < t, i \in I \) where the \( \alpha, i \)'s are pairwise distinct. This is done by the function \( \text{merge}(C, M) \) where \( C \) is a normalized constraint-set and \( M \) a set containing the types already visited by the function.

\[
\text{merge}(C, M) =
\]

\[
\begin{cases}
\text{Rewrite } C \text{ by applying as long as possible the following rules according to the order } \leq \text{ on the variables (smallest first)}:\ 
- \text{if } (\alpha, t_1) \text{ and } (\alpha, t_2) \text{ are in } C, \text{ then replace them by } (\alpha, t_1 \land t_2); \\
- \text{if } (s_1, \alpha) \text{ and } (s_2, \alpha) \text{ are in } C, \text{ then replace them by } (s_1 \lor s_2, \alpha); \\
\end{cases}
\]

\[
\text{if there exist two constraints } (s, \alpha) \text{ and } (\alpha, t) \text{ in } C \text{ s.t. } s \land \alpha \not\subseteq M, \\
\text{then let } D = \{C\} \cap \text{norm}(s \land \alpha, \emptyset) \\
\text{in return } \bigcup_{\alpha \leq \beta \leq \alpha} \text{merge}(C', M \cup \{s \land \alpha\}) \\
\text{else return } C
\]

The function \( \text{merge} \) performs two steps. In the first step it scans (using \( \leq \) 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 taking the union of all its lower bounds and the intersection of all its upper bounds. For instance, if \( C \) contains the following five constraints for \( \alpha \): \( (s_1, \alpha), (s_2, \alpha), (\alpha, t_1), (\alpha, t_2), (\alpha, t_3) \), then the first step replaces them by \( (s_1 \lor s_2, \alpha) \) and \( (\alpha, t_1 \land t_2, \alpha) \), which corresponds to having the constraint \( s_1 \lor s_2 \leq \alpha \leq t_1 \land t_2 \). Such a constraint is satisfiable only if the constraint that the lower bound of \( \alpha \) is smaller than its upper bound is satisfiable. This is checked in the second step, which looks for pairs of constraints of the form \( (s, \alpha) \) and \( (\alpha, t) \) (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 \( (s \land \alpha, t) \) but neither it or \( (s, t) \) is normalized. Thus before adding it to \( C \) we normalize it by calling \( \text{norm}(s \land \alpha, t) \).

Remark that \( \text{norm} \) 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 \( \text{norm}(s \land \alpha, t) \) via the \( \cap \) 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 \( \cap \) and we record \( s \land \alpha \) 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 \( s \land \alpha \) is not already in \( M \). Note that merge preserves the property that in every constraint \((\alpha, t)\) or \((t,\alpha)\), every variable smaller than or equal to \( \alpha \) may occur in \( t \) only under an arrow.

**Example (Cont’d).** If we apply \( \text{norm} \) also to the second constraint of our example we obtain a second set of constraint-sets: \( \{\emptyset\}, \{(\alpha, 0), (\beta, 0)\}, \{(\alpha, 1\land \text{Bool}), (\beta, 1), (\beta, 0)\}, \{(\beta, 0), (\alpha, 0)\}, \{(\text{Bool}, \beta), (\beta, 0)\}, \{(\beta, 0), (\alpha, 0)\}, \{(\alpha, 1\land \text{Bool})\} \} \}

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 \( \text{Bool} \leq 0 \). The same happens for the third since it first adds \( \beta \leq 0 \) and at the recursive call tries to solve \( \text{Bool} \leq 0 \). The fourth one is more interesting: in step 1 it replaces \( (\text{Bool}, \beta) \) and \( (\alpha, 0) \) (see Footnote 3) which contains every arrow type, in particular \( (\text{Bool}, \beta) \) and \( (\alpha, 0) \). Merge on the second step it tries to solve \( (\text{Bool}, \beta) \) and \( (\alpha, 0) \) via the \( \cap \) 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 \( \cap \) and we record \( s \land \alpha \) 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 \( s \land \alpha \) is not already in \( M \). Note that merge preserves the property that in every constraint \((\alpha, t)\) or \((t,\alpha)\), every variable smaller than or equal to \( \alpha \) may occur in \( t \) only under an arrow.

### 3. Constraint solving

**norm** and merge yield a set in which every constraint-set is of the form \( \{s_1 \leq \alpha, i < t, i \in I\} \) where \( \alpha, i \) are pairwise distinct variables and \( s_1 \) and \( t \) are respectively \( 0 \) or \( \text{false} \) 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 \( \alpha \prec \beta \), then \( \{\alpha, \beta\} \) will be considered to represent \( \{0 \leq \alpha, \beta \leq 0, \beta < 0, \beta < \alpha \leq \text{false}\} \). Thanks to this assumption the system so obtained is well ordered, that is, for every constraint \( s \leq \alpha \land \text{false} \) in it, the top-level variables of \( s \) and \( t \) are strictly larger than \( \alpha \). Notice that in doing that we do not lose any information: the bounds for larger variables are still recorded in those of smaller ones and any bound for larger variables obtained by transitivity on the smaller variables 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 $\text{solve}(C)$ as follows:

$$\text{solve}(C) = \{s = (s \lor \beta) \land t \mid (s \leq \alpha \leq t) \in C, \beta \text{ fresh}\}$$

The function $\text{solve}(C)$ takes every constraint $s \leq \alpha \leq t$ in $C$ and replaces it by $\alpha = (s \lor \beta) \land t$ (with $\beta$ fresh). It is clear that the constraint-set $C$ has a solution for every possible (unconstrained) assignment of $\alpha$ included between $s$ and $t$ if and only if the new constraint-set has a solution for every possible (unconstrained) assignment of $\beta$. By Courcelle [7] (and Lemma C.44) there exists a solution of this set, namely, a substitution from the type variables $\alpha$ variable. By Courcelle [7] (and Lemma C.44) there exists a solution $\alpha$.

Lemma C.44 states that the set of equations is well ordered. This last property ensures the contractivity of the equation defining the smallest type.

Theorem 3.6 (Soundness and completeness). The algorithm to solve the tallying problem for $C$ and variables not in $\Delta$, then, proceeds in three steps:

Step 1. Let $\mathcal{M} = \prod_{(s,t) \in \mathcal{C}} \text{norm}(s \land t, \emptyset)$, if $\mathcal{M} = \emptyset$ then fail else proceed to the next step.

Step 2. Let $\mathcal{N} = \bigcup_{C \in \mathcal{M}} \text{merge}(C, \emptyset)$, if $\mathcal{N} = \emptyset$ then fail else proceed to the next step.

Step 3. Let $\mathcal{S} = \bigcup_{C \in \mathcal{N}} \text{solve}(C)$. Return $\{ \text{Unify}(E) \mid E \in \mathcal{S} \}$.

Let $\text{Sol}_\Delta(C)$ 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).

$$\sigma \in \text{Sol}_\Delta(C) \Rightarrow \sigma \sqsubseteq_\Delta C$$

$$\sigma \sqsubseteq_\Delta C \Rightarrow \exists \sigma' \in \text{Sol}_\Delta(C), \sigma'' , s.t. \sigma \approx \sigma'' \circ \sigma'$$

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 C).

**Example (End).** After Step 1 and 2 our initial tallying problem $\{(c \rightarrow \text{Bool}, \beta \rightarrow \beta), (\text{Int} \rightarrow \text{Bool} \rightarrow \text{Int}, \alpha \rightarrow \beta)\}$ has become

$$\{(c \rightarrow \text{Bool}, (\beta, \emptyset)) \cup \{(\beta, \alpha), (\alpha, \text{Int} \rightarrow \text{Bool}), (\beta, \text{Int} \rightarrow \text{Bool})\} \}.$$ Let us apply Step 3. The first constraint-set is trivial and it is easy to see that it yields the solution $\{\alpha \}, \{\beta \}$. The second constraint-set is $\{ (\beta \alpha \in \text{Int} \rightarrow \text{Bool}), (\text{Int} \rightarrow \text{Bool} \leq \beta) \leq \text{Int} \rightarrow \text{Bool} \}$. We apply solve to the constraint for $\alpha$ obtaining $\alpha = (\gamma \lor \beta) \land \text{Int} \rightarrow \text{Bool}$. We replace $\beta$ in the solution of $\alpha$ obtaining $\alpha = (\gamma \lor \text{Int} \rightarrow \text{Bool}) \land \text{Int} \rightarrow \text{Bool})$. The solution for this second constraint-set is then $\{ \text{Int} \rightarrow \text{Bool} \}$, $\text{Int} \rightarrow \text{Bool} \}$, which with $\{\alpha \}, \{\beta \}$ 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 after that the substitutions have been applied to obtain the types deduced by inference system: all variables that occur only in covariant (resp. contravariant) position in a type, can be replaced by 0 (resp. 1). This is what we implicitly did in our example to solve $\beta$ and eliminate $\gamma$ from the constraint of $\alpha$.

### 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-solutions $\{\sigma_i\}_{i \in I}, \{\sigma_j\}_{j \in J}$ for variables not in $\Delta$ that make both of these two inequations

$$\bigwedge_{i \in I} t\sigma_i \leq 0 \rightarrow \emptyset \bigwedge_{j \in J} \sigma_j \leq \text{dom}(\bigwedge_{i \in I} t\sigma_i)$$

hold. Two complications are to be dealt with: (i) we must find sets of type substitutions, rather than a single substitution as in the tallying problem and (ii) we have to get rid of the $\text{dom()}$ 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 $\sigma_k$ (for $k \in I \cup J$) in two substitutions: a renaming substitution $\rho_k$ that maps each variable in the domain of $\sigma_k$ into a different fresh variable, and a second substitution $\sigma_k'\rho_k$ defined such that $\sigma_k = \sigma_k' \circ \rho_k$. The two inequations thus become

$$\bigwedge_{i \in I} \text{dom}(\rho_i)\sigma_i' \leq 0 \rightarrow \emptyset$$

and

$$\bigwedge_{i \in I} \text{dom}(\rho_i\sigma_i) \sigma_i' \leq \text{dom}(\bigwedge_{i \in I} \rho_i\sigma_i)\sigma_i$$

Since the various $\rho_k^i$ (for $k \in I \cup J$) have disjoint domains, then we can take their union to get a single substitution $\sigma = \bigcup_{i \in I} \rho_i\sigma_i$, and the two inequations respectively become

$$\bigwedge_{i \in I} \rho_i \sigma_i \leq 0 \rightarrow \emptyset$$

and

$$\bigwedge_{i \in I} \rho_i \sigma_i \leq \text{dom}(\bigwedge_{i \in I} \rho_i \sigma_i)\sigma_i$$

Now if we fix the cardinalities of $I$ and $J$ since the $\rho_i$ 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 $\sigma$ such that

$$t_1 \sigma \leq 0 \rightarrow \emptyset$$

and

$$t_2 \sigma \leq \text{dom}(t_1 \sigma)$$

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

$$t_1 \sigma \leq 0 \rightarrow \emptyset$$

and

$$t_2 \sigma \leq (t_2 \rightarrow t_1) \sigma$$

with $\gamma$ fresh. We transformed the application problem (with fixed cardinalities) into the tallying problem for $\{(t_1, 0 \rightarrow t_1), (t_1, t_2 \rightarrow t_2)\}$, 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 $I$

---

1 Precisely, we have $t_1 = \bigwedge_{i \leq h} \text{Int} \rightarrow t_i^1$ and $t_2 = \bigwedge_{i \leq h} \text{Int} \rightarrow t_i^2$ where for

$$h = 1, 2$$ each $t_i^h$ is obtained from $t_0$ by renaming the variables not in $\Delta$ into fresh variables.
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 $\{((t_1, \xi \to \gamma))\}$ as explained in Subsection 3.2.2 (the constraint $t_1 \leq \xi \to \gamma$ is implied by this one since $0 \to \xi$ 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 $\Delta$) 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 some variable are merged. Thus even if the constraints fail, it may be the case that they will be satisfied by expanding different occurrences of some variable into different variables. Therefore an expansion is tried. Solving the problem for $s \subseteq t$ 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 C.54). We defined some heuristics (omitted for space reasons: see Section C.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 heuristics are based on the cardinalities and depths of the unions to be decidable. However, we fail to prove it when the substitutions inference is decidable, is an open problem. We believe

halting conditions preserve completeness, that is, whether type-

for space reasons: see Section C.2.3) to stop the algorithm

expansion that can make the constraint-set satisfiable. In

occurrences of type variables: if they fail, there is no possible

variable into different variables. Therefore an expansion is tried.

Solving the problem for $s \subseteq t$ 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 C.54). We defined some heuristics (omitted

for space reasons: see Section C.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 $\{((\alpha_1 \to \beta_1) \to \alpha_1 \to \beta_1) \leq t \to \gamma\}$ where $t = \{\text{Int} \to \text{Bool}\} \land (\alpha \to \text{Int} \to \alpha \to \text{Int})$ 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 [25] for more details on this example): one is unsatisfiable since it contains the constraint $t \leq 0$ (an intersection of arrows is never empty since it always contains $1 \to 0$ the type of the diverging functions); four of these are less general than some

(they are included in the solutions of the other) and the remaining four are obtained by adding the constraint $\gamma \geq \alpha_1 \to \beta_1 \land [(\alpha_1 \to \beta_1) \to \alpha_1 \to \beta_1] \leq t \to \gamma$, yielding the following four solutions for $\gamma$: $\{\gamma = [\text{Int} \to \text{Bool}]\}$, or $\gamma = [\text{Int} \to \text{Bool}]$, or $\gamma = [\text{Int} \to \text{Bool}]$). 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 strictly follow the algorithm, then we have to perform a further iteration, expand the type of the function, yielding $\{\{\alpha_1 \to \beta_1 \to \alpha_1 \to \beta_1\} \land ((\alpha_2 \to \beta_2) \to \alpha_2 \to \beta_2) \leq t \to \gamma\}$ for which the minimal solution is, as expected:

$\{\gamma = [\text{Int} \to \text{Int}] \land [\alpha \to \text{Int}] \to [\alpha \to \text{Int}]\}$

A final remark 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,
analyses are used to minimize the impact of polymorphism and so on). In this section we show examples of the latter, that is of

CDuce syntax extended with type variables.\footnote{Following the OCaml convention, in the concrete syntax type variables start by a quote. To enhance readability here we pretty print them by Greek letters and so write α → β rather than "α" \to "β".} In this extension map has type (α\to β)\to(\alpha\to\beta)\to[β|\alpha] since in CDuce sequence types are denoted by brackets whose content is described by a regular expression on types. Functions are explicitly typed and thus must specify both their input and output types

\[
\text{let pretty (x : Int): String = string_of x}
\]
or the whole interface when they are typed by an intersection type:

\[
\text{let even (x : Int\to Bool; (\alpha\Int)\to(\alpha\Int)) \mid x & Int \Rightarrow (x \mod 2) = 0} \mid x \Rightarrow x
\]
The type returned for the partial application \text{map} even is then

\[
\{((\alpha\Int)\to[(\alpha\Int)*])\&((\alpha\Int)*)\to[(\alpha\Int\Bool)*])\}
\]

(unsions and intersections are denoted by \& and \&\&, respectively) while the one for \text{map pretty} is \{([\alpha]\to[\alpha])\&([\alpha]*\to[\alpha]*)\}

While the right-hand side arrow of this intersection is the type

an ML programmer would expect, our inference algorithm also

deduces the special case

\[
\text{β}\to\alpha\
\]

for every type in the union. Instantiation works as expected as the

following snippet of CDuce interactive toplevel shows ("#" is the

prompt of the toplevel while its output is displayed in italics):

\[
\begin{align*}
\text{# let g : ((Int \to Int) \to Int \to Int) ; (Bool \to Bool) \to Bool \to Bool) x \to x \; ;}
\text{val g : (Bool \to Bool) \to Bool \to Bool} \\
\text{& (Int \to Int) \to Int \to Int = <fun>}
\end{align*}
\]

Our system includes singleton types (types that contain a single

value) and thus the type returned, for instance, for

\[
\text{let churchtrue (x: α; y: β): α = x in churchtrue 42 ;}
\]

is β \to 42 (ie, the type of functions that accept any argument and

return 42) and, likewise, id 42 has type 42. More surprising may be

the case for a function such as \text{max} (whose definition uses the

CDuce’s polymorphic comparison operator \gg (greater than)):

\[
\begin{align*}
\text{# let max : (x: α; y: α): α = if (x >> y) then x else y ;}
\text{val max : α \to α \to α = <fun>}
\end{align*}
\]

An ML programmer would probably expect the partial application

\text{max 42} to be typed as 42 \to 42 or \text{Int\to\Int} (at least, we were

naively expecting that). Instead, for this application the system re-

turns the type (β\to42)\to(β\to42), and rightly so. The point is that

our system includes union types and, therefore, an application such as

\text{max 42 "3"} is well typed: it suffices to instantiate the variable

α in the type of \text{max} by the union type \text{Int\toString}. To give the

final instantiation for α the type system must know the type of

both arguments of \text{max}, therefore in the case of the partial ap-

plication, it instantiates α with \beta\to42 stating that it knows that α con-

tains at least the value 42 and waits for the second argument to in-

stantiate the missing part, represented by \beta; and the type returned for

\text{max 42 "3"} is 42 \to "3". In this example we specified the type

α\toα\toα for \text{max}, since this is what an ML programmer would

have written. However, in a system with polymorphic union types a

more meaningful type for \text{max} is α\toβ\toα\toβ: if we specify such a

type for \text{max}, then the type deduced for the partial application

\text{max 42} is, more intuitively, β\to(β\to42). The fact that

α\toβ\toα\toβ and α\toα\toα cannot, from any practical point of view, be distin-

guished seems a nice feature of our system. Nevertheless notice that

the same type deduction as for \text{max} would have happened if in the

definition of \text{churchtrue} we had used \alpha instead of \beta; in that case

\text{churchtrue 42 "3"} would have been typed by (the less precise)

union type 42\to"3", too. Thus, in order to achieve precise typing, it

is important to use distinct type variables for distinct parameters.

Finally, we said in the introduction that the typing of even fol-

ows a pattern that is common in programming functional data

structures. This can be seen by examining Okasaki’s implementa-

tion of red-black trees [19]. These are balanced binary search

trees whose nodes are colored either in black or in red and such

that every red node has two (possibly empty) black children; a red

node with a red child is a “wrong” tree. Insertions must keep the

tree balanced and the key definition is a function balance which

transforms every “unbalanced” tree —ie, a black-rooted tree with a

“wrong” child— into a red-rooted tree, and leaves all other trees

unchanged. Okasaki gives a very compact and elegant definition of

balance consisting of a pattern matching with two cases (for a

union and for a capture variable), but current type systems are not

effective enough to verify that his code, without any modification,

satisfies color invariants. Our types, instead, can do it as follows:

\[
\begin{align*}
\text{type RBtree = Btree | Rtree} \\
\text{type Rtree = <red elem= α\to α> | BLtree} \\
\text{type Unbal = <red elem= α> | (Wrong RBtree)} \\
\text{type Wrong = <red elem= α> | (Wrong RBtree | RBtree Wrong)}
\end{align*}
\]

\[
\begin{align*}
\text{let balance ( Unbal -> Rtree ; (\exists Unbal) -> (\exists Unbal) )} \mid [] \mid \text{<blk (a)\[ a <red (y)\[ b <red (z)\[ c d \] \] \] \] } \\
\text{<blk (x)\[ a <red (z)\[ <red (y)\[ b c d \] ] \] } \\
\text{<blk (x)\[ a <red (y)\[ b <red (z)\[ c d ] \] ] } \\
\text{<blk (y)\[ <red (y)\[ b a ] \] } \\
\text{\mid x \Rightarrow x}
\end{align*}
\]

The only (irrelevant) difference of this definition with Okasaki’s

definition of balance is that we used CDuce’s syntax for trees, that

is, XML elements tagged by their color, with an attribute ele-

ment for their content, and that described sequences of two sub-

trees. The type of balance (which has the same form as the type of

even) precisely describes the behavior of the function and this type

information is enough to prove that the insertion function has type

α\toβ\toβ\toβ, that is, that when it inserts an α-element into a

well-formed black-rooted red-black it returns another well-formed

black-rooted red-black tree (see Appendix A for the complete code

and how to run it on the development version of CDuce).

Transposing the results and algorithms of this paper, to full-

fledged CDuce was not easy. Adapting the internal representation of

types and its algorithms is challenging (to give an idea of such a

challenge, consider that a simple type variable is internally repre-

sented as a hash-cased union of 7 binary decision diagrams each

intersecting the top type of a type constructor) and so are type pretty

printing and error message generation. For what is specific to this

work, the main challenge is not only to extend the typing rules of

Figure 2 to missing data structures and expressions (XML trees,

general pattern matching, products,…) but, above all, to modify the

rules so that they return expressions decorated with sets of explicit

type substitutions. Also the various internal languages used by the

compiler must be modified (the CDuce compiler performs several

passes that translate each intermediate language into a lower level

one) and each transformation phase must be enriched with specific

static analyses to optimize the evaluation of polymorphic expres-
sions. Finally, the propagation of type substitutions must be lazily implemented for all constructed values (ie, pairs and XML documents). The main modifications are summarized in Appendix E.

For what concerns performances, the results of some preliminary experiments are reported in Appendix F. In summary, we generated a test suite of 1 859 applications of higher-order polymorphic functions by taking all the 43 functions exported by the List module of OCaml, and cross applying one to each other. Whenever a given function can be applied (ie, the application type-checks in OCaml) to two other functions, then we applied it to their intersection and their union; whenever two functions can be applied to the same function then we applied their union and their intersection to it; and so on so forth up to a maximum of 15 top-level connectives. We also tested all applications resulting ill-typed in OCaml, so as to check cases in which local type-inference may fail. The results are significant and encouraging. The test suite was defined to maximize the possible exponential blow up (which is essentially due to the presence of arrows and intersections that may trigger multiple expansions). The combinations of the functions of the List module cover a wide range of use cases and involve recursive types (specifically, polymorphic lists), and the limit of 15 connectives on arrows more than doubles what we ever met in practice. To type check the 1 859 applications on an average laptop took 27 secs with an average time for application of 14ms and 2.1ms of median time. This means that apart from few pathological cases (which took a couple of seconds) our implementation performs local type inference within acceptable delays. We also verified that our implementation smoothly handles the application of curried functions to 20 arguments (cf, OCaml standard library whose functions have at most 5 arguments). Furthermore, the room for improvement is still important. Our implementation uses the highly-optimized data-structures of CDuec types and aggressive memoization, however normalization and constraint generation are implemented as described in this paper. In particular, as in Line 5 in Figure 3, normalization performs a full expansion. By modifying the algorithms so that, like in the subtyping algorithm, normalization and constraint generation are performed lazily, we hope to improve performance by an order of magnitude.

4. Type reconstruction

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. The idea is to type the body of a function under the hypothesis that the function has the generic type α→β and deduce the corresponding constraints. Formally, we consider expressions produced by the following grammar:

\[ m ::= c \mid x \mid mn \mid \lambda x.m \mid m@t?m : m \]

together with the judgment \( \Gamma \vdash m : t \rightarrow \mathcal{J} \) that states that under the typing environment \( \Gamma \), \( m \) has type \( t \) under the constraints in \( \mathcal{J} \), provided that \( \mathcal{J} \) is satisfiable (the turnstile subscript \( \mathcal{J} \) indicates that this is a type Reconstruction system). 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 solutions of the satisfiable constraint-sets in \( \mathcal{J} \)) 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 \( \alpha \rightarrow \beta \) and, to compensate that, we infer type-substitutions

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) \rightarrow \{ \emptyset \} & \quad \text{(R-VAR)} \\
\Gamma \vdash m_1 : t_1 \rightarrow \mathcal{J}_1 & \quad \text{(R-APP)} \\
\Gamma \vdash m_0 \rightarrow \mathcal{J}_0 \rightarrow \mathcal{J}_2 \land \{ \{ t_1 \leq t_2 \rightarrow \alpha \} \} & \quad \text{(R-ABSTR)} \\
\end{align*}
\]

where \( \alpha, \alpha_1 \) and \( \beta \) in each rule are free type variables.

\begin{figure}

Figure 4. Type reconstruction rules

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 \( \Pi \rightarrow 0 \rightarrow 0 \) — 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, \( \alpha \rightarrow \beta \)). 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, [21]) 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 can type auto-application \( \lambda x.x \) for which it returns the recursive type \( t = \mu X.(\alpha \wedge (X \rightarrow \beta)) \rightarrow \beta \). This type is a subtype of \( \alpha \rightarrow \beta \) and is not always very readable and often needs to be simplified. For instance, the type we showed for \( \text{map} \) was obtained after apply-
ing some simplifications—one of which was done by hand—and defining an algorithm that does the right simplifications is not obvious (e.g., how to detect that the type \((\alpha \land (\alpha \rightarrow \beta)) \rightarrow \beta\) is much more readable than the type \(\mu X. (\alpha \land (X \rightarrow \beta)) \rightarrow X\) 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, even though we will probably have to couple it with bidirectional typing techniques \[9\] to provide informative error messages when the check fails.

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 and are dealt with in a standard way, while pairs are more challenging. The rule for pairs in inference system \(\vdash_{f}\) 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: \(t_{1} \neq 0\) and \((t_{1} \times t_{2}) \subseteq (s_{1} \times s_{2}) \iff t_{1} \subseteq s_{1}\)). Instead, as expected, the rule for projection \(\pi_{e}\) needs some special care since if the type inferred for \(e\) is say, \(t\), then we need to find a set of substitutions \([_{i}]\mid \in S\) such that \(\lambda _{i} \vdash \pi_{e} t_{i} \leq 2 \times 1\). 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 \(I\) (see the appendix).

In the first part of this work \[3\] we studied the extension of the explicitly-typed calculus with \(\lambda\text{-}\text{polymorphism}, in particular, its typing and efficient execution (see Section 5.4 of \[3\]). There we distinguished \(\lambda\text{-}\text{bound variables by underlining them. Reconstruction is mostly useful when combined with } \lambda\text{-polymorphism. To extend our reconstruction to } \lambda\text{- we use a separate type environment } \Phi \text{ for these variables (while we reserve } \Gamma \text{ for } \lambda\text{-abstracted variables). As in Damas-Milner } \pi \text{ algorithm \[8\] we need to define } \overline{\Gamma}(t), \text{ the generalization (closure in \[8\]) of a type } t \text{ wrt the type environment } \Gamma, \text{ that is, } \overline{\Gamma}(t) \equiv t \{\forall t_{i} \in \text{var}(t), \forall \alpha \in \text{var}(\Gamma)\} \text{ where } t_{i} \text{ are fresh. Then the rules for type reconstruction are} \begin{align*}
\text{(let-var)} \quad & \Phi \vdash_{f} \Gamma \vdash_{f} e_{1} : t_{1} \rightarrow \mathcal{J} \quad \Phi \vdash_{f} (\Phi(x)) \vdash_{f} \{\emptyset\} \\
\text{(let)} \quad & \Phi \vdash_{f} \Gamma \vdash_{f} 1 e_{1} : t_{1} \rightarrow \mathcal{J} \quad \Phi \vdash_{f} (\Phi(x)) \vdash_{f} e_{2} : t_{2} \rightarrow \mathcal{J'} \quad \Phi \vdash_{f} (\Phi(x)) \vdash_{f} e_{1} \in e_{2} : t_{2} \rightarrow \mathcal{J} \land \mathcal{J'}
\end{align*}

Finally, we want to stress that there is at least a case in which we should have used 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 a polymorphic type (see Section B.3).

6. Related work

This section discusses related work on constraint-based type inference and inference for intersection/union type systems. Discussion about work on explicitly-typed intersection type systems and on XML processing languages can be found in Part I of this work \[3\].

Type inference in ML has essentially been considered as a constraint solving problem \[18, 21\]. 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 (ie, a supertype of) 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, solving the problem of application demands to find sets of substitutions rather than just one substitution. This is reflected by the definition of our \(\subseteq\) 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 \(s_{1} \leq \alpha \leq s_{2}\) into sets of equations of the form \(\alpha = (\beta \lor s) \land t\), a technique that, in our ken, is original to our work. This set of equations is then solved using Courcelle’s work on infinite trees \[7\]. 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 \[18, 21\] 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 \[18, 21\] to encompass our richer setting. This could be a first step towards the study of efficient constraint solving algorithms for our system.

Feature-wise the programming language closest to our language—is, polymorphic C\text{Duce} is Typed Racket \[22, 23\] since it has recursive types, union types, top and singleton types, subtyping, dynamic type-cases (called occurrence typing in \[22, 23\]), and explicitly-typed polymorphic functions. The goal of Typed Racket is to type an existing untyped programming language and it is superior to our system in that it allows the combination of both typed and untyped code in a single program. For what concerns typed features, however, Typed Racket is just a small fragment of our system: type-cases can only test basic types and tests for just some constructed types can be encoded by using Boolean connectives; there are no intersection types (thus, no overloaded functions); there are no negation or difference types; union types and their subtyping are quite naive (e.g., a type is a subtype of a union of types only if it is a subtype of some type of the union, distribution laws over type constructors are absent, etc.). The typing of Typed Racket is inherently defined in terms of propositional logic where atoms are the elements of a type environment (e.g., \(x : \tau\)). The use of logical formulas coincides in \text{Cduce} to computing the types of capture variable of patterns (cf. the operator \(t/p\) in Appendix E or in \[11\]): the use of propositional logic corresponds to the use of Boolean connectives in \text{Cduce}’s patterns and unsatisfiability of a formula to type emptiness. This is why all the examples in the “Challenges” section of \[23\] can be easily defined and precisely typed in our system (straightforwardly with the syntactic sugar defined in the Appendix E of Part I \[3\]). Actually, these examples can already be defined and typed in monomorphic \text{Cduce} \[2\] since it already captures all the features characteristics of Typed Racket (recursive and union types, subtyping, occurrence typing, etc.) with the sole exception of polymorphism, a gap filled by this work. Typed Racket uses a limited form of local type inference: the application of a function to a polymorphic argument requires the application of an explicit type substitution to the argument. We do not have this limitation thanks to our tallying procedure that computes instantiations (type substitutions) both for the function and for its argument. It is not clear whether using generic SMT solvers for typing (as suggested by \[23\]) also in \text{Cduce} case (where subtyping is checked by type emptiness) would yield a better (sub)typing algorithm.
Local type inference was first formalized, as far as we know, by Pierce and Turner [20]. They consider (i) a type system with type variables, arrow, top, and bottom types, (ii) an internal language with explicitly typed polymorphic functions that, to be applied, must be explicitly instantiated, and (iii) an external language in which some or all such instantiations can be omitted. Then they show how to infer type instantiations for programs of the external language in order to obtain, when possible, well-typed programs of the internal language. Our work shares much of the philosophy and goals of Pierce and Turner [20]: expressions of grammar (3) form our external language, those of grammar (7) the internal one, and our sets of type-substitutions generalize Pierce and Turner’s instantiations. Our work extends and generalizes Pierce and Turner’s one in several ways. First, in an application we infer instantiation/type-substitutions both for the function and for the argument, while [20] does just the former (Typed Racket does the same). As a consequence while the application of the polymorphic identity \( \lambda \, x.x \) to a function \( f \) of type \( \text{Int} \to \text{Int} \to \text{Int} \) can be typed in their systems (by inferring the instantiation \( \{ \text{Int} \to \text{Int} \to \text{Int}/f \} \)) the application of the same \( f \) to the polymorphic identity cannot be typed (while our system types it by instantiating the argument by the substitution \( \{ \text{Int}/f \} \)). Second, our system accounts for more expressive types, expressions, and subtyping relations. For instance, [20] essentially solves the tallying problem for simple constraint-sets whose form is the same as those obtained after applying our merge: instead we manipulate sets of constraint-sets (to account for alternatives generated to check either the typing of type-cases or the subtyping relation) and iterate the tallying problem with different cardinalities because of the presence of intersection types. Third, [20] synthesizes an instantiation for a type variable only if its occurrences are either all positive or all negative and fails otherwise, while our system, thanks to the use of type connectives and recursive types, always generates a set of solvable equations.

Finally, we want to stress as a caveat 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 intersection 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 a few similarities with some techniques developed for pure intersection type systems that we briefly discuss next.

Coppo and Giannini [6] 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 \( M \to N \), 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 [24] 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 here, together with the one in [3], provides the theoretical bases and all the algorithmic tools needed to design and implement polymorphic languages for semi-structured data and, more generally, generic functional languages with recursive types and set-theoretic unions, intersections, and negations. In particular, our results made the polymorphic extension of \( \text{C}\text{Duce} \) [2] possible and pave the way to the definition of a real type system for \( \text{XQuery} \) 3.0 [10] (not the current one in which all higher-order functions have type \( \text{function}() \)). Thanks to type reconstruction, these languages can have a syntax and semantics close to those of \( \text{OCaml} \) or Haskell, but also include primitives (in particular, complex patterns) that exploit the great expressive power of full-fledged set-theoretic types. Symmetrically, as the red-black trees and max examples in Section 3.3 demonstrate, \( \text{OCaml} \) and Haskell would certainly benefit from the addition of set-theoretic type connectives: we plan to study such an extension in the near future.

Some problems are still open, notably the decidability of type substitution inference, but these are of theoretical nature and, as our experiments hitherto confirm, should not have any impact in practice. 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 \( \text{C}\text{Duce} \) 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, by implementing the polymorphic extension of \( \text{C}\text{Duce} \) and applying it we realized that it would be useful to allow monomorphic type variables to occur in patterns (see Appendix A for examples) which in this work this would correspond to have a type case on types that may contain monomorphic type variables. How to do it is not straightforward and looks as a promising research direction. Other interesting practically-oriented 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 extension of the bridge between \( \text{OCaml} \) and \( \text{C}\text{Duce} \) to include polymorphic types and, later on, the inclusion of GADTs.

Acknowledgments. We want to thank the reviewers of POPL who gave detailed suggestions to improve our presentation. 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

Appendix
A. Examples of code

In this section we show examples of real code that follows the same pattern as the `even` function we defined in the introduction.

A.1 Red-Black trees

As a first example we show that the use of polymorphic set-theoretic types yields a better definition of Okasaki’s implementation of red-black trees.

A red and black tree is a colored binary search tree in which all nodes are colored either black or red and that satisfies 4 invariants:

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from the root to a leaf contains the same number of black nodes

Thanks to our type system (and contrary to Okasaki’s version) the implementation below ensures by typing that the operations on red-black trees (notably, the insertion) satisfy the first three invariants, as well as, that the `ins_aux` function, local to insertion, never returns empty trees (yet another important property that, in ML/Haskell Okasaki’s version, types cannot ensure).

```plaintext
type RBtree(α) = Btree(α) | Rtree(α)

(* Black rooted RB tree: *)
type Btree(α) = [[] | <black elem=α>[ RBtree(α) RBtree(α) ]

(* Red rooted RB tree: *)
type Rtree(α) = <red elem=α>[ Btree(α) Btree(α) ]

type Wrongtree(α) = <red elem=α>[ [ Rtree(α) Btree(α) ] |
                     [ Btree(α) Rtree(α) ] ]

type Unbalanced(α) = <black elem=α>[ [ Wrongtree(α) RBtree(α) ] |
                     [ RBtree(α) Wrongtree(α) ] ]

let balance ( Unbalanced(α) -> Rtree(α) ; β|Unbalanced(α) -> β\Unbalanced(α) ) |
      ( <black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ] |
      <black (z)>[ a <red (x)][ b <red (y)][ c d ] ] ) & Unbalanced(α) |
     -> <red (y)>[ <black (x) )[ a b ] <black (z)][ c d ] ]
      |
     x -> x

let ins_aux (x : α) (t : Btree(α)) : Btree(α) =
  let ins_aux ( [] -> Rtree(α) ;
               Btree(α)\[] -> RBtree(α)\[] ;
               Rtree(α) -> Rtree(α)|Wrongtree(α) ) |
  | [] -> <red elem=x>[ [] [] ]
  | [<(color) elem=y][ a b ] & z ->
     if x << y then balance <(color) elem=y>[ (ins_aux a) b ]
     else if x >> y then balance <(color) elem=y>[ a (ins_aux b) ]
     else z
  in match ins_aux t with |
  | _ (y)[ a b ] -> <black (y) )[ a b ]
```

We invite the reader to refer to the excellent Okasaki’s monograph [19] for details about Okasaki’s algorithm—that our code faithfully follows—and the documentation of the language CDuce for details about the syntax we used, which is standard CDuce syntax apart from the presence of type variables. 8 Let us instead spend some words to comment the types, since they are the real novelty and the added value of our definition. First, notice that we used the full palette of our types: unions, intersections, negations (difference), and type variables. Red-black trees (Btrees) are black-rooted RBtrees (invariant 1), which

---

8 For the reader convenience we recall that in CDuce XML types/pattern/expressions may have the form

```
<tag attr=type/pattern/expression ... attr=type/pattern/expression>[ sequence of types/patterns/expressions ]
```

and that possibly recursive functions can be defined as

```
let name (type->type; ... ; type->type)
   | pattern -> expression | ... | pattern -> expression
```

where the list of arrow types that follow the function name form the intersection type (ie, the interface) of the function.
are themselves black-rooted trees or red-rooted trees. The difference between the last two is that the latter
cannot be leaves (invariant 2) and their children can only be black-rooted trees (invariant 3).

The \texttt{insert} function takes an element $x$ of type $\alpha$, and returns a function that maps red-black trees into
red-black trees.

\begin{verbatim}
insert :: $\alpha$ -> Btree($\alpha$) -> Btree($\alpha$)
\end{verbatim}

By examining the code of \texttt{insert} it is easy to see that if the argument tree is empty a red-rooted tree is
returned, otherwise the element is inserted in the appropriate subtree and the whole tree is then balanced by
the function \texttt{balance}. This function has the following type (which follows the same typing pattern as the
function \texttt{even} defined in the introduction):

\begin{verbatim}
balance :: ( Unbalanced($\alpha$)->Rtree($\alpha$) ) & ( $\beta$\Unbalanced($\alpha$)->$\beta$\Unbalanced($\alpha$) )
\end{verbatim}

This type states that \texttt{balance} transforms an unbalanced tree into a (balanced) red-rooted tree and leaves all
other trees (in particular the balanced ones) unchanged. The core of our definition is the type of \texttt{ins_aux}:

\begin{verbatim}
ins_aux :: ( [] -> Rtree($\alpha$) )
& ( Btree($\alpha$)
\[
\] -> RBtree($\alpha$)
\[
\] )
& ( Rtree($\alpha$) -> Rtree($\alpha$)|Wrongtree($\alpha$) )
\end{verbatim}

which precisely describes the behaviour of the function. Notice that the domain of \texttt{ins_aux} (which is
the union of the three domains of the arrows forming its intersection type) is exactly \texttt{RBtree($\alpha$)}. The
intersection type describes the behaviour of \texttt{ins_aux} for each form of an \texttt{RBtree}—ie, empty, black-rooted,
and red-rooted—. The type system needs the full precision of this type to infer whether the calls to \texttt{balance}
in the body of \texttt{ins_aux} are applied to a balanced or an unbalanced tree: even a slight approximation of this
type, such as

\begin{verbatim}
ins_aux :: ( Btree($\alpha$)
\[
\] -> RBtree($\alpha$)
\[
\] )
& ( Rtree($\alpha$) | [] -> Rtree($\alpha$) | Wrongtree($\alpha$) )
\end{verbatim}

makes type-checking fail. By examining the type of \texttt{ins_aux} it is easy to see that \texttt{ins_aux} always returns
either a (balanced) black-rooted tree or a tree with a red root in which one of the children may be a \texttt{Rtree}.
In case of a tree with a red root, a (balanced) red-black tree is then obtained by changing the color of the
root to black, as it is done in the last line of \texttt{insert}.

The implementation above must be compared with the corresponding version in monomorphic C\textsc{Duce}:

\begin{verbatim}
type RBtree = Btree | Rtree
type Btree = [ ] | <black elem=Int>[ RBtree RBtree ]
type Rtree = <red elem=Int>[ Btree Btree ];;
type Wrongtree = Wrongleft | Wrongright
type Wrongleft = <red elem=Int>[ Rtree Btree ]
type Wrongright = <red elem=Int>[ Btree Rtree ]
type Unbalanced = <black elem=Int>([ Wrongtree RBtree ] | [ RBtree Wrongtree ]);;

let balance ( Unbalanced($\alpha$)->Rtree($\alpha$) ) & ( $\beta$\Unbalanced($\alpha$)->$\beta$\Unbalanced($\alpha$) )
\end{verbatim}

which, besides being monomorphic, requires the introduction of several intermediate types (in particular
Wrongleft and Wrongright) in order to describe the polymorphic behavior of \texttt{balance}—whose type
results, thus, much more obscure—. Our implementation must also be compared with the version given by
Rowan Davies in his PhD Thesis \cite{Davies} which uses polymorphic intersection types and type inference. Contrary
to our definition, Davies's implementation \((i)\) does not statically verify invariant 1, \((ii)\) it introduces—as
our monomorphic version does—several intermediate type definitions to specify the behavior of local
functions, and (iii) it does not faithfully reproduce Okasaki implementation since it needs the definition of several auxiliary functions absent from Okasaki’s (and our) formulation.

Likewise, there exist other implementations that are able to ensure/verify the first invariants of red-black trees (e.g., by using GADTs or finite tree automata) but they all need extra definitions of intermediate functions or operations: as far as we know ours types are the only system that can statically ensure the invariants above simply by decorating (with types) the original Okasaki’s code without any further modification.

Notice that the definition of balance given above and the one in Section 3.3 differ for a couple of details. First, the name of the types require mandatory type parameters when their definitions contain free type variable (e.g., we have to write RBtree(α) rather than just RBtree): this is the behavior implemented in the current development version of CDuce, and we omitted this detail in the code of Section 3.3 just for space reasons. More importantly, the union pattern of the first branch of the pattern matching is intersected with the type Unbalanced(α). The reason is that this type is strictly contained in the type accepted by the union pattern and, therefore, this branch can be selected for values that are not in Unbalanced(α) (notice that balance can be applied to any value). Now for these values the interfaces of balance declares that a result of the same type is returned, which is not true since the first rather than the second branch is selected (and the latter transforms a black-rooted tree into a red-rooted one). This is why without the intersection in the pattern the type-checker rejects the definition by pointing out the problem. By adding the intersection we force the first branch to be selected only for values of type Unbalanced(α), and the function type-checks.

There is still a last glitch, at least to run the example on the current development version of CDuce. Notice that we used Unbalanced(α) in the pattern and that this type contains the monomorphic variable α. The current development version of CDuce does not allow monomorphic variables to occur in patterns, yet (this is listed as future work in Section 7). In order to execute balance on the current development version of CDuce there are (at least) three solutions, which are all as valid as the one presented before. The first solution is to restrict the domain of balance to Unbalanced(α) | RBtree(α). This can be done by declaring for balance the interface ( Unbalanced(α) -> Rtree(α) ; β & RBtree(α) -> β & RBtree(α) ) (notice that the intersection of Unbalanced(α) and RBtree(α) is empty so the difference in the interface and the intersection in the pattern are no longer necessary). The second solution is to use as intersection in the pattern the type Unbalanced(Any) to overapproximate Unbalanced(α). This however requires to modify the interface of balance accordingly into ( Unbalanced(α) -> Rtree(α) ; β | Unbalanced(Any) -> β | Unbalanced(Any) ) to capture the precise cases in which the second branch is selected. The third, more verbose, solution is to get rid again of the intersection pattern by declaring in the interface that the second arrow type applies only to values that are not in the accepted type of the first union pattern, that is, ( Unbalanced(α) -> Rtree(α) ; β | UTree -> β | UTree ) where UTree is the accepted type of the first union pattern, which is obtained by replacing in the pattern Any for every capture variable occurring in it, namely:

```haskell
```

A.2 Soap envelopes

As a second usage example of the typing pattern followed by even we explore a typical XML application to process envelopes. Soap envelopes are a standardized format to communicate information wrapped in XML. An envelope contains a body and an optional header as described by the following type definitions.

```haskell
type Envelope(α,β) = <Envelope>[ Header(β) ? Body(α) ]
type Header(β) = <Header>β
type Body(α) = <Body>α
```

We define an enrich function which maps functions into a function with the same typing pattern as even (NOTE: in the current development version of CDuce this function cannot be executed because neither the :: notation, nor monomorphic variables in patterns are implemented, yet).

```haskell
define enrich :: ( α -> β ) ->
  ( ( Envelope(α,β) -> <Envelope>[Header(β) Body(α)] )
  & ( γ | Envelope(α,β) -> γ | Envelope(α,β) )
)
```

```haskell
enrich f x = match x with
  | (Envelope[Header f b]) & Envelope(α,β) ->
    <Envelope> [ <Header>(f b) ] Body(α | enrich f b ]
  | (Envelope[Header h Body f b]) & Envelope(α,β) ->
    <Envelope> [ <Header>((f b)b@h) ] Body(α | enrich f b ]
  | lst & ( AnyXML | Envelope(α,β)) | Char* ] -> (map lst with y -> enrich f y)
```
| <(x)>y -> <(x)>(enrich f y) |
| y -> y |

When applied to a function \( f \), \( \text{enrich} \) returns a function that adds to the header of an envelope the result obtained by applying \( f \) to the body of the envelope, and recursively applies this transformation inside the body and in possible subtrees. Once more, in the definition of \( \text{enrich} \) we used pattern with monomorphic variables, therefore the same considerations as for the red-black tree example apply, too. The function \( \text{enrich} \) can then be typically used as in

\[
\text{xtransform \ anXMLdoc with} \\
| x & T -> enrich f_T x \\
| x & <Envelope>_ -> enrich f x
\]

where \( f_T \) is specific for type \( T \) and \( f \) is generic. The expression above transforms all the envelopes in the \( \text{anXMLdoc} \) document by preserving the type of all its subcomponents with the addition of the information on the headers, when it is missing.

Again this must be contrasted with the monomorphic version which must list all possible alternatives for the input type and in which the types of the contents of the envelope and the header are not preserved since they are subsumed to \( [(\text{AnyXml}|\text{Char})^*] \):

\[
\text{type Envelope} = <\text{Envelope}[ \text{Header} \text{ Body} ]> \\
\text{type Header} = <\text{Header}> B \\
\text{type Body} = <\text{Body}> A \\
\text{type A} = [(\text{AnyXml}|\text{Char})^*] \\
\text{type B} = [(\text{AnyXml}|\text{Char})^*] \\
\]

let enrich (f: A -> B): ( (Envelope -> <Envelope>[ Header Body ]) \\
k & (A -> A) & (Char -> Char) & (AnyXml -> AnyXml)) = \\
(fun ( Envelope -> <Envelope>[ Header Body ] ; \\
A -> A ; Char -> Char ; AnyXml -> AnyXml) \\
| <Envelope>[ <Body> b ] -> \\
<Envelope> [ <Header>(f b) <Body>(enrich f b) ] \\
| <Envelope> [ <Header> h <Body> b ] -> \\
<Envelope> [ <Header>((f b)@h) <Body>(enrich f b) ] \\
| lst & [ (AnyXml|Char)* ] -> (map lst with y -> enrich f y) \\
| <(x)>y -> <(x)> (enrich f y) \\
| y -> y);;

B. 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, 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 \( \lambda \)-abstractions, and we therefore look for completeness of the type-substitutions inference system with respect to the expressions written according to the following grammar:

\[
e ::= c | x | (e, e) | \pi_i(e) | e \ e | \lambda^{i_1, \ldots, i_s}_x. e | e \in t \ ? \ e | e [\sigma_j]_{j \in J}.
\]

We write \( \mathcal{E}_0 \) for the set of such expressions. The implicitly-typed calculus defined in this section corresponds to the type-substitution erasures of the expressions of \( \mathcal{E}_0 \). 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 \( (\text{ALG-INST}) \) 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.

B.1 Implicitly-typed Calculus

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

\[
a ::= c | x | (a, a) | \pi_i(a) | a \ a | \lambda^{i_1, \ldots, i_s}_x. a | a \in t \ ? \ a : a
\]

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

Clearly, \( \mathcal{E}_A \) is a proper subset of \( \mathcal{E}_0 \).

The erasure of explicitly-typed expressions to implicitly-typed expressions is defined as follows:
Definition B.2. The erasure is the mapping from $\mathcal{E}_0$ to $\mathcal{E}_A$ defined as
\[
\begin{align*}
erase(c) & = c \\
erase(x) & = x \\
erase((e_1, e_2)) & = (\ erase(e_1), \ erase(e_2)) \\
erase(\pi_i(e)) & = \pi_i(\ erase(e)) \\
erase(\lambda^{i : I} x \rightarrow s_1) & = \lambda^{\pi_i : I} \ erase(x) \\
erase(e_1) & = \ erase(e_1) \\
erase(e_2) & = \ erase(e_2) \\
erase(e[\sigma_j]_{i \in J}) & = \ erase(e)
\end{align*}
\]

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 B.3. Let $s$ and $t$ be two types and $\Delta$ a set of type variables. We define the following relations:
\[
\begin{align*}
[s_i]_{i \in I} \triangleright s & \subseteq \Delta t & \iff & \bigwedge_{i \in I} s_i \leq t \text{ and } \forall i, s_i \notin \Delta \\
 s & \triangleleft \Delta t & \iff & \exists [s_i]_{i \in I} \text{ such that } [s_i]_{i \in I} \triangleright s \subseteq \Delta t
\end{align*}
\]

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

Lemma B.4. Let $t_1$ and $t_2$ be two types and $\Delta$ a set of type variables. If $t_1 \subseteq \Delta s_1$ and $t_2 \subseteq \Delta s_2$, then $(t_1 \times t_2) \subseteq \Delta (s_1 \times s_2)$.

Proof. Let $[s_i]_{i \in I} \triangleright t_1 \subseteq \Delta s_1$ and $[s_i]_{i \in I} \triangleright t_2 \subseteq \Delta s_2$. Then
\[
\begin{align*}
\bigwedge_{i \in I} (t_1 \times t_2) & = \bigwedge_{i \in I} (t_1 \sigma_i) \times \bigwedge_{i \in I} (t_2 \sigma_i) \\
 & \leq \bigwedge_{i \in I} s_i \times s_{i}
\end{align*}
\]

and
\[
\begin{align*}
\bigwedge_{i \in I} (t_1 \times t_2) & = \bigwedge_{i \in I} (t_1 \times s_i) \times \bigwedge_{i \in I} (t_2 \times s_i) \\
 & \leq \bigwedge_{i \in I} (s_1 \times s_i) \\
 & \leq \bigwedge_{i \in I} (s_1 \times s_i) \times \bigwedge_{i \in I} (s_2 \times s_i)
\end{align*}
\]

\hfill $\square$

Lemma B.5. Let $t_1$ and $t_2$ be two types and $\Delta$ a set of type variables such that $(\text{var}(t_1) \setminus \Delta) \cap (\text{var}(t_2) \setminus \Delta) = \emptyset$. If $t_1 \subseteq \Delta s_1$ and $t_2 \subseteq \Delta s_2$, then $t_1 \triangleright t_2 \subseteq \Delta s_1 \triangleright s_2$.

Proof. Let $[s_i]_{i \in I} \triangleright t_1 \subseteq \Delta s_1$ and $[s_i]_{i \in I} \triangleright t_2 \subseteq \Delta s_2$. Then we construct another set of type substitutions $[s_{i_1, i_2}]_{i_1 \in I, i_2 \in I}$ such that
\[
s_{i_1, i_2} = \begin{cases} 
\sigma_{i_1}(\alpha) & \text{if } \alpha \in (\text{var}(t_1) \setminus \Delta) \\
\sigma_{i_2}(\alpha) & \text{if } \alpha \in (\text{var}(t_2) \setminus \Delta) \\
\alpha & \text{otherwise}
\end{cases}
\]

So we have
\[
\begin{align*}
\bigwedge_{i_1 \in I, i_2 \in I} (t_1 \lor t_2) & = \bigwedge_{i_1 \in I} \bigwedge_{i_2 \in I} (t_1 \lor t_2) \\
 & = \bigwedge_{i_1 \in I} (s_{i_1} \lor s_{i_2}) \\
 & = \bigwedge_{i_1 \in I} \bigwedge_{i_2 \in I} (t_1 \lor t_2)
\end{align*}
\]

Notice that two successive instantiations can be safely merged into one (see Lemma B.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 B.6. Let $e$ be an explicitly-typed expression and $[s_i]_{i \in I}, [s_j]_{j \in J}$ two sets of type substitutions. Then
\[
\Delta; \Gamma \vdash_{\text{xt}} (e[s_i]_{i \in I})[s_j]_{j \in J} : t \iff \Delta; \Gamma \vdash_{\text{xt}} e[s_{i, j}]_{i \in I, j \in J} : t
\]
Formally, we have the following inference rule for projections have Lemma B.7.

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e : s, \sigma_i \not\in \Delta}{\Delta, \Gamma \vdash \pi(e)[\sigma_i] : \bigwedge_{i \in I} s(\sigma_i), \sigma_j \not\in \Delta} \\
\end{align*}
\]

As \(\sigma_i \not\in \Delta, \sigma_j \not\in \Delta\) and \(\text{dom}(\sigma_j \cup \sigma_i) = \text{dom}(\sigma_i) \cup \text{dom}(\sigma_j)\), we have \(\sigma_j \circ \sigma_i \not\in \Delta\). Then by (ALG-INST), we have \(\Delta, \Gamma \vdash e(\sigma_j \circ \sigma_i) : \bigwedge_{i \in I} s(\sigma_j \circ \sigma_i)\), that is \(\Delta, \Gamma \vdash e(\sigma_j \circ \sigma_i) : \bigwedge_{i \in I} s(\sigma_i)\).

\[\iff \]

Consider the following derivation:

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e : s, \sigma_i \not\in \Delta}{\Delta, \Gamma \vdash \pi(e)[\sigma_i] : \bigwedge_{i \in I} s(\sigma_i), \sigma_j \not\in \Delta} \\
\end{align*}
\]

As \(\sigma_j \circ \sigma_i \not\in \Delta\) and \(\text{dom}(\sigma_j \circ \sigma_i) = \text{dom}(\sigma_i) \cup \text{dom}(\sigma_j)\), we have \(\sigma_i \not\in \Delta\) and \(\sigma_j \not\in \Delta\). Then applying the rule (ALG-INST) twice, we have \(\Delta, \Gamma \vdash e(\sigma_i) : \bigwedge_{i \in I} s(\sigma_i)\), that is \(\Delta, \Gamma \vdash e(\sigma_i) : \bigwedge_{i \in I} s(\sigma_i)\).

\[\Box\]

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

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e_1 : s \quad \Delta, \Gamma \vdash e_2 : t}{\Delta, \Gamma \vdash e_1 \times e_2 : s \times t} \\
\end{align*}
\]

We rewrite such a derivation as follows:

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e_1 : s \quad \Delta, \Gamma \vdash e_2 : t}{\Delta, \Gamma \vdash e_1 \times e_2 : s \times t} \\
\end{align*}
\]

Clearly, \(\bigwedge_{i \in I \cup J} (t_1 \times t_2)\sigma_i \leq (\bigwedge_{i \in I} t_1 \sigma_i) \times (\bigwedge_{i \in J} t_2 \sigma_i)\). Then by subsumption we can deduce that \((e_1, e_2)[\sigma_i]_{i \in I \cup J}\) also has the type \((\bigwedge_{i \in I} t_1 \sigma_i) \times (\bigwedge_{i \in J} t_2 \sigma_i)\). 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.

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e_1 \times e_2 : s \times t}{\Delta, \Gamma \vdash e_1 \times e_2 : s \times t} \\
\end{align*}
\]

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

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e : s \quad \forall j \in J, \sigma_j \not\in \Delta}{\Delta, \Gamma \vdash e(\sigma_j) : \bigwedge_{i \in I \cup J} \sigma_j} \\
\end{align*}
\]

According to Lemma C.8 in the companion paper [3], we have \(\pi_i(\bigwedge_{j \in J} \sigma_j) \leq \bigwedge_{j \in J} \pi_i(\sigma_j)\), but the converse does not necessarily hold. For example, \(\pi_1(\{(t_1 \times t_2) \lor (s_1 \times s_2)\} \{\alpha \} \{\beta\}) = \{t_1 \lor s_1\} \{\alpha \} \{\beta\}\). We cannot exchange the instantiation and projection rules without losing completeness. However, as \((\bigwedge_{j \in J} \sigma_j) \leq 1 \times 1 \land \forall j \in J, \sigma_j \not\in \Delta\), we have \(t \sqsubseteq \Delta \land \exists u \in \pi_i(\sigma_j)\). This indicates that for an implicitly-typed expression \(\pi_i(a)\), if the inferred type for \(a\) and there exists \([\sigma_j]_{j \in J}\) such that \([\sigma_j]_{j \in J} \sqdess \Delta \land \exists u = \pi_i(\bigwedge_{j \in J} \sigma_j)\).

Let \(\Pi(\sigma_j)\) denote the set of such result types, that is,

\[
\Pi(\sigma_j) = \{u \mid [\sigma_j]_{j \in J} \sqdess \Delta \land \exists u = \pi_i(\bigwedge_{j \in J} \sigma_j)\}
\]

Formally, we have the following inference rule for projections:

\[
\begin{align*}
\frac{\Delta, \Gamma \vdash e : a \quad \forall j \in J, \sigma_j \not\in \Delta}{\Delta, \Gamma \vdash e(\sigma_j) : \pi_i(\sigma_j) \sqdess \Delta} \\
\end{align*}
\]

The following lemma tells us that \(\Pi(\sigma_j)\) is “morally” closed by intersection, in the sense that if we take two solutions in \(\Pi(\sigma_j)\), then we can take also their intersection as a solution, since there always exists in \(\Pi(\sigma_j)\) a solution at least as precise as their intersection.

Lemma B.7. Let \(t\) be a type and \(\Delta\) a set of type variables. If \(u_1 \in \Pi(\sigma_j)\) and \(u_2 \in \Pi(\sigma_j)\), then \(\exists u_0 \in \Pi(\sigma_j), u_0 \sqdess u_1 \land u_2\).
Proof. Let $[\sigma_{jk}]_{j \in J_k} \vdash t \sqsubseteq_\Delta 1 \times 1$ and $u_k = \pi_i(\bigwedge_{j \in J_k} t_{\sigma_{jk}})$ for $k = 1, 2$. Then $[\sigma_{jk}]_{j \in J_1 \cup J_2} \vdash t \sqsubseteq_\Delta 1 \times 1$. So $\pi_i(\bigwedge_{j \in J_1 \cup J_2} t_{\sigma_{jk}}) \in I_{\Delta}(t)$. Moreover, by Lemma C.6 in the companion paper [3], we have

$$\pi_i \left( \bigwedge_{j \in J_1 \cup J_2} t_{\sigma_{jk}} \right) \leq \pi_i \left( \bigwedge_{j_1 \in J_1} t_{\sigma_{jk_1}} \right) \wedge \pi_i \left( \bigwedge_{j_2 \in J_2} t_{\sigma_{jk_2}} \right) = u_1 \wedge u_2$$

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

$$\forall i \in I. \Delta \cup \text{var}(\bigwedge_{j \in J_i} (t_i \rightarrow s_i)) \vdash \Gamma, x : t_i \vdash e : s'_i \text{ and } s'_i \leq s_i$$

$$(\text{ALG-ABST})$$

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

$$\forall i \in I. \left\{ \begin{array}{l}
\Delta \vdash \Gamma, x : t_i \vdash e : s'_i \\
\forall j \in J_i, \sigma_j \not\in \Delta' \\
\forall j \in J_i, s'_i \sigma_j \leq s_i \\
\Delta' = \Delta \cup \text{var}(\bigwedge_{i \in I} (t_i \rightarrow s_i)) \\
\Delta \vdash \text{var}(\bigwedge_{i \in I} (t_i \rightarrow s_i))
\end{array} \right\}$$

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.

$$\forall i \in I. \left\{ \begin{array}{l}
\Delta \cup \text{var}(\bigwedge_{i \in I} (t_i \rightarrow s_i)) \\
\forall i \in I. \left\{ \begin{array}{l}
\Delta \vdash \Gamma, x : t_i \vdash a : s'_i \\
\Delta \vdash \text{var}(\bigwedge_{i \in I} (t_i \rightarrow s_i)) \\
\Delta \vdash \Delta \cup \text{var}(\bigwedge_{i \in I} (t_i \rightarrow s_i))
\end{array} \right\}
\end{array} \right\}$$

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

$$\Delta \vdash \Gamma \vdash e_1 : t \quad \forall j_1 \in J_1, \sigma_{j_1} \not\in \Delta \\
\Delta \vdash \Gamma \vdash e_2 : s \quad \forall j_2 \in J_2, \sigma_{j_2} \not\in \Delta \\
\bigwedge_{j_1 \in J_1} t_{\sigma_{j_1}} \in [0, 1] \\
\bigwedge_{j_2 \in J_2} s_{\sigma_{j_2}} \in \text{dom}(\bigwedge_{j_1 \in J_1} t_{\sigma_{j_1}})$$

$$\left\{ \begin{array}{l}
\Delta \vdash \Gamma \vdash (e_1(\sigma_{j_1}))_{j_1 \in J_1} : (\bigwedge_{j_1 \in J_1} t_{\sigma_{j_1}}) \\
\Delta \vdash \Gamma \vdash (e_2(\sigma_{j_2}))_{j_2 \in J_2} : (\bigwedge_{j_2 \in J_2} s_{\sigma_{j_2}})
\end{array} \right\}$$

Instantiation may be needed to bridge the gap between the (domain of the) function type and its argument (e.g., to apply $\lambda^a \rightarrow^a x.x$ to 42). The side conditions imply that $\sigma_{j_1} \mid_{j_1 \in J_1} \vdash t \sqsubseteq_\Delta 0 \rightarrow 1$ and $\sigma_{j_2} \mid_{j_2 \in J_2} \vdash s \sqsubseteq_\Delta \text{dom}(\bigwedge_{j_1 \in J_1} t_{\sigma_{j_1}})$. 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 $t \cdot_\Delta s$ denote the set of such result types, that is,

$$t \cdot_\Delta s = \left\{ u \mid \begin{array}{l}
[t_{\sigma_{j_1}}]_{j_1 \in J_1} \vdash t \sqsubseteq_\Delta 0 \rightarrow 1 \\
[s_{\sigma_{j_2}}]_{j_2 \in J_2} \vdash s \sqsubseteq_\Delta \text{dom}(\bigwedge_{j_1 \in J_1} t_{\sigma_{j_1}}) \\
u = \bigwedge_{j_1 \in J_1} t_{\sigma_{j_1}} \cdot \bigwedge_{j_2 \in J_2} s_{\sigma_{j_2}}
\end{array} \right\}$$

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

$$\Delta \vdash \Gamma \vdash a_1 : t \\
\Delta \vdash \Gamma \vdash a_2 : s \\
\Delta \vdash t \cdot_\Delta s$$

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

Proof. Let $u_k = \bigwedge_{k \in I_k} t_{\sigma_{j_k}} \cdot (\bigwedge_{j_k \in J_k} s_{\sigma_{j_k}})$ for $k = 1, 2$. According to Lemma C.18 in the companion paper [3], we have $[\bigwedge_{i \in I} t_{\sigma_{j_i}}] \cdot (\bigwedge_{j \in J} s_{\sigma_{j}}) \in t \cdot_\Delta s$ and $[\bigwedge_{i \in I} t_{\sigma_{j_i}}] \cdot (\bigwedge_{j \in J} s_{\sigma_{j}}) \leq \bigwedge_{k=1,2} (\bigwedge_{i \in I_k} t_{\sigma_{j_k}}) \cdot (\bigwedge_{j_k \in J_k} s_{\sigma_{j_k}}) = u_1 \wedge u_2$. 

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:

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad \forall j \in J, \sigma_j \not\Delta \\
\Delta \vdash \Gamma \vdash t' \sigma_j & : \bigwedge_{j \in J} t' \sigma_j \\
\Delta \vdash \Gamma \vdash (e \sigma_j) \in e ? : e_1 : s_1 & \quad \forall j \in J, \sigma_j \not\Delta \\
\end{align*}$$

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

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad t' \not\Delta \Delta \\
\Delta \vdash \Gamma \vdash (a \in t ? a_1 : a_2) : 0 & \\
\end{align*}$$

For case (ii), consider the following derivation:

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad \sigma_j \not\Delta \\
\Delta \vdash \Gamma \vdash e \sigma_j : \bigwedge_{j \in J} t' \sigma_j \\
\Delta \vdash \Gamma \vdash (e \sigma_j) \in e ? : e_1 : s_1 & \quad \forall j \in J, \sigma_j \not\Delta \\
\end{align*}$$

First, such a derivation can be rewritten as

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad \sigma_j \not\Delta \\
\Delta \vdash \Gamma \vdash e \sigma_j : \bigwedge_{j \in J} t' \sigma_j \\
\Delta \vdash \Gamma \vdash (e \sigma_j) \in e ? : e_1 : s_1 & \quad \forall j \in J, \sigma_j \not\Delta \\
\end{align*}$$

This indicates that it is equivalent to apply the substitutions $[\sigma_{j_1}]_{j \in J}$ to $e_1$ or to the whole type expression. Looking at the derivation for $e$, for the first branch to be selected we must have $t' \not\Delta \Delta$. Note that if $t' \not\Delta \not\Delta$, we would have $t' \not\Delta \Delta \not\Delta$ by Lemma B.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.

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad t' \not\Delta \not\Delta & \quad t' \not\Delta \not\Delta \\
\Delta \vdash \Gamma \vdash a : t' & \quad t' \not\Delta \not\Delta & \quad t' \not\Delta \not\Delta \\
\end{align*}$$

Case (iii) is similar to case (ii) where $t$ is replaced by $\not\Delta t$.

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

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad \forall j \in J, \sigma_j \not\Delta \\
\Delta \vdash \Gamma \vdash e \sigma_j : \bigwedge_{j \in J} t' \sigma_j \\
\end{align*}$$

Using $\alpha$-conversion if necessary, we can assume that the polymorphic type variables of $e_1$ and $e_2$ are distinct, and therefore we have $(\var{\sigma_1} \setminus \Delta) \cap (\var{\sigma_2} \setminus \Delta) = \emptyset$. According to Lemma B.5, we get $\forall j \in J, \sigma_j \not\Delta$. Let $\forall j \in J$.

We can rewrite this derivation as

$$\begin{align*}
\Delta \vdash \Gamma \vdash e : t' & \quad \forall j \in J, \sigma_j \not\Delta \\
\Delta \vdash \Gamma \vdash e \sigma_j : \bigwedge_{j \in J} t' \sigma_j \\
\end{align*}$$

As $\forall j \in J, \sigma_j \not\Delta$. Hence, we eliminate the substitutions that are applied to these two branches.

We now consider the case where the derivation tree which concerns $e$. With the specific set of substitutions $[\sigma_{j_1}]_{j \in J}$, we have $\bigwedge_{j \in J} t' \sigma_j \not\Delta$ and $\bigwedge_{j \in J} t' \sigma_j \not\Delta$, but it does not mean that we have $t' \not\Delta \Delta$ and
t' \not\subseteq_{\Delta} \neg t \text{ in general. If } t' \subseteq_{\Delta} t \text{ and } \text{or } t' \subseteq_{\Delta} \neg t \text{ 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 } \emptyset, s_1 \text{ or } s_2. \text{ We can then use subsumption to type the whole type-case expression with } s_1 \lor s_2. \text{ Otherwise, both branches are type-checked, and we deduce the corresponding inference rule as follows.}

\[
\Delta \vdash_\Gamma \Gamma' \vdash_\psi \frac{t' \not\subseteq_{\Delta} \neg t \quad \text{and} \quad t' \subseteq_{\Delta} t \quad \text{and} \quad \Delta \vdash_\psi \Gamma' a_1 : s_1 \quad \Delta \vdash_\psi \Gamma' a_2 : s_2}{a \in t \ ? \ a_1 : a_2} : s_1 \lor s_2
\]

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.

\[
\begin{align*}
\Delta \vdash_\Gamma \Gamma' \vdash c : b_c & \quad (\text{INF-CONST}) \quad \Delta \vdash_\Gamma \Gamma' \vdash x : \Gamma(x) \quad (\text{INF-VAR}) \\
\Delta \vdash_\Gamma \Gamma' \vdash a_1 : t_1 & \quad \Delta \vdash_\Gamma \Gamma' \vdash a_2 : t_2 \quad (\text{INF-PAIR}) \\
\Delta \vdash_\Gamma \Gamma' \vdash (a_1, a_2) : t_1 \times t_2 & \\
\Delta \vdash_\Gamma \Gamma' \vdash a_1 : t & \quad \Delta \vdash_\Gamma \Gamma' \vdash a_2 : s & \text{ u } & \in t \bullet \Delta s \quad (\text{INF-APPL}) \\
\forall i \in I. & \{ \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \} \vdash_\psi (x : t_i) \vdash \psi a : s_i' \\
& \bigwedge_{i \in I} t_i \rightarrow s_i \\
\Delta \vdash_\Gamma \Gamma' \vdash \lambda^{\wedge_{i \in I} t_i \rightarrow s_i} x.a : \bigwedge_{i \in I} t_i \rightarrow s_i \quad (\text{INF-ABSTR}) \\
\Delta \vdash_\Gamma \Gamma' \vdash a : t' & \quad t' \not\subseteq_{\Delta} 0 \quad (\text{INF-CASE-NONE}) \\
\Delta \vdash_\Gamma \Gamma' \vdash (a \in t \ ? \ a_1 : a_2) : 0 \\
\Delta \vdash_\Gamma \Gamma' \vdash a : t' & \quad t' \subseteq_{\Delta} t \quad t' \not\subseteq_{\Delta} \neg t \quad \Delta \vdash_\Gamma \Gamma' \vdash a_1 : s \quad (\text{INF-CASE-FST}) \\
\Delta \vdash_\Gamma \Gamma' \vdash (a \in t \ ? \ a_1 : a_2) : s \\
\Delta \vdash_\Gamma \Gamma' \vdash a : t' & \quad t' \subseteq_{\Delta} \neg t \quad t' \not\subseteq_{\Delta} t \quad \Delta \vdash_\Gamma \Gamma' \vdash a_2 : s \quad (\text{INF-CASE-SND}) \\
\Delta \vdash_\Gamma \Gamma' \vdash (a \in t \ ? \ a_1 : a_2) : s \\
\Delta \vdash_\Gamma \Gamma' \vdash a : t' & \{ t' \not\subseteq_{\Delta} \neg t \quad \text{and} \quad \Delta \vdash_\Gamma \Gamma' \vdash a_1 : s_1 \} \\
& \{ t' \not\subseteq_{\Delta} \neg t \quad \text{and} \quad \Delta \vdash_\Gamma \Gamma' \vdash a_2 : s_2 \} \quad (\text{INF-CASE-BOTH}) \\
\Delta \vdash_\Gamma \Gamma' \vdash (a \in t \ ? \ a_1 : a_2) : s_1 \lor s_2
\end{align*}
\]

Figure 5. Type-substitution inference rules

B.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 \(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} x.a\), different sets of substitutions may be constructed when type checking the body under the different hypotheses \(x : t_i\). For example, let \(a = \gamma^{\text{Int} \rightarrow \text{Int}}(\text{Bool} \rightarrow \text{Bool}) x.(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} y.x)\). When \(a\) is type-checked against \(\text{Int} \rightarrow \text{Int}\), that is, \(x\) is assumed to have type \(\text{Int}\), we infer the type substitution \(\{\text{Int}_{t_i}/a\}\) for \(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} y.x\). Similarly, we infer \(\{\text{Bool}_{t_i}/a\}\) for \(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} y.x\), when \(a\) is type-checked against \(\text{Bool} \rightarrow \text{Bool}\). 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 \(e \cap e'\) of expressions which is defined only for pair of expressions that have similar structure but different type substitutions. For example, the intersection of \(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} x.a\) and \(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} y.x\) will be \(\lambda^{\wedge_{i \in I} t_i \rightarrow s_i} y.x\).
Definition B.9. Let \( e, e' \in \mathcal{B}_0 \) be two expressions. Their intersection \( e \cap e' \) is defined by induction as:

\[
c \cap c = c
\]

\[
x \cap x = x
\]

\[
(e_1, e_2) \cap (e_1', e_2') = ((e_1 \cap e_1'), (e_2 \cap e_2'))
\]

\[
\pi_i(e) \cap \pi_i(e') = \pi_i(e \cap e')
\]

\[
e_1 \cap e_2 \cap (e_1', e_2') = (e_1 \cap e_1') (e_2 \cap e_2')
\]

\[
(\lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.e) \cap (\lambda^{\wedge \in \sigma_{l_2 \rightarrow s_2}} x.e') = \lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.(e \cap e')
\]

\[
(e_0 \in t_1 : e_2) \cap (e_0' \in t_1 : e_2') = e_0 \cap e_0' \in t_1 \cap e_1 \cap e_1' \cap e_2 \cap e_2'
\]

\[
(e_1[\sigma_{j_1 \in \mathcal{J}}] \cap (e_1'[\sigma_{j_1 \in \mathcal{J}}]) = (e_1 \cap e_1')[\sigma_{j_1 \in \mathcal{J}}]
\]

\[
e \cap (e_1'[\sigma_{j_1 \in \mathcal{J}}]) = (e \cap e_1')[\sigma_{j_1 \in \mathcal{J}}]
\]

\[
(\pi_i[\sigma_{j_1 \in \mathcal{J}}]) \cap e' = (\pi_i[\sigma_{j_1 \in \mathcal{J}}]) \cap e' 
\]

where \( \sigma_{id} \) is the identity type substitution and is undefined otherwise.

The intersection of the same constant or the same variable is the constant or the same variable. If \( e \) and \( e' \) have the same form, then their intersection is defined if the intersections of their corresponding sub-expressions are defined. In particular when \( e \) is of the form \( e_1[\sigma_{j_1 \in \mathcal{J}}] \) and \( e' \) is of the form \( e_1'[\sigma_{j_1 \in \mathcal{J}}] \), we merge the sets of substitutions \( \sigma_{j_1 \in \mathcal{J}} \) and \( \sigma_{j_1 \in \mathcal{J}}' \) into one set \( \sigma_{j_1 \in \mathcal{J} \cup \mathcal{J}'} \). Otherwise, \( e \) and \( e' \) have different forms. The only possible case where their intersection is well-defined is when they have similar structures but one with instantiations and the other without (i.e., \( e = e_1[\sigma_{j_1 \in \mathcal{J}}], e' \neq e_1'[\sigma_{j_1 \in \mathcal{J}}] \) or \( e \neq e_1[\sigma_{j_1 \in \mathcal{J}}], e' = e_1'[\sigma_{j_1 \in \mathcal{J}}] \)). In order not to lose any inferred information and be able to reuse the cases defined above, we add the identity substitution \( \sigma_{id} \) to the expression without substitutions (i.e., \( e[\sigma_{id}] \) or \( e'[\sigma_{id}] \)). Let us infer the substitutions for the abstraction \( \lambda^{l_{t_1 \rightarrow s_1}} x.(e_1 \cap e_2) \). Assume that we have inferred some substitutions for the body \( e \) under \( t_1 \rightarrow s_1 \) and \( t_2 \rightarrow s_2 \) respectively, yielding two explicitly-typed expressions \( e_1 \) and \( e_2[\sigma_{j_1 \in \mathcal{J}}] \). If we did not add the identity substitution \( \sigma_{id} \) for the intersection of \( e_1 \) and \( e_2[\sigma_{j_1 \in \mathcal{J}}] \), that is, \( e_1 \cap e_2[\sigma_{j_1 \in \mathcal{J}}] \) were \( e_1 \cap e_2[\sigma_{j_1 \in \mathcal{J}}] \) rather than \( (e_1 \cap e_2)[\sigma_{id}] \cup [\sigma_{j_1 \in \mathcal{J}}] \), then the substitutions we inferred under \( t_1 \rightarrow s_1 \) would be lost since they may be modified by \( [\sigma_{j_1 \in \mathcal{J}}] \).

Lemma B.10. Let \( e, e' \in \mathcal{B}_0 \) be two expressions. If \( \text{erase}(e) = \text{erase}(e') \), then \( e \cap e' \) exists and \( \text{erase}(e \cap e') = \text{erase}(e) = \text{erase}(e') \).

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

\[ e, e' \cap \mathcal{C} \text{ straightforward.} \]

\[ x, x' \cap \mathcal{C} \text{ straightforward.} \]

\[ (e_1, e_2), (e_1', e_2') \text{ have } \text{erase}(e_1) = \text{erase}(e_1'). \text{ By induction, } e_1 \cap e_1' \text{ exists and } \text{erase}(e_1) = \text{erase}(e_1'). \text{ Therefore } (e_1, e_2) \cap (e_1', e_2') \text{ exists and} \]

\[ \text{erase}((e_1, e_2) \cap (e_1', e_2')) = \text{erase}(((e_1 \cap e_1'), (e_2 \cap e_2')) = (\text{erase}(e_1 \cap e_1'), \text{erase}(e_2 \cap e_2')) = (\text{erase}(e_1), \text{erase}(e_2)) = \text{erase}(e_1, e_2) \]

Similarly, we also have \( \text{erase}((e_1, e_2) \cap (e_1', e_2')) = \text{erase}((e_1', e_2')). \)

\[ \pi_i(e), \pi_i(e') \text{ have } \text{erase}(e) = \text{erase}(e'). \text{ By induction, } e \cap e' \text{ exists and } \text{erase}(e \cap e') = \text{erase}(e) = \text{erase}(e') \text{. Therefore } \pi_i(e) \cap \pi_i(e') \text{ exists and} \]

\[ \text{erase}(\pi_i(e) \cap \pi_i(e')) = \text{erase}(\pi_i(e \cap e')) = \pi_i(\text{erase}(e \cap e')) = \pi_i(\text{erase}(e)) = \pi_i(\text{erase}(e)) = \text{erase}(\pi_i(e)) \]

Similarly, we also have \( \text{erase}(\pi_i(e) \cap \pi_i(e')) = \text{erase}(\pi_i(e')) \).

\[ (e_1 \cap e_2), (e_1', e_2') \text{ have } \text{erase}(e_1) = \text{erase}(e_1'). \text{ By induction, } e_1 \cap e_1' \text{ exists and } \text{erase}(e_1 \cap e_1') = \text{erase}(e_1) = \text{erase}(e_1'). \text{ Therefore } (e_1 \cap e_2) \cap (e_1', e_2') \text{ exists and} \]

\[ \text{erase}((e_1 \cap e_2) \cap (e_1', e_2')) = \text{erase}((e_1 \cap e_1')(e_2 \cap e_2')) = \text{erase}(e_1 \cap e_1') \text{erase}(e_2 \cap e_2') = \text{erase}(e_1 \cap e_2) = \text{erase}(e_1 \cap e_2) \]

Similarly, we also have \( \text{erase}((e_1 \cap e_2) \cap (e_1', e_2')) = \text{erase}(e_1'). \)

\[ \lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.e, \lambda^{\wedge \in \sigma_{l_2 \rightarrow s_2}} x.e' \text{ have } \text{erase}(e) = \text{erase}(e'). \text{ By induction, } e \cap e' \text{ exists and } \text{erase}(e \cap e') = \text{erase}(e) = \text{erase}(e'). \text{ Therefore } (\lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.e) \cap (\lambda^{\wedge \in \sigma_{l_2 \rightarrow s_2}} x.e') \text{ exists and} \]

\[ \text{erase}((\lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.e) \cap (\lambda^{\wedge \in \sigma_{l_2 \rightarrow s_2}} x.e')) = \text{erase}(\lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.(e \cap e')) = \lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.\text{erase}(e \cap e') = \lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.\text{erase}(e) = \text{erase}(\lambda^{\wedge \in \sigma_{l_1 \rightarrow s_1}} x.e) \]
Similarly, we also have

\[
\text{erase}(\lambda^x : \text{e} \rightarrow \lambda^x : \text{e}) = \text{erase}(\lambda^x : \text{e} \rightarrow \lambda^x : \text{e})
\]

\[
e_0 \in \text{e} \Rightarrow e_0 \in \text{e} \Rightarrow e_1 \Rightarrow e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5
\]

Since we have \(\text{erase}(e_1) = \text{erase}(e_2)\). By induction, \(e_1 \ni e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5\) and \(\text{erase}(e_1) = \text{erase}(e_2)\). Therefore, \(e_0 \in \text{e} \Rightarrow e_1 \Rightarrow e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5\).

Similarly, we also have

\[
\text{erase}(e_0 \in \text{e} \Rightarrow e_1 \Rightarrow e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5) = \text{erase}(e_0 \in \text{e} \Rightarrow e_1 \Rightarrow e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5)
\]

Similarly, we also have

\[
e_0 \in \text{e} \Rightarrow e_1 \Rightarrow e_2 \Rightarrow e_3 \Rightarrow e_4 \Rightarrow e_5
\]

Lemma B.11. Let \(c, e' \in \text{e} \Rightarrow \text{e}\) be two expressions. If \(\text{erase}(e') = \text{erase}(e)\), \(\Delta_1 \vdash e : t, \Delta' \vdash e' : t'\), \(e \ni e' \Rightarrow e \ni e' \Rightarrow t\), and \(\text{e} \ni e' \Rightarrow \text{e} \ni e' \Rightarrow t\).

Proof. According to Lemma B.10, \(e \ni e' \Rightarrow e \ni e' \Rightarrow t\) 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 (\text{instructer}) several times. The proof proceeds by induction on \(\Delta \vdash e : t\).

\begin{align*}
&\text{(const)}: \Delta_1 \vdash c : b_. \quad \text{As \(\text{erase}(e') = c, e'\) is either \(c\) or \(e_0\). If \(e' = c\), then \(e \ni e' \Rightarrow e \ni e' \Rightarrow t\), and the result follows straightforwardly. Otherwise, we have \(e \ni e' \Rightarrow e_0\). Since \(e' \ni \Delta\), we have \(e_0 \ni \Delta\). By (instructer), we have \(\Delta_1 \vdash e_0 : \sigma_0, \sigma_1 \ni \Delta, \sigma_2 \ni \Delta\), that is, \(\Delta_1 \vdash e_0 : \sigma_0, \sigma_1 \ni \Delta\).}

&\text{(var):} \quad \Delta_1 \vdash x : \Gamma(x). \quad \text{As \(\text{erase}(e') = x, e'\) is either \(x\) or \(e_0\). If \(e' = x\), then \(e \ni e' \Rightarrow e \ni e' \Rightarrow t\), and the result follows straightforwardly. Otherwise, we have \(e \ni e' \Rightarrow e_0\). Since \(e' \ni \Delta\), we have \(e_0 \ni \Delta\). By (instructer), we have \(\Delta_1 \vdash e_0 : \sigma_0, \sigma_1 \ni \Delta, \sigma_2 \ni \Delta\).}

&\text{(pair): consider the following derivation:}

&\begin{align*}
&\Delta_1 \vdash e_1 : t_1, \Delta_1 \vdash e_2 : t_2 \\
&\Delta_1 \vdash (e_1, e_2) : t_1 \times t_2
\end{align*}

As \(\text{erase}(e') = (\text{erase}(e_1), \text{erase}(e_2)), e'\) is either \((e_1, e_2)\) or \((e_0, e_2)\). By induction, we have \(\Delta_1 \vdash e_1 \ni e_1'), \Delta_1 \vdash e_2 \ni e_2').\) Then by (pair), we have \(\Delta_1 \vdash (e_1 \ni e_1', e_2 \ni e_2') : (t_1 \times t_2).\) If \(e' = (e_1, e_2), \) then \(e \ni e' \Rightarrow (e_1 \ni e_1', e_2 \ni e_2').\) So the result follows.

\text{Similarly, we have \(\Delta_1 \vdash e_1 \ni e_1', e_2 \ni e_2').\) By (instructer), we have \(\Delta_1 \vdash (e_1 \ni e_1', e_2 \ni e_2') : (t_1 \times t_2).\) Finally, by (subsum), we get \(\Delta_1 \vdash (e_1 \ni e_1', e_2 \ni e_2') : (t_1 \times t_2).\)

\text{(proj): consider the following derivation:}

&\begin{align*}
&\Delta_1 \vdash e_0 : t_1 \times t_2 \\
&\Delta_1 \vdash \pi_1(e_0) : t_1
\end{align*}

As \(\text{erase}(e') = \pi_1(\text{erase}(e_0)), e'\) is either \(\pi_1(e_0)\) or \(\pi_2(e_0)\). By induction, we have \(\Delta_1 \vdash e_0 \ni e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow t, \Delta_1 \vdash e_0 \ni e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow t\). If \(e' = \pi_1(e_0), \) then \(e \ni e' \Rightarrow (e_0 \ni e_0).\) So the result follows.

\text{Similarly, we have \(\Delta_1 \vdash e_0 \ni e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow t\). By (instructer), we have \(\Delta_1 \vdash (e_0 \ni e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow t) : t_1 \times t_2.\) Finally, by (subsum), we get \(\Delta_1 \vdash \pi_1(e_0 \ni e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow t) : t_1 \times t_2.\)

\text{(app): consider the following derivation:}

&\begin{align*}
&\Delta_1 \vdash e_1 : t \Rightarrow s, \Delta_1 \vdash e_2 : t \\
&\Delta_1 \vdash e_1 e_2 : s
\end{align*}
As \( \text{erase}(e') = \text{erase}(e_1) \text{erase}(e_2) \), \( e' \) is either \( e'_1 e'_2 \) or \( e'_1 e'_2 \lbrack \sigma_j \rbrack_{j \in J} \) such that \( \text{erase}(e'_1) = \text{erase}(e_1) \).

By induction, we have \( \Delta' ; \Gamma \vdash e_1 \cap e_2 : t \rightarrow s \) and \( \Delta'' ; \Gamma \vdash e_2 \cap e_2 : t \). Then by \( \text{(app)} \), we have \( \Delta' ; \Gamma \vdash (e_1 \cap e_2) (e_2 \cap e_2) : t \). So \( \Delta' ; \Gamma \vdash e_1 \cap e_2 : t \). Hence, \( \Delta' = \Delta'' \cap \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \).

As \( \text{erase}(e') = \lambda^{\Delta'} \Gamma \lbrack \sigma \rbrack_{J} \) \( \text{erase}(e_0) \), \( e' \) is either \( e_0 \in \sigma \) or \( e_2 \in \sigma \) such that \( \text{erase}(e_0) = \text{erase}(e_0) \).

By induction, we have \( \Delta' ; \Gamma \vdash e_0 \cap e_0 : t \) and \( \Delta' ; \Gamma \vdash e_2 \cap e_2 : s \). Hence, \( \Delta' ; \Gamma \vdash (e_0 \cap e_0) (e_2 \cap e_2) : t \). So the result follows.

Otherwise, \( \Delta' = \Delta'' \cap \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \).

By \( \text{(app)} \), we have \( \Delta'' ; \Gamma \vdash (e_1 \cap e_2) (e_2 \cap e_2) : t \). So the result follows.

Finally, by \( \text{(subsum)} \), we get \( \Delta'' ; \Gamma \vdash (\lambda^{\Delta'} \Gamma \lbrack \sigma \rbrack_{J}) \lbrack \sigma \rbrack_{J} \in J : \bigwedge_{i \in I} t_i \rightarrow s_i \).

\( \text{(case)} \): consider the following derivation:

\[
\begin{align*}
\Delta'' &\vdash e_1 : s \\
\Delta'' &\vdash e_2 : s
\end{align*}
\]

As \( \text{erase}(e') = \text{erase}(e_0) \in \Delta' \text{erase}(e_2) \).

By induction, we have \( \Delta ; \Gamma \vdash e_0 \cap e_0 : t \) and \( \Delta ; \Gamma \vdash e_2 \cap e_2 : s \). Then by \( \text{(app)} \), we have \( \Delta ; \Gamma \vdash (e_0 \cap e_0) (e_2 \cap e_2) : t \). Finally, by \( \text{(subsum)} \), we get \( \Delta ; \Gamma \vdash (e_0 \cap e_0) (e_2 \cap e_2) : t \).

\( \text{(instinter)} \): consider the following derivation:

\[
\begin{align*}
\Delta'' &\vdash e_0 : t \\
\sigma_j \vdash \Delta
\end{align*}
\]

As \( \text{erase}(e') = \text{erase}(e_0) \in \Delta' \text{erase}(e_2) \).

By induction, we have \( \Delta ; \Gamma \vdash e_0 \cap e_0 : t \). If \( e_0 \in e_0 \), then \( \sigma_j \vdash \Delta \).

Finally, by \( \text{(subsum)} \), we get \( \Delta ; \Gamma \vdash (e_0 \cap e_0) \lbrack \sigma \rbrack_{J} \in J : \bigwedge_{i \in I} t_i \rightarrow s_i \).

\( \text{Corollary B.12} \). Let \( e, e' \in \mathbb{E}_0 \) be two expressions. If \( \text{erase}(e) = \text{erase}(e') \), \( \Delta ; \Gamma \vdash e : t \), \( \Delta' ; \Gamma' \vdash e' : t \), \( e' \not\in \Delta' \) and \( e' \not\in \Delta' \), then

1. there exists \( s \) such that \( \Delta'' ; \Gamma \vdash e : s \) and \( s \leq t \).
2. there exists \( s' \) such that \( \Delta'' ; \Gamma' \vdash e : s' \) and \( s' \leq t' \).

\( \text{Proof} \). Immediate consequence of Lemma B.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 \( \lambda \)-abstractions are empty.
Theorem B.13 (Soundness). If \( \Delta \Gamma \vdash e : t \) and there exists an explicitly-typed expression \( c \in \mathcal{E}_0 \) such that \( \text{erase}(c) = a \) and \( \Delta \Gamma \vdash e : t \).

Proof. By induction on the derivation of \( \Delta \Gamma \vdash e : t \). 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

\[
\frac{\Delta \Gamma \vdash e_1 : t_1 \quad \Delta \Gamma \vdash e_2 : t_2}{\Delta \Gamma \vdash (e_1, e_2) : t_1 \times t_2}
\]

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

**(INF-PROJ):** consider the derivation

\[
\frac{\Delta \Gamma \vdash e : t \quad a \in \Pi(t)}{\Delta \Gamma \vdash \pi_a(e) : u}
\]

By induction, there exists an expression \( e \) such that \( \text{erase}(e) = \Delta \Gamma \vdash e : t \). Let \( u = \pi_a(\langle \bigwedge_{i \in I} \sigma_i \rangle) \). As \( \sigma_i \not\in \Delta \), by (ALG-INST), we have \( \Delta \Gamma \vdash e(\langle \bigwedge_{i \in I} \sigma_i \rangle) : \bigwedge_{i \in I} \tau_i \). Moreover, since \( \bigwedge_{i \in I} \tau_i \leq 1 \times 1 \), by (ALG-PROJ), we get \( \Delta \Gamma \vdash \pi_a(e(\langle \bigwedge_{i \in I} \sigma_i \rangle)) : \pi_a(\langle \bigwedge_{i \in I} \tau_i \rangle) \). Finally, according to Definition B.2, we have \( \text{erase}(\pi_a(e(\langle \bigwedge_{i \in I} \sigma_i \rangle))) = \pi_a(\text{erase}(e(\langle \bigwedge_{i \in I} \sigma_i \rangle))) = \pi_a(\text{erase}(e)) = \pi_a(a) \).

**(INF-APP):** consider the derivation

\[
\frac{\Delta \Gamma \vdash e_1 : t \quad \Delta \Gamma \vdash e_2 : s \quad u \in \Pi(t \bullet s)}{\Delta \Gamma \vdash e_1 e_2 : u}
\]

By induction, we have that \( (i) \) there exists an expression \( e_1 \) such that \( \text{erase}(e_1) = a_1 \) and \( \Delta \Gamma \vdash e_1 : t \) and \( (ii) \) there exists an expression \( e_2 \) such that \( \text{erase}(e_2) = a_2 \) and \( \Delta \Gamma \vdash e_2 : s \). Let \( u = \langle \bigwedge_{i \in I} \sigma_i \rangle \). As \( \sigma_i \not\in \Delta \) for \( h \in I \cup J \), applying (ALG-INST), we get \( \Delta \Gamma \vdash e_1(\langle \bigwedge_{i \in I} \sigma_i \rangle) : \bigwedge_{i \in I} \tau_i \) and \( \Delta \Gamma \vdash e_2(\langle \bigwedge_{i \in J} \sigma_i \rangle) : \bigwedge_{i \in J} \sigma_i \). Then by (ALG-APPL), we have \( \Delta \Gamma \vdash (e_1(\langle \bigwedge_{i \in I} \sigma_i \rangle))(e_2(\langle \bigwedge_{i \in J} \sigma_i \rangle)) : \bigwedge_{i \in I} \tau_i \). Furthermore, according to Definition B.2, we have \( \text{erase}(e_1(\langle \bigwedge_{i \in I} \sigma_i \rangle))(e_2(\langle \bigwedge_{i \in J} \sigma_i \rangle)) = \text{erase}(e_1)\text{erase}(e_2) = a_1a_2 \).

**(INF-ABST):** consider the derivation

\[
\frac{\forall i \in I. \left( \begin{array}{c} \Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle) \vdash_{\text{VAR}} (x : t_i) \vdash_{\text{VAR}} a : s' \\ s_i' \subseteq \Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle) s_i \end{array} \right) \quad \Delta \vdash e : t \cdot x}{\Delta \vdash \lambda^{\Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle)} x : s_i' \cdot \lambda^{\Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle)} x \cdot a : \bigwedge_{i \in I} t_i \to s_i}
\]

Let \( \Delta' = \Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle) \) and \( \langle \sigma_i \rangle_{J \in I} \vdash s'_i \subseteq \Delta' \subseteq s_i \). By induction, there exists an expression \( e_i \) such that \( \text{erase}(e_i) = a_i \) and \( \Delta \vdash e_i : s_i \) for all \( i \in I \). Since \( \sigma_i \not\in \Delta \), by (ALG-INST), we have \( \Delta \vdash e_i(\langle \bigwedge_{i \in J} \sigma_i \rangle) : \bigwedge_{i \in J} \tau_i \). Clearly, \( e_i(\langle \bigwedge_{i \in J} \sigma_i \rangle) \vdash e_i' \) and \( \text{erase}(e_i(\langle \bigwedge_{i \in J} \sigma_i \rangle)) = \text{erase}(e_i) = a_i \). Then by Lemma B.10, the intersection \( \bigcap_{i \in I} e_i(\langle \bigwedge_{i \in J} \sigma_i \rangle) \) exists and we have \( \text{erase}(\bigcap_{i \in I} e_i(\langle \bigwedge_{i \in J} \sigma_i \rangle)) = a_i \) for any non-empty \( I' \subseteq I \). Let \( e = \bigcap_{i \in I} e_i(\langle \bigwedge_{i \in J} \sigma_i \rangle) \) exists.

According to Corollary B.12, there exists a type \( t_i' \) such that \( \Delta' \vdash_{\text{VAR}} (x : t_i) \vdash e : t_i' \) and \( t_i' \leq \bigwedge_{i \in I} s_i' \sigma_i \). Moreover, since \( t_i' \leq \bigwedge_{i \in J} s_i' \sigma_i \leq s_i \), by (ALG-ABST), we have \( \Delta \vdash e : \bigwedge_{i \in I} t_i \to s_i \cdot x.e : \bigwedge_{i \in I} t_i \to s_i \). Finally, according to Definition B.2, we have \( \text{erase}(\lambda^{\Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle)})x.e = \lambda^{\Delta \cup \text{var}(\langle \bigwedge_{i \in I} t_i \to s_i \rangle)}x.e \).

**(INF-CASE-NONE):** consider the derivation

\[
\frac{\Delta \vdash e : t' \quad t' \subseteq \Delta \emptyset}{\Delta \vdash e : t' \quad \text{Not} \quad a \vdash e_1 : 0 \quad a \vdash e_2 : 0}
\]

By induction, there exists an expression \( e \) such that \( \text{erase}(e) = a \) and \( \Delta \vdash e : t' \). Let \( \langle \sigma_i \rangle_{I \vdash t' \subseteq \Delta \emptyset} \). Since \( \sigma_i \not\in \Delta \), by (ALG-INST), we have \( \Delta \vdash e_i : \bigwedge_{i \in I} t_i \sigma_i \). Let \( e_1 \\ e_2 \) be two expressions such that \( \text{erase}(e_1) = a_1 \) and \( \text{erase}(e_2) = a_2 \). Then we have \( \text{erase}(e_i) = a \) and \( a_i \vdash e_i \). Moreover, since \( \bigwedge_{i \in I} t_i \sigma_i \leq 0 \), by (ALG-CASE-NONE), we have \( \Delta \vdash \text{erase}(e_i) \).
Theorem B.14. Consider the derivation

\[
\frac{\Delta \Gamma \vdash \varphi \quad \varphi \equiv \Delta \Gamma \vdash \phi \quad \Delta \Gamma \vdash \phi \quad \Delta \Gamma \vdash \phi}{\Delta \Gamma \vdash \phi} \quad \Delta \Gamma \vdash \phi \quad \Delta \Gamma \vdash \phi \quad \Delta \Gamma \vdash \phi
\]

By induction, there exist \( e, e_1 \) such that \( \text{erase}(e) = a, \text{erase}(e_1) = a_1, \Delta \Gamma \vdash e : t' \), and \( \Delta \Gamma \vdash e_1 : s_1 \). Let \( \sigma_{i_1} \in I \) \( i_1 \vdash t \Delta \). Since \( \sigma_{i_1} \not\in \Delta \), applying (ALG-INST), we get \( \Delta \Gamma \vdash e[\sigma_{i_1}] \in I_i \vdash t_{i_1} \sigma_{i_1} \). Let \( e_2 \) be an expression such that \( \text{erase}(e_2) = a_2 \). Then we have

\[\text{erase}((e[\sigma_{i_1}] \in I_i \vdash t \sigma_{i_1}))(e \in t ? e_1 : e_2) = (a \in t ? a_1 : a_2) .\]

Finally, since \( \Lambda_{i_1 \in I_i} t_{i_1} \sigma_{i_1} \leq t \), by (ALG-TERM), we have

\[\Delta \Gamma \vdash ((e[\sigma_{i_1}] \in I_i \vdash t \sigma_{i_1}))(e \in t ? e_1 : e_2) : s .\]

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

**Theorem B.14 (Completeness).** Let \( e \in \mathcal{E} \) be an explicitly-typed expression. If \( \Delta \Gamma \vdash e : t \), then there exists a type \( t' \) such that \( \Delta \Gamma \vdash \text{erase}(e) : t' \) and \( t' \equiv \Delta \).

**Proof.** By induction on the typing derivation of \( \Delta \Gamma \vdash e : t \). We proceed by a case analysis on the last rule used in the derivation.

1. (ALG-CONST): take \( t' \) as \( b_e \).
2. (ALG-VAR): take \( t' \) as \( \Gamma(x) \).
3. (ALG-PAIR): consider the derivation

\[
\frac{\Delta \Gamma \vdash e_1 \vdash t_1 \quad \Delta \Gamma \vdash e_2 \vdash t_2}{\Delta \Gamma \vdash e_1 \times e_2 \vdash t_1 \times t_2}
\]

Applying the induction hypothesis twice, we have

\[
\exists t', \Delta \Gamma \vdash \text{erase}(e_1) : t'_1 \text{ and } t'_1 \equiv \Delta \text{ and } t'_1 \leq t_1,
\]

\[
\exists t', \Delta \Gamma \vdash \text{erase}(e_2) : t'_2 \text{ and } t'_2 \equiv \Delta \text{ and } t'_2 \leq t_2 .
\]

Then by (INF-PAIR), we have \( \Delta \Gamma \vdash (\text{erase}(e_1), \text{erase}(e_2)) : t'_1 \times t'_2 \), that is, \( \Delta \Gamma \vdash \text{erase}(e_1), \text{erase}(e_2)) : t'_1 \times t'_2 \). Finally, Applying Lemma B.4, we have \( (t'_1 \times t'_2) \equiv t_1 \times t_2 \).

4. (ALG-PROJ): consider the derivation

\[
\frac{\Delta \Gamma \vdash e : t \quad t \leq 1 \times 1}{\Delta \Gamma \vdash \pi_i(e) : \pi_i(t)}
\]

By induction, we have

\[
\exists t', [\sigma_k]_{k < K}, \Delta \Gamma \vdash \text{erase}(e) : t' \text{ and } [\sigma_k]_{k < K} \vdash t' \equiv \Delta .
\]

It is clear that \( \Lambda_{k < K} \sigma_k \leq 1 \times 1 \). So \( \pi_i(\Lambda_{k < K} \sigma_k) \in \Pi_1(t') \). Then by (INF-PROJ), we have \( \Delta \Gamma \vdash \pi_i(\text{erase}(e)) : \pi_i(\Lambda_{k < K} \sigma_k) \), that is, \( \Delta \Gamma \vdash \text{erase}(\pi_i(e)) : \pi_i(\Lambda_{k < K} \sigma_k) \). According to Lemma C.5 in the companion paper [3], \( t \leq \pi_1(t), \pi_2(t) \). Then \( \Lambda_{k < K} \sigma_k \leq (\pi_1(t), \pi_2(t)) \). Finally, applying Lemma C.5 again, we get \( \pi_i(\Lambda_{k < K} \sigma_k) \leq \pi_i(t) \) and \( a \text{ for } \pi_i(\Lambda_{k < K} \sigma_k) \equiv \pi_i(t) \).

5. (ALG-APPL): consider the derivation

\[
\frac{\Delta \Gamma \vdash e_1 \vdash t \quad \Delta \Gamma \vdash e_2 \vdash s \quad t \leq 0 \rightarrow 1 \text{ and } s \equiv \text{dom}(t)}{\Delta \Gamma \vdash e_1 e_2 \vdash t \cdot s}
\]
Applying the induction hypothesis twice, we have
\[
\exists t', \sigma_k^1 \in K_1, \Delta \vdash e : t' \quad \text{and} \quad [\sigma_k^1]_{k \in K_1} \models t' \sqsubseteq t, \\
\exists t', \sigma_k^2 \in K_2, \Delta \vdash e : t' \quad \text{and} \quad [\sigma_k^2]_{k \in K_2} \models t' \sqsubseteq s.
\]

It is clear that \(\Delta \vdash e : t\), that is, \(\Delta \vdash e : t\) is a function type. So we get \(\text{dom}(\Delta) \leq \text{dom}(\Delta')\). Then we have \(\Delta \vdash e : t\) \(\sqsubseteq \Delta \vdash e : t\). Therefore, \(\Delta \vdash e : t\) \(\sqsubseteq \Delta \vdash e : t\). Then applying (INF-APPL), we have
\[
\Delta \vdash e : t\quad (\Delta \vdash e : t),
\]

that is, \(\Delta \vdash e : t\). Moreover, as \(\Delta \vdash e : t\) \(\sqsubseteq \Delta \vdash e : t\), \(t \cdot (\Delta \vdash e : t) \sqsubseteq \Delta \vdash e : t\). Thus, \((\Delta \vdash e : t) \sqsubseteq \Delta \vdash e : t\) \(\sqsubseteq \Delta \vdash e : t\).

\textsc{(Alg-Abstr0):} consider the derivation
\[
\forall i \in I. \Delta \cup \text{var}(\Delta_{i \in I} t_i \rightarrow s_i) \vdash \Gamma, (x : t_i) \vdash e : s_i' \quad \text{and} \quad s_i' \sqsubseteq s_i.
\]

\textsc{(Alg-Case-None):} consider the derivation
\[
\Delta \vdash e : 0 \quad \Delta \vdash (e \in \Gamma ? e_1 : e_2) : 0.
\]

By induction, we have
\[
\exists t_0', \Delta \vdash e : t_0' \quad \text{and} \quad t_0' \sqsubseteq \Delta \vdash 0.
\]

By (INF-Case-None), we have \(\Delta \vdash (e \in \Gamma ? e_1 : e_2) : 0\), that is, \(\Delta \vdash e : e_1 \quad \text{and} \quad e_1 \sqsubseteq \Delta \vdash e_2\). By (Alg-Case-Fst), we have
\[
\Delta \vdash e : t' \quad t' \sqsubseteq t \quad \Delta \vdash e : s_1.
\]

Applying the induction hypothesis twice, we have
\[
\exists t_0', \Delta \vdash e : t_0' \quad \text{and} \quad t_0' \sqsubseteq \Delta \vdash t'.
\]

Clearly, we have \(t_0' \sqsubseteq t\). If \(t_0' \sqsubseteq t\), then by Lemma B.4, we have \(t_0' \leq \Delta \vdash 0\). By (Inf-Case-None), we get
\[
\Delta \vdash e : t' \quad \text{that is,} \quad \Delta \vdash e : t' \quad \text{and} \quad t' \sqsubseteq \Delta \vdash 0. \]

Otherwise, by (Inf-Case-Fst), we have
\[
\Delta \vdash e : t' \quad \text{that is,} \quad \Delta \vdash e : t' \quad \text{and} \quad t' \sqsubseteq \Delta \vdash 0.
\]

The result follows as well.

\textsc{(Alg-Case-Snd):} similar to the case of (Alg-Case-Fst).

\textsc{(Alg-Case-Both):} consider the derivation
\[
\begin{cases} 
\Delta \vdash e : t' \quad t' \leq t \quad \text{and} \quad \Delta \vdash e : s_1 \\
\Delta \vdash e : s_2
\end{cases}
\]

By induction, we have
\[
\exists t_0', \Delta \vdash e : t_0' \quad \text{and} \quad t_0' \sqsubseteq \Delta \vdash t'.
\]

\[
\exists t_1', \Delta \vdash e : t_1' \quad t_1' \sqsubseteq \Delta \vdash s_1,
\]

\[
\exists t_2', \Delta \vdash e : t_2' \quad t_2' \sqsubseteq \Delta \vdash s_2.
\]
If \( t'_0 \subseteq \Delta \), then by (INF-CASE-NONE), we get
\[
\Delta_1 \Gamma \vdash e \triangleleft \begin{cases} \text{erase}(e) \in \tau \Rightarrow \text{erase}(e_1) : \text{erase}(e_2) & \text{if } \tau \end{cases},
\]
that is, \( \Delta_1 \Gamma \vdash \text{erase}(e) \in \tau ? e_1 : e_2 : 0 \). Clearly, we have \( 0 \subseteq \Delta \) s.t. \( s_1 \lor s_2 \).
If \( t'_0 \nsubseteq \Delta \), then by (INF-CASE-FST), we get
\[
\Delta_1 \Gamma \vdash \text{erase}(e) \in \tau ? e_1 : e_2 : t'_1,
\]
that is, \( \Delta_1 \Gamma \vdash \text{erase}(e) \in \tau \) ? \( e_1 : e_2 : t'_1 \). Moreover, it is clear that \( t'_1 \subseteq \Delta \) s.t. \( s_1 \lor s_2 \), the result follows as well. Similarly for \( t'_0 \nsubseteq \Delta \) \( \neg t \).
Otherwise, by (INF-CASE-BOTH), we have
\[
\Delta_1 \Gamma \vdash \text{erase}(e) \in \tau ? e_1 : e_2 : t'_1 \lor t'_2,
\]
that is, \( \Delta_1 \Gamma \vdash \text{erase}(e) \in \tau ? e_1 : e_2 : t'_1 \lor t'_2 \). Using \( \alpha \)-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., \( \text{var}(t'_1) \setminus \Delta \) \( \cap \text{var}(t'_2) \setminus \Delta = \emptyset \). Then applying Lemma B.5, we have \( t'_1 \lor t'_2 \subseteq \Delta_1 \lor t \lor t_2 \).

(Alg-Inst): consider the derivation
\[
\Delta_1 \Gamma \vdash e : t \quad \forall j \in J, \sigma_j \not\subseteq \Delta \quad |J| > 0
\]
\[
\Delta_1 \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j
\]
By induction, we have
\[
\exists \Delta, [\sigma_k]_{k \in K}, \Delta_1 \Gamma \vdash \text{erase}(e) : t' \land [\sigma_k]_{k \in K} \vdash t' \subseteq \Delta.
\]
Since \( \text{erase}(e[\sigma_j]_{j \in J}) = \text{erase}(e) \), we have \( \Delta_1 \Gamma \vdash \text{erase}(e[\sigma_j]_{j \in J}) : t' \). As \( \bigwedge_{k \in K} t' \sigma_k \leq t \), we have \( \bigwedge_{j \in J} (\bigwedge_{k \in K} \sigma_j \sigma_k) \leq \bigwedge_{j \in J} t \sigma_j \). Therefore, it is clear that \( \sigma_j \circ \sigma_k \not\subseteq \Delta \). Therefore, we get \( t' \subseteq \Delta \bigwedge_{j \in J} t \sigma_j \).

The inference system is syntax directed and describes an algorithm that is parametric in the decision procedures for \( \subseteq \Delta \), \( \Pi_\Delta(t) \) and \( t \bullet s \). The problem of deciding them is tackling in Section C.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—indeed, independently from the fact that we started from an explicitly typed expression or not—and we cannot avoid it. The (big-step) semantics for relabeling at run-time is an essential feature—independently from the fact that we started from an implicitly typed expression or not— and we cannot avoid it.

\[\Delta_1 \Gamma \vdash \text{erase}(e) \in \tau ? e_1 : e_2 : t'_1 \lor t'_2,\]
that is, \( \Delta_1 \Gamma \vdash \text{erase}(e) \in \tau \) ? \( e_1 : e_2 : t'_1 \lor t'_2 \). Using \( \alpha \)-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., \( \text{var}(t'_1) \setminus \Delta \) \( \cap \text{var}(t'_2) \setminus \Delta = \emptyset \). Then applying Lemma B.5, we have \( t'_1 \lor t'_2 \subseteq \Delta_1 \lor t \lor t_2 \).

B.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:
\[
\Delta_1 \Gamma \vdash a : t \\
\Delta_1 \Gamma \vdash a : u \\
\end{array}
\]
We now look at the type inference rules for the type case $a \in t \ ? t_1 : a_2$. 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 $((\lambda^{\alpha \rightarrow \alpha} \ x \ x) \in \text{Int} \rightarrow \text{Int}) \ ? \ true : false$, 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 $\text{Int} \rightarrow \text{Int}$ (and thus make the test succeed) rather than $\text{Bool} \rightarrow \text{Bool}$ (and thus make the test fail). If we decide not to infer types for polymorphic arguments of type-case expression, then since $\alpha \rightarrow \alpha$ is not a subtype of $\text{Int} \rightarrow \text{Int}$ (we have $\alpha \rightarrow \alpha \not\subseteq \text{Int} \rightarrow \text{Int}$) the expression evaluates to $false$. With this choice, we can merge the different inference rules into the following one.

$$
\Delta \vdash \Gamma \vdash a : t', t_1 = t' \land t \quad t_2 = t' \land \neg t
$$

$$
\Delta \vdash \Gamma \vdash a_1 : s_1
$$

$$
\Delta \vdash \Gamma \vdash\ (a \in t ? a_1 : a_2) : \bigvee_{t_1 \not\equiv \mathbf{0}} (\text{INF-CASE'})
$$

Finally, consider the inference rule for abstractions:

$$
\forall i \in I. \left\{ \begin{array}{l}
\Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \Gamma, (x : t_i) \vdash a : s'_i
\
s'_i \subseteq \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i)
\end{array} \right\}
$$

$$
\Delta \vdash \Gamma \vdash \lambda^{\bigwedge_{i \in I} t_i \rightarrow s_i} x : a : \bigwedge_{i \in I} t_i \rightarrow s_i
$$

We verify that the abstraction can be typed with each arrow type $t_i \rightarrow s_i$ 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

$$
\lambda^{\text{Bool} \rightarrow (\text{Int} \rightarrow \text{Int})} x.x \in \text{true} ? ((\lambda^{\alpha \rightarrow \alpha} y : y) : ((\lambda^{\alpha \rightarrow \alpha} y : y) z : z) y)
$$

is not well-typed while

$$
\lambda^{\text{Bool} \rightarrow (\alpha \rightarrow \alpha) \rightarrow \text{true} ? ((\lambda^{\alpha \rightarrow \alpha} y : y) : ((\lambda^{\alpha \rightarrow \alpha} y : y) z : z) y)
$$

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

$$
\forall i \in I. \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \Gamma, (x : t_i) \vdash a : s'_i \text{ and } s'_i \subseteq s_i
$$

$$
\Delta \vdash \Gamma \vdash \lambda^{\bigwedge_{i \in I} t_i \rightarrow s_i} x : a : \bigwedge_{i \in I} t_i \rightarrow s_i
$$

(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.

**Theorem B.15.** If $\Gamma \vdash a : t$, then there exists an expression $e \in \mathcal{E}_0$ such that $\text{erase}(e) = a$ and $\Gamma \vdash e : t$.

**Proof.** Similar to the proof of Theorem B.13.

\[\blacklozenge\]

C. 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 $\bigwedge_{i \in I} \sigma_I \subseteq \bigvee_{j \in J} \sigma_J$. 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 $J$.

C.1 Type Tallying Problem

Given a finite set $C$ of pairs of types and a finite set $\Delta$ of type variables, the tallying problem for $C$ and $\Delta$ consists in verifying whether there exists a substitution $\sigma$ such that $\sigma \not\subseteq \Delta$ 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 C.1 (Constraints).** A constraint $(t, c, s)$ is a triple belonging to $\mathcal{C} \times \{\leq, \geq\} \times \mathcal{C}$. Let $\mathcal{C}$ denote the set of all constraints. Given a constraint-set $C \subseteq \mathcal{C}$, the set of type variables occurring in $C$ is defined as

$$
\text{var}(C) = \bigcup_{(t, c, s) \in C} \text{var}(t) \cup \text{var}(s)
$$
**Definition C.2 (Normalized constraint).** A constraint \((t, c, s)\) is said to be normalized if \(t\) is a type variable. A constraint-set \(C \subseteq \mathcal{C}\) 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 \(\text{dom}(C) = \{\alpha \mid \exists c, s, (\alpha, c, s) \in C\}\).

**Definition C.3 (Constraint solution).** Let \(C \subseteq \mathcal{C}\) be a constraint-set. A solution to \(C\) is a substitution \(\sigma\) such that
\[
\forall (t, \leq, s) \in C. \; \sigma \leq s \text{ holds} \quad \text{and} \quad \forall (t, \geq, s) \in C. \; s \leq \sigma \text{ holds}.
\]
If \(\sigma\) is a solution to \(C\), we write \(\sigma \models C\).

**Definition C.4.** Given two sets of constraint-sets \(\mathcal{R}_1, \mathcal{R}_2 \subseteq \mathcal{P}(\mathcal{C})\), we define their union as
\[
\mathcal{R}_1 \cup \mathcal{R}_2 : = \mathcal{R}_1 \cup \mathcal{R}_2
\]
and their intersection as
\[
\mathcal{R}_1 \cap \mathcal{R}_2 : = \{C_1 \cup C_2 \mid C_1 \in \mathcal{R}_1, C_2 \in \mathcal{R}_2\}
\]
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 \(c_1 = ((\alpha \times \alpha) \leq ((\text{Int} \times 1) \times (1 \times \text{Int}))\) can be normalized into the set \(\mathcal{S}_1 = \{(\{\alpha, \leq, 0\} \cup (\{\alpha, \leq, (\text{Int} \times 1)\})), (\alpha, \leq, (1 \times \text{Int})\}\). Another example is the constraint \(c_2 = ((\beta \times \alpha) \leq (\text{Int} \times \text{Int}), \leq, \alpha \rightarrow \beta)\), which is equivalent to the following set of normalized constraint-sets \(\mathcal{S}_2 = \{(\{\alpha, \leq, 0\} \cup (\{\alpha, \leq, (\beta \times \beta)\})), (\alpha, \leq, (\text{Int} \times \text{Int}))\}\). 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 \(\mathcal{S}\) of \(\{c_1, c_2\}\) is \(\mathcal{S}_1 \cap \mathcal{S}_2\). It is easy to see that the constraint-set \(C_1 = \{(\alpha, \leq, (\text{Int} \times 1)) \cup (\{\alpha, \leq, (1 \times \text{Int})\})), (\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\text{Int} \times \text{Int}))\}\) in \(\mathcal{S}\) (see Definition C.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 \((\alpha, \geq, t)\) (resp. \((\alpha, \leq, t)\)), which give different lower bounds (resp. upper bounds) for \(\alpha\). 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 \(C_2 = \{(\{\alpha, \leq, (\text{Int} \times 1) \cup (1 \times \text{Int})\} \cup (\{\alpha, \leq, (\beta \times \beta)\})), (\alpha, \geq, (\text{Int} \times \text{Int}))\}\), which is equivalent to \(\{(\alpha, \leq, (\text{Int} \times \beta \times \text{Int} \times \beta)), (\alpha, \geq, (\text{Int} \times \text{Int}))\}\).

**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
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 any set of constraints is satisfied for all assignments of \( \alpha \) included between \( s \) and \( t \) if and only if the same set in which we replace \( \alpha \) by \((s \lor \alpha') \land t^{10}\) is satisfied for all possible assignments of \( \alpha' \) (with \( \alpha' \) 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 \( E \):

\[
\alpha = ((\text{Int} \times \text{Int}) \lor \alpha') \land (\text{Int} \land \beta \land \text{Int} \land \beta)
\]

\[
\beta = \text{Int} \lor \beta'
\]

where \( \alpha', \beta' \) are fresh type variables.

Step 3.2: extract a substitution from each equation system.

Finally, using the Courcelle’s work on infinite trees \[7\], 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 \( S, \text{Int} \lor \beta' \beta \), 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.

C.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, \( (\alpha \lor \beta, \geq, 0) \) holds if and only if \( (\alpha, \geq, 0) \lor (\beta, \geq, 0) \) holds; therefore the constraint \( (\alpha \lor \beta, \geq, 0) \) is equivalent to the normalized constraint-set \( \{\alpha, \geq, 0\}, \{\beta, \geq, 0\} \}. \) Consequently, the normalization of a constraint-set \( C \) yields a set \( \mathcal{F} \) of normalized constraint-sets.

Several normalized sets may be suitable replacements for a given constraint; for example, \( \{\alpha, \leq, \beta \lor v_1\}, \{\beta, \leq, \alpha \lor v_2\} \} \) and \( \{\alpha, \leq, \beta \lor v_1\}, \{\alpha, \geq, \beta \lor v_2\} \} \) 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 C.1.3 for more details). Concretely, \( \{\alpha, \leq, \beta \lor v_1\}, \{\beta, \leq, \alpha \lor v_2\} \} \) generates the equation system \( \{\alpha = \alpha' \land (\beta \lor v_1), \beta = \beta' \land (\alpha \lor v_2)\} \), which in turn gives the substitution \( \sigma_1 \) such that

\[
\sigma_1(\alpha) = \mu x. ((\alpha' \land \beta' \land x) \lor (\alpha' \land \beta' \land v_2) \lor (\alpha' \land v_1))
\]

\[
\sigma_1(\beta) = \mu x. ((\beta' \land \alpha' \land x) \lor (\beta' \land \alpha' \land v_1) \lor (\beta' \land v_2))
\]

where \( \alpha' \) and \( \beta' \) are fresh type variables and we used the \( \mu \) 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 \( \{\alpha, \leq, \beta \lor v_1\}, \{\alpha, \geq, \beta \lor v_2\} \} \) is \( \alpha = ((\beta \land v_2) \lor \alpha') \land (\beta \lor v_1), \) and the corresponding substitution is \( \sigma_2 = \{((\beta \land v_2) \lor \alpha') \land (\beta \lor v_1) / \alpha\} \), which is valid since it maps the type variable \( \alpha \) 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 \( B_i \in \{\land, \lor\} \) for all \( i \) and \( n \geq 0 \)) in the equation system built from the normalized constraint-set. This chain implies the equation \( \alpha_0 = \alpha_0 B^{l'} \) for some \( B \in \{\land, \lor\} \) and \( l' \), and the corresponding solution for \( \alpha_0 \) 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 \( (\alpha, c, t) \) such that the order of \( \alpha \) 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 C.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 \( \Sigma \vdash_{\mathcal{F}} C \sim \mathcal{F} \), which states that under the environment \( \Sigma \) (which, informally, contains the types that have already been processed at this point), \( C \) is normalized to \( \mathcal{F} \). The judgement is derived according the rules of Figure 7. These rules describe the same algorithm
as the function norm given in Figure 3 (i.e., $\Sigma \vdash_{\mathcal{A}} \{ (t, \leq, 0) \} \leadsto \text{norm}(t, \Sigma)$ 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.

![Normalization rules](image)

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 $(\tau, \leq, 0)$ where $\tau$ 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 $(t \land \neg t', \leq, 0)$, and finally the last rule puts $t \land \neg t'$ in disjunctive normal form. Such a type $\tau$ is the type to be normalized. If $\tau$ 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 $(\tau, \leq, 0)$, where $\tau$ 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 $\tau$ has already been normalized (i.e., it belongs to $\Sigma$), then it is not processed again (rule (NHYP)). Otherwise, we memoize it and then process for single normal forms $\Sigma \vdash_{\mathcal{A}} C \leadsto \mathcal{F}$ (rule (NASSUM)). Note that switching from $\Sigma \vdash_{\mathcal{A}} C \leadsto \mathcal{F}$ to $\Sigma \vdash_{\mathcal{A}} C \leadsto \mathcal{F}$ 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 $(\tau, \leq, \emptyset)$ where $\tau$ is a single normal form and contains no top-level type variables. Thereby $\tau$ should be an intersection of atoms with the same constructor. If $\tau$ is an intersection of basic types, normalizing is equivalent to checking whether $\tau$ 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 $\tau$ is an intersection of products, the rule (NPROD) decomposes $\tau$ into several candidate types (following Lemma 3.11 in [4]), which are to be further normalized. The case when $\tau$ is an intersection of arrows (rule (NARROW)) is treated similarly. Note that, in the last two rules, we switch from $\Sigma \vdash_{n,\gamma} C \rightsquigarrow S$ back to $\Sigma \vdash_{n,\gamma} C \rightsquigarrow S$ in the premises to ensure termination.

If $\emptyset \vdash_{n,\gamma} C \rightsquigarrow S$, then $S$ 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 $\leq^{11}$ (i.e., such that $\bigcup_{n \in \mathbb{N}} \leq_n = \leq$) and a family of satisfaction predicates $\vdash_n$, that layer $\vdash$ (i.e., such that $\bigcup_{n \in \mathbb{N}} \vdash_n = \vdash$), which are defined as follows.

**Definition C.6.** Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support $([\cdot], \emptyset)$. We define the family $(\leq_n)_{n \in \mathbb{N}}$ of subtyping relations as

$$\leq_n \triangleq \forall t, s \in \mathcal{D} \colon t \leq_n s \iff \forall \eta. [t]_n \eta \subseteq [s]_n \eta$$

where $[\cdot]_n$ is the rank $n$ interpretation of a type, defined as

$$[t]_n \eta = \{ d \in [t]_\eta \mid \text{height}(d) \leq n \}$$

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

$$\text{height}(c) = 1$$

$$\text{height}((d, d')) = \max(\text{height}(d), \text{height}(d')) + 1$$

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

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

1. $t \leq_n s$ for all $t, s \in \mathcal{D}$.
2. $t \leq s \iff \forall n. t \leq_n s$.
3. $\bigwedge_{i \in I} (t_i \times s_i) \leq_{n+1} \bigvee_{j \in J} (t_j \times s_j) \iff \forall J' \subseteq J \colon \bigvee_{i \in I} t_i \leq_n \bigvee_{j \in J \setminus J'} t_j$
4. $\bigwedge_{i \in I} (t_i \rightarrow s_i) \leq_{n+1} \bigvee_{j \in J} (t_j \rightarrow s_j) \iff \exists j_0 \in J : \forall I' \subseteq I. \bigwedge_{i \in I \setminus I'} t_i \leq_n s_{j_0}$

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

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

$$\sigma \vdash_n C \triangleq \forall (t, \leq, s) \in C. t \leq_n s \text{ and } \forall (t, \geq, s) \in C. s \leq_n t$$

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

1. $\sigma \vdash_n C$ for all $\sigma$ and $C$.
2. $\sigma \vdash_n C \iff \forall n. \sigma \vdash_n C$.

**Proof.** Consequence of Lemma C.7.

---

See [4] for the definitions of the notions of models, interpretations, and assignments.
Lemma C.10 (Soundness). Let $C$ be a constraint-set. If $\emptyset \vdash_{\neg \neg} C \leadsto \mathcal{J}$, then for all normalized constraint-set $C' \in \mathcal{J}$ and all substitution $\sigma$, we have $\sigma \models C' \Rightarrow \sigma \models C$.

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\neg \neg} C \leadsto \mathcal{J}$. For all $C' \in \mathcal{J}$, $\sigma$ and $n$, if $\sigma \models_n C(\Sigma)$ and $\sigma \models_n C'$, then $\sigma \models_n C$.

(2) Assume $\Sigma \vdash_{\neg \neg} C \leadsto \mathcal{J}$. For all $C' \in \mathcal{J}$, $\sigma$ and $n$, if $\sigma \models_n C(\Sigma)$ and $\sigma \models_n C'$, then $\sigma \models_{n+1} C$.

Before proving these statements, we explain how the first property implies the lemma. Suppose $\emptyset \vdash_{\neg \neg} C \leadsto \mathcal{J}$ and $\Sigma \vdash_{\neg \neg} C \leadsto \mathcal{J}$. It is easy to check that $\sigma \models_n C(\emptyset)$ holds for all $n$. From $\Sigma \vdash_{\neg \neg} C'$, we deduce $\sigma \models_n C'$ for all $n$ (by Lemma C.9). By Property (1), we have $\sigma \models_n C$ for all $n$, and we have then the required result by Lemma C.9.

We prove these two properties simultaneously by induction on the derivations of $\Sigma \vdash_{\neg \neg} C \leadsto \mathcal{J}$ and $\Sigma \vdash_{\neg \neg} C \leadsto \mathcal{J}$.

(NEMPTY): straightforward.

(NJOIN): according to Definition C.4, if there exists $C_i \in \mathcal{J}$ such that $C_i = \emptyset$, then $\bigcap_{i \in I} \mathcal{J}_i = \emptyset$, and the result follows immediately. Otherwise, we have $C' = \bigcup_{i \in I} C_i$, where $C_i \in \mathcal{J}_i$. As $\sigma \models_n C'$, then clearly $\sigma \models_n C_i$. By induction, we have $\sigma \models_n \{ (t_i, c_i, t_i') \}$. Therefore, we get $\sigma \models_n \{ (t_i, c_i, t_i') \mid i \in I \}$.

(NSYM): by induction, we have $\sigma \models_n \{ (t \leq t') \}$. Then clearly $\sigma \models_n \{ (t' \leq t) \}$.

(NZERO): by induction, we have $\sigma \models_n \{ (t \land \neg t' \leq 0) \}$. According to set-theory, we have $\sigma \models_n \{ (t \leq t') \}$.

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

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

(NTLV): assume $\alpha'$ has the smallest order in $P \cup N$. If $\alpha' \in P$, then we have $C' = (\alpha', \leq, \neg \alpha')$. From $\sigma \models_n C'$, we deduce $\sigma(\alpha') \subseteq \neg \alpha'. \sigma$. Intersecting both sides of the inequality by the same type, we obtain $\sigma(\alpha') \land t_\alpha' \sigma \subseteq \subseteq 0$, that is, $\sigma \models_n \{ (\forall \alpha \in P \sigma(\alpha) \land \forall \alpha \in N \neg \alpha \land \tau_\sigma \leq 0) \}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.

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

(NASSUM): if $n = 0$, then $\sigma \models_0 \{ (\tau_0 \leq 0) \}$ holds. Suppose $n > 0$. From $\sigma \models_n C(\Sigma)$ and $\sigma \models_k C'$, it is easy to prove that $\sigma \models_k C(\Sigma) \; (\ast)$ and $\sigma \models_k C' \; (\ast \ast)$ hold for all $0 \leq k \leq n$. We now prove that $\sigma \models_k \{ (\tau_0 \leq 0) \} \; (\ast \ast \ast)$ holds for all $1 \leq k \leq n$. By definition of $\models_0$, we have $\sigma \models_0 C(\Sigma \cup \{ \tau_0 \})$ and $\sigma \models_k C'$. Consequently, by the induction hypothesis (item (2)), we have $\sigma \models_1 \{ \tau_0 \leq 0 \}$. From this and (\ast \ast), we deduce $\sigma \models_1 C(\Sigma \cup \{ \tau_0 \})$. Because we also have $\sigma \models_1 C'$ (by \ast \ast \ast), we can use the induction hypothesis (item (2)) again to deduce $\sigma \models_2 \{ (\tau_0 \leq 0) \}$. Hence, we can prove \ast \ast \ast by induction on $1 \leq k \leq n$. In particular, we have $\sigma \models_n \{ (\tau_0 \leq 0) \}$, which is the required result.

(NBASIC): straightforward.

(NPROD): if $\bigcup_{N' \subseteq N} \{ \mathcal{J}_{N'} \cup \mathcal{J}_{N'} \} \mapsto_{\neg \neg} \emptyset$, then the result follows straightforwardly. Otherwise, we have $C = \bigcup_{N' \subseteq N} C_{N'}$, where $C_{N'} \in \mathcal{J}_{N'} \cup \mathcal{J}_{N'}$. Since $\sigma \models_n C'$, we have $\sigma \models_n C_{N'}$ for all subset $N' \subseteq N$. Moreover, following Definition C.4, either $C_{N'} \in \mathcal{J}_{N'}$ or $C_{N'} \in \mathcal{J}_{N'}$. By induction, we have either $\sigma \models_n \{ \bigvee_{i \in P} \; t_i \sigma \land \bigwedge_{j \in N' \setminus N} \neg t_j \sigma \leq 0 \}$ or $\sigma \models_n \{ \bigvee_{i \in P} \; t_i \sigma \land \bigwedge_{j \in N \setminus N'} \neg t_j \sigma \leq 0 \}$. That is, for all subset $N' \subseteq N$, we have

$$\bigwedge_{i \in P} \; t_i \sigma \land \bigwedge_{j \in N'} \neg t_j \sigma \leq 0 \quad \text{or} \quad \bigwedge_{i \in P} \; t_i \sigma \land \bigwedge_{j \in N \setminus N'} \neg t_j \sigma \leq 0$$

Applying Lemma C.7, we have

$$\bigwedge_{i \in P} \; (t_i \sigma \land \bigvee_{j \in N' \setminus N'} \neg t_j \sigma) \leq_{n+1} 0$$

Thus, $\sigma \models_{n+1} \{ \bigwedge_{i \in P} \; (t_i \sigma \land \bigvee_{j \in N'} \neg t_j \sigma \leq 0) \}$.

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

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 $\{ (\alpha, e, t) \in C \mid \alpha \in X \}$.

Lemma C.11. Let $t$ be a type and $\emptyset \vdash_{\neg \neg} \{ (t, \leq, 0) \} \leadsto \mathcal{J}$. Then for all normalized constraint-set $C \in \mathcal{J}$, all substitution $\sigma$ and all $n$, if $\sigma \models_n C|_{\text{type}(t)}$ and $\sigma \models_{n+1} C \setminus \text{type}(t)$, then $\sigma \models_n \{ (t, \leq, 0) \}$.

Proof. By applying the rules (NDNNF) and (NUNION), the constraint-set $\{ (t, \leq, 0) \}$ is normalized into a new constraint-set $C'$, consisting of the constraints of the form $(\tau, \leq, 0)$, where $\tau$ is a single normal form. That is, $\emptyset \vdash_{\neg \neg} \{ (t, \leq, 0) \} \leadsto \{ C' \}$. Let $C_1 = \{ (\tau, \leq, 0) \in C' \mid \tau \text{ is not a single normal form} \}$ and $C_2 = C' \setminus C_1$. It is easy to deduce that all the constraints in $C \setminus \text{type}(t)$ are generated from $C_2$ and must pass at least one instance of $\neg \neg$ (i.e., being decomposed at least once). Since $\sigma \models_{n+1} C \setminus \text{type}(t)$, then according to the statement (2) in the proof of Lemma C.10, we have $\sigma \models_n C_2$. Moreover, from $\sigma \models_n C_2|_{\text{type}(t)}$, we have $\sigma \models_n C_2^t$. Thus, $\sigma \models_n C'$ and a fortiori $\sigma \models_n \{ (t, \leq, 0) \}$.
Lemma C.12 (Completeness). Let $C$ be a constraint-set such that $\emptyset \vdash \neg C \leadsto \mathcal{J}$. For all substitution $\sigma$, if $\sigma \vdash C$, then there exists $C' \in \mathcal{J}$ such that $\sigma \vdash C'$.

Proof. We prove the following stronger statements.

1. Assume $\Sigma \vdash \neg C \leadsto \mathcal{J}$. For all $\sigma$, if $\sigma \vdash C(\Sigma)$ and $\sigma \vdash C$, then there exists $C' \in \mathcal{J}$ such that $\sigma \vdash C'$.

2. Assume $\Sigma \vdash^* \neg C \leadsto \mathcal{J}$. For all $\sigma$, if $\sigma \vdash C(\Sigma)$ and $\sigma \vdash C$, then there exists $C' \in \mathcal{J}$ 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 $\sigma$). We prove the two items simultaneously by induction on the derivations of $\Sigma \vdash \neg C \leadsto \mathcal{J}$ and $\Sigma \vdash^* \neg C \leadsto \mathcal{J}$.

(NEMPTY): straightforward.

(NJOIN): as $\sigma \vdash \{ \mathit{t}(i, c_i, t'_i) \mid i \in I \}$, we have in particular $\sigma \vdash \{ \mathit{t}(i, c_i, t'_i) \}$ for all $i$. By induction, there exists $C_i \in \mathcal{J}$ such that $\sigma \vdash C_i$. So $\sigma \vdash \bigcup_{i \in I} C_i$. Moreover, according to Definition C.4, $\bigcup_{i \in I} C_i$ must be in $\bigcap_{i \in I} \mathcal{J}_i$. Therefore, the result follows.

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

(NZERO): since $\sigma \vdash \{ (t \leq t'_i) \}$, we can subtract $t'$ from both sides obtain $\sigma \vdash \{ (t \land \neg t' \leq 0) \}$. By induction, the result follows.

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

(NTLV): assume $\alpha'$ has the smallest order in $P \cup N$. If $\alpha' \in P$, then according to set-theory, we have $\alpha_\sigma \leq \neg(\bigcap_{a \in (P \setminus \{\alpha'\})} \alpha_\sigma \land \bigcap_{a \in N} \neg(\alpha_\sigma \land \tau_i))$, that is $\sigma \vdash \{ (\forall i : \alpha'_i \leq \neg \alpha_i) \}$. 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 $\sigma \vdash \{ (\tau_0 \leq 0) \}$, we have $\sigma \vdash C(\Sigma \cup \{ \tau_0 \})$. By induction, the result follows.

(NBASIC): straightforward.

(NPROD): as

$$\sigma \vdash \{ \bigwedge_{i \in P} (t_i \times t'_i) \land \bigwedge_{j \in N} \neg(t_j \times t'_j) \leq 0 \}$$

we have

$$\bigwedge_{i \in P} (t_i \times t'_i) \land \bigwedge_{j \in N} \neg(t_j \times t'_j) \sigma \leq 0$$

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

$$\bigwedge_{i \in P} t_i \sigma \land \bigwedge_{j \in N'} \neg t_j \sigma \leq 0 \lor \bigwedge_{i \in P} t_i \sigma \land \bigwedge_{j \in N' \setminus N'} \neg t_j \sigma \leq 0$$

that is,

$$\sigma \vdash \{ \bigwedge_{i \in P} (t_i \land \bigwedge_{j \in N'} \neg t_j \leq 0) \} \lor \sigma \vdash \{ \bigwedge_{i \in P} (t_i \land \bigwedge_{j \in N' \setminus N'} \neg t_j \leq 0) \}$$

By induction, either there exists $C^1_{N'} \in \mathcal{J}^{1}_{N'}$ such that $\sigma \vdash C^1_{N'}$ or there exists $C^2_{N'} \in \mathcal{J}^{2}_{N'}$ such that $\sigma \vdash C^2_{N'}$. According to Definition C.4, we have $C^1_{N'}, C^2_{N'} \in \mathcal{J}_{N'} \cup \mathcal{J}_{N'}$. Thus there exists $C'_{N'} \in \mathcal{J}_{N'} \cup \mathcal{J}_{N'}$ such that $\sigma \vdash C'_{N'}$. Therefore $\sigma \vdash \bigcup_{N' \subseteq N} C'_{N'}$. Moreover, according to Definition C.4 again, $\bigcup_{N' \subseteq N} C'_{N'} \in \bigcap_{N' \subseteq N} (\mathcal{J}_{N'} \cup \mathcal{J}_{N'})$. 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 C.13 (Plinth). A plinth $\Xi \subseteq \mathcal{J}$ is a set of types with the following properties:

- $\Xi$ is finite;
- $\Xi$ contains $\bot, 0$ and is closed under Boolean connectives ($\land, \lor, \neg$);
- for all types ($t_1 \times t_2$ or ($t_1 \rightarrow t_2$)) in $\Xi$, we have $t_1 \in \Xi$ and $t_2 \in \Xi$.

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 $\Xi$, then the set of its subtrees is contained in $\Xi$. This is used to show the termination of algorithms working on types.

Lemma C.14 (Termination). Let $C$ be a finite constraint-set. Then the normalization of $C$ terminates.
extend to the case of |1| following stronger statement:

BASIC negative parts and two types for the positive parts from make the type empty.

By induction on the number of (N \{null, s, t\} \leq 0) during the process of \emptyset \vdash \sigma \arrow S, t would belong to \Sigma. We prove the lemma by induction on (|\Sigma|, U, |C|) lexicographically ordered, where \Sigma is the set of types we have normalized, U is the number of unions \lor occurring in the constraint-set C plus the number of constraints (t \leq s) and the number of constraint (t \leq s) where s \neq 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, and neither |\Sigma| nor U increase.

(NSYM): U decreases and \Sigma is unchanged.

(NZERO): U decreases and \Sigma is unchanged.

(NDNF): U decreases and \Sigma is unchanged.

(NUNION): although |C| increases, U decreases and \Sigma is unchanged.

(NTLV): it terminates immediately.

(NHYP): it terminates immediately.

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

(NBASIC): it terminates immediately.

(NPROD): although (|\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).

\[\square\]

**Lemma C.15 (Finiteness).** Let C be a constraint-set and \emptyset \vdash \sigma \arrow S. Then S is finite.

**Proof.** It is easy to prove that each normalizing rule generates a finite set of finite sets of normalized constraints. \[\square\]

**Definition C.16.** Let C be a normalized constraint-set and O an ordering on \text{var}(C). We say C is well-ordered if for all normalized constraint (\alpha, c, t, s) \in C and for all \beta \in \text{tv}(t, s), O(\alpha) < O(\beta).

**Lemma C.17.** Let C be a constraint-set and \emptyset \vdash \sigma \arrow S. Then for all normalized constraint-set C' \subseteq 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 \alpha whose order is minimum as the representative one, that is, \forall \beta \in \text{tv}(t, s). O(\alpha) < O(\beta). Therefore, the result follows. \[\square\]

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

**Lemma C.19.** Let t, s be two types and \{p_i\}_{i \in I}, \{q_j\}_{j \in J} two sets of general renamings. Then if \emptyset \vdash \sigma \arrow \emptyset, then \emptyset \vdash \sigma \arrow ((\bigwedge_{j \in J} q_j) \land (\bigwedge_{i \in I} t p_i), \leq, 0) \arrow \emptyset.

**Proof.** By induction on the number of (NPROD) and (NARROW) used in the derivation of \emptyset \vdash ((s \land \neg t, \leq, 0)) and by cases on the disjunctive normal form \tau of s \land \neg t. The failure of the normalization of (s \land t, \leq, 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 the general renaming is a little complicated, as we need to consider more than two types: one for the negative parts and two for the positive parts from t and s respectively. Indeed, what we prove is the following stronger statement:

\[\emptyset \vdash ((\bigwedge_{k \in K} t_k, \leq, 0)) \rightarrow \emptyset \Rightarrow \emptyset \vdash ((\bigwedge_{k \in K} t_k p_{i_k}, \leq, 0)) \rightarrow \emptyset\]

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

**Case 1:** \(\tau = \tau_{s_b} \land \tau_{s_b} \land \tau \neq 0\), where \(\tau_{s_b}\) (\(\tau_{s_b}\) resp.) is an intersection of basic types from s (t resp.).

Then the expansion of \tau is

\[(\bigwedge_{j \in J} \tau_{s_b} p_j) \land (\bigwedge_{i \in I} \tau_{s_b} p_i) \simeq \tau_{s_b} \land \tau_{s_b} \neq 0\]

And so \(\emptyset \vdash ((\bigwedge_{j \in J} \tau_{s_b} p_j) \land (\bigwedge_{i \in I} \tau_{s_b} p_i), \leq, 0) \rightarrow \emptyset\).
Case 2: \( \tau = \bigwedge_{p,s \in P_s} (w_{p,s} \times v_{p,s}) \land \bigwedge_{n \in N_s} \neg (w_{n,s} \times v_{n,s}) \land \bigwedge_{p,t \in P_t} (w_{p,t} \times v_{p,t}) \land \bigwedge_{n \in N_t} \neg (w_{n,t} \times v_{n,t}) \), where \( P_s, N_s \) are from \( s \) and \( P_t, N_t \) are from \( t \). Since \( \emptyset \vdash \ \{ \tau, \leq, 0 \} \) \( \vdash \emptyset \), by the rule \((\text{NPROD})\), there exist two sets \( N'_s \subseteq N_s \) and \( N'_t \subseteq N_t \) such that

\[
\begin{align*}
\emptyset \vdash \tau' \ & \ \
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\end{align*}
\]

By induction, we have

\[
\begin{align*}
\emptyset \vdash \tau' \ & \ \
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\end{align*}
\]

Then by the rule \((\text{NPROD})\) again, we get

\[
\begin{align*}
\emptyset \vdash \tau' \ & \ \
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\end{align*}
\]

where \( \tau_s = \bigwedge_{p \in P_s} (w_{p,s} \times v_{p,s}) \land \bigwedge_{n \in N_s} \neg (w_{n,s} \times v_{n,s}) \) and \( \tau_t = \bigwedge_{p \in P_t} (w_{p,t} \times v_{p,t}) \land \bigwedge_{n \in N_t} \neg (w_{n,t} \times v_{n,t}) \).

Case 3: \( \tau = \bigwedge_{p,s \in P_s} (w_{p,s} \to v_{p,s}) \land \bigwedge_{n \in N_s} \neg (w_{n,s} \to v_{n,s}) \land \bigwedge_{p,t \in P_t} (w_{p,t} \to v_{p,t}) \land \bigwedge_{n \in N_t} \neg (w_{n,t} \to v_{n,t}) \), where \( P_s, N_s \) are from \( s \) and \( P_t, N_t \) are from \( t \). Since \( \emptyset \vdash \ \{ \tau, \leq, 0 \} \) \( \vdash \emptyset \), by the rule \((\text{NARROW})\), for all \( v \in N_s \cup N_t \), there exist a set \( P'_s \subseteq P_s \) and a set \( P'_t \subseteq P_t \) such that

\[
\begin{align*}
\emptyset \vdash \tau' \ & \ \
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\end{align*}
\]

By induction, for all \( \rho \in [p]_{i \in I} \cup [\rho]_{j \in J} \), we have

\[
\begin{align*}
\emptyset \vdash \tau' \ & \ \
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\end{align*}
\]

Then by the rule \((\text{NARROW})\) again, we get

\[
\begin{align*}
\emptyset \vdash \tau' \ & \ \
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\emptyset \vdash \tau' \ & \ 
\end{align*}
\]

where \( \tau_s = \bigwedge_{p \in P_s} (w_{p,s} \to v_{p,s}) \land \bigwedge_{n \in N_s} \neg (w_{n,s} \to v_{n,s}) \) and \( \tau_t = \bigwedge_{p \in P_t} (w_{p,t} \to v_{p,t}) \land \bigwedge_{n \in N_t} \neg (w_{n,t} \to v_{n,t}) \).

Case 4: \( \tau = \bigvee_{k \in K_s} \tau_{k,s} \) \( \land \bigvee_{k \in K_t} \tau_{k,t} \), where \( \tau_{k,s} \) and \( \tau_{k,t} \) are single normal forms. As \( \emptyset \vdash \ \{ \tau, \leq, 0 \} \) \( \vdash \emptyset \), there must exist at least one \( k_s \in K_s \) and at least one \( k_t \in K_t \) such that \( \emptyset \vdash \ \{(\tau_{k,s} \land \tau_{k,t}, \leq, 0)\} \) \( \vdash \emptyset \). By Cases (1) – (3), the result follows.

\[\Box\]

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

\[\text{tlv}(\tau_0) = \emptyset \ \ \alpha' \notin \tau_0 \cup N \ \ \mathcal{J} = \begin{cases} \{\{\alpha', \leq, \neg \tau_{0'}\}\} & \alpha' \in P \setminus \Delta \\ \{\{\alpha', \geq, \tau_{0'}\}\} & \alpha' \in N \setminus \Delta \\ \Sigma \vdash \ \{\tau_0, \leq, 0\} & \alpha' \in \Delta \\ \end{cases}\]

\[\Sigma \vdash \ \{\bigwedge_{\alpha \in \tau_0} \alpha \land \bigwedge_{\alpha \in \neg \tau_0} \neg \alpha \land \tau_0 \leq 0\} \vdash \mathcal{J}\]
Furthermore, it is easy to prove the soundness, completeness, and termination of the algorithm extended with \( \Delta \).

### C.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 \( \alpha \geq t_1 \lor t_2 \), and the constraints \( \alpha \leq t_1 \) and \( \alpha \leq t_2 \) can be replaced by \( \alpha \leq t_1 \land t_2 \). 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).

\[
\forall i \in I. \ (\alpha \geq t_i) \in C \quad |I| \geq 2 \\
\vdash_{\#} C \rightsquigarrow (C \setminus \{(\alpha \geq t_i) \mid i \in I\}) \cup \{(\alpha \geq \bigvee_{i \in I} t_i)\}
\]  
\text{ (MLB)}

\[
\forall i \in I. \ (\alpha \leq t_i) \in C \quad |I| \geq 2 \\
\vdash_{\#} C \rightsquigarrow (C \setminus \{(\alpha \leq t_i) \mid i \in I\}) \cup \{(\alpha \leq \bigwedge_{i \in I} t_i)\}
\]  
\text{ (MUB)}

**Figure 8.** Merging rules

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 \( \alpha \), 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 \( \mathcal{F} \), and then saturate \( C \) with any normalized constraint-set \( C' \in \mathcal{F} \). Formally, we describe the saturation process as the saturation rule \( \Sigma_p, C_S \vdash \# \rightarrow \mathcal{F} \), where \( \Sigma_p \) 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), \( C_S \) 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 \( \mathcal{F} \) is a set of sets of normalized constraints (the result of the saturation of \( C \) with \( C_S \)). 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.

\[
\Sigma_p, C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \vdash_{\#} C \rightsquigarrow \mathcal{F} \quad (t_1, t_2) \in \Sigma_p \quad \vdash_{\#} C \rightsquigarrow \mathcal{F} \nabla
\]

\[
\forall c \in \mathcal{F}, \Sigma_p, C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \vdash_{\#} C \rightsquigarrow \mathcal{F}_c
\]

\[
\forall \alpha, t_1, t_2 \not\in \mathcal{F}, \Sigma_p, C_S \vdash \{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C \quad \vdash_{\#} C \rightsquigarrow \mathcal{F} \quad \vdash_{\#} C \rightsquigarrow \{(C \cup C_S)\}
\]  
\text{ (SHYP)}

**Figure 9.** Saturation rules

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 \( C_S \) (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 \( (t_1, \leq, t_2) \), yielding a set of normalized constraint-sets \( \mathcal{F} \). It then saturates (joins) \( C \) and \( C_S \) with each constraint-set \( C_{\mathcal{F}} \in \mathcal{F} \), the union of which gives a new set \( \mathcal{F}' \) of normalized constrain-sets. Each \( C' \) in \( \mathcal{F}' \) 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 \( \alpha \), the process is over and the rule (SDONE) simply returns \( C \cup C_S \).

If \( \emptyset, \emptyset \vdash_{\#} C \rightsquigarrow \mathcal{F} \), then the result of the merging of \( C \) is \( \mathcal{F} \).
Lemma C.20 (Soundness). Let $C$ be a normalized constraint-set. If $\emptyset, \emptyset \vdash_{\mathcal{M}} C \leadsto \mathcal{S}$, then for all normalized constraint-set $C' \in \mathcal{S}$ and all substitution $\sigma$, we have $\sigma \vdash C' \Rightarrow \sigma \vdash C$.

Proof. We prove the following statements.

- Assume $\vdash_{\mathcal{M}} C \leadsto C'$. For all $\sigma$, if $\sigma \vdash C'$, then $\sigma \vdash C$.
- Assume $\Sigma_p, C_0 \vdash_{\mathcal{M}} C \leadsto \mathcal{S}$. For all $\sigma$ and $C_0 \in \mathcal{S}$, if $\sigma \vdash C_0$, then $\sigma \vdash C_0 \cup C$.

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 $\Sigma_p, C_2 \vdash_{\mathcal{M}} C \leadsto \mathcal{S}$.

(Shyp): by induction, we have $\sigma \vdash (C_2 \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\}) \cup C$, that is $\sigma \vdash C_2 \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C$.

(Sassum): according to Definition C.4, $C_0 \in \mathcal{S}_p$, for some $C' \in \mathcal{S}$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{M}} C' \leadsto \mathcal{S}$, we have $\sigma \vdash C'$. Moreover, the equation $\mathcal{S} = \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_2 \subseteq C'$. Therefore, we have $\sigma \vdash C_2 \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C$.

(SdOne): straightforward.

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

Proof. We prove the following statements.

- Assume $\vdash_{\mathcal{M}} C \leadsto C'$. For all $\sigma$, if $\sigma \vdash C$, then $\sigma \vdash C'$.
- Assume $\Sigma_p, C_0 \vdash_{\mathcal{M}} C \leadsto \mathcal{S}$. For all $\sigma$, if $\sigma \vdash C_0 \cup C$, then there exists $C_0 \in \mathcal{S}$ 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 $\Sigma_p, C_2 \vdash_{\mathcal{M}} C \leadsto \mathcal{S}$.

(Shyp): the result follows by induction.

(Sassum): as $\sigma \vdash C_2 \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C$, we have $\sigma \vdash \{(t_1 \leq t_2)\}$. As $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\}$, applying Lemma C.12, there exists $C_0 \in \mathcal{S}$ such that $\sigma \vdash C_0$. Let $C' = C_2 \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C_0$. Clearly we have $\sigma \vdash C'$ and $C' \in \mathcal{S}'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{M}} C' \leadsto \mathcal{S}_c$, there exists $C_0 \in \mathcal{S}_c$ such that $\sigma \vdash C_0$. Moreover, it is clear that $C_0 \in \bigcup_{C \in \mathcal{S}_c} \mathcal{S}_c$. Therefore, the result follows.

(SdOne): straightforward.

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

Proof. Let $T$ be the set of types occurring in $C$. As $C$ is finite, $T$ is finite as well. Let $\mathfrak{T}$ be a plinth such that $T \subseteq \mathfrak{T}$. Then when we saturate a fresh constraint $(t_1, \leq t_2)$ during the process of $\emptyset, \emptyset \vdash_{\mathcal{M}} C$, $(t_1, t_2)$ would belong to $\mathfrak{T} \times \mathfrak{T}$. According to Lemma C.14, we know that $\emptyset \vdash_{\mathcal{N}} \{(t_1, \leq t_2)\}$ 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 $|\mathfrak{T} | - |\Sigma_p, |C| |$ lexicographically ordered:

(Shyp): $|C|$ decreases.

(Sassum): as $(t_1, t_2) \notin \Sigma_p$ and $t_1, t_2 \in \mathfrak{T}$, $|\mathfrak{T} \times \mathfrak{T} | - |\Sigma_p|$ decreases.

(SdOne): it terminates immediately.

Definition C.23 (Sub-constraint). Let $C_1, C_2 \subseteq \mathcal{G}$ be two normalized constraint-sets. We say $C_1$ is a sub-constraint of $C_2$, denoted as $C_1 \ll C_2$, 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 C.24. Let $C_1, C_2 \subseteq \mathcal{G}$ be two normalized constraint-sets and $C_1 \ll C_2$. Then for all substitution $\sigma$, if $\sigma \vdash_{\mathcal{M}} C_2$, then $\sigma \vdash_{\mathcal{M}} C_1$.

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 $\sigma \vdash_{\mathcal{M}} C_2$, then $\sigma(\alpha) t' c \sigma t$. Thus $\sigma(\alpha) c t \sigma$. Therefore, we have $t' c t \sigma t$. Thus $\sigma(\alpha) c t \sigma$.

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

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 \(\emptyset \vdash_{\mathcal{A}} \{(t_1 \leq t_2)\} \sim \mathcal{I}\) and there exists \(C' \in \mathcal{I}\) such that \(C'\) is a sub-constraint of \(C\) (i.e., \(C' \subseteq C\)).

**Lemma C.26.** Let \(C\) be a finite normalized constraint-set and \(\emptyset, \emptyset \vdash_{\mathcal{A}} C \sim \mathcal{I}\). Then for all normalized constraint-set \(C' \in \mathcal{I}\), \(C'\) is saturated.

**Proof.** We prove a stronger statement: assume \(\Sigma_p, C_S \vdash_{\mathcal{A}} C \sim \mathcal{I}\). If

(i) for all \((t_1, t_2) \in \Sigma_p\), there exists \(C' \in \{(\emptyset \vdash_{\mathcal{A}} \{(t_1 \leq t_2)\}\} \) such that \(C' \subseteq C_S \cup C\) and

(ii) for all \((\alpha \geq t_1), (\alpha \leq t_2) \subseteq \Sigma_p\) the pair \((t_1, t_2)\) is in \(\Sigma_p\),

then \(C_0\) is saturated for all \(C_0 \in \mathcal{I}\).

The proof of conditions (1) and (2) for a saturated constraint-set is straightforward for all \(C_0 \in \mathcal{I}\). The proof of the condition (3) proceeds by induction on the derivation \(\Sigma_p, C_S \vdash_{\mathcal{A}} C \sim \mathcal{I}\) and a case analysis on the last rule used in the derivation.

**Lemma C.27 (Finiteness).** Let \(C\) be a constraint-set and \(\emptyset, \emptyset \vdash_{\mathcal{A}} C \sim \mathcal{I}\). Then \(\mathcal{I}\) is finite.

**Proof.** It follows by Lemma C.15.

**Lemma C.28.** Let \(C\) be a well-ordered normalized constraint-set and \(\emptyset, \emptyset \vdash_{\mathcal{A}} C \sim \mathcal{I}\). Then for all normalized constraint-set \(C' \in \mathcal{I}\), \(C'\) 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 C.17.

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

\[
\{(\alpha, \leq, 0), (\beta, \leq, 0), (\alpha, \leq, 0), (\beta, \leq, 0), (\alpha, \leq, \text{Int}), (\beta, \leq, \text{Bool})\}.
\]

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

**Definition C.29.** Let \(\mathcal{I}\) be a set of normalized constraint-sets. We say that \(\mathcal{I}\) is minimal if for any two different normalized constraint-sets \(C_1, C_2 \in \mathcal{I}\), neither \(C_1 \leq C_2\) nor \(C_2 \leq C_1\). Moreover, we say \(\mathcal{I} \simeq \mathcal{I}'\) if for all substitution \(\sigma\) such that \(\exists C \in \mathcal{I}. \sigma \vdash C \iff \exists C' \in \mathcal{I}'. \sigma \vdash C'\).

**Lemma C.30.** Let \(C\) be a well-ordered normalized constraint-set and \(\emptyset, \emptyset \vdash_{\mathcal{A}} C \sim \mathcal{I}\). Then there exists a minimal set \(\mathcal{I}_0\) such that \(\mathcal{I}_0 \simeq \mathcal{I}\).

**Proof.** By eliminating the redundant constraint-sets in \(\mathcal{I}\).

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

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

\[
\forall \alpha \equiv t \in E. \sigma(\alpha) \equiv t\sigma \text{ holds}
\]

If \(\sigma\) is a solution to \(E\), we write \(\sigma \vdash E\).
From a normalized constraint-set $C$, we obtain some explicit conditions for the substitution $\sigma$ 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 $\alpha$ must be a subtype of $t$ (resp. a super type of $t$).

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

$$
\begin{aligned}
&\begin{cases}
  t_1 \leq \alpha \leq 1 & \text{if } \alpha \geq t_1 \in C \text{ and } \not\exists t \in C \\
  0 \leq \alpha \leq t_2 & \text{if } \alpha \leq t_2 \in C \text{ and } \not\exists t \geq \alpha \in C \\
  t_1 \leq \alpha \leq t_2 & \text{if } \alpha \geq t_1, \alpha \leq t_2 \in C
\end{cases}
\end{aligned}
$$

We then transform each constraint $t_1 \leq \alpha \leq t_2$ in $C$ into an equation $\alpha = (t_1 \lor \alpha') \land t_2$, where $\alpha'$ is a fresh type variable. The type $(t_1 \lor \alpha') \land t_2$ ranges from $t_1$ to $t_2$, so the equation $\alpha = (t_1 \lor \alpha') \land t_2$ 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 $\models_n$ for equation systems, which is similar to the one for constraint-sets.

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

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

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

Here $C_{\Sigma}$ 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 $\mathcal{O}(\alpha)$ is the maximum in $\operatorname{dom}(C \setminus C_{\Sigma})$.
  
  Clearly, there exists a corresponding equation $\alpha = (t_1 \lor \alpha') \land t_2 \in E$. As $\sigma \models_n E$, we have $\sigma(\alpha) \simeq_n ((t_1 \lor \alpha') \land t_2)$. Therefore, $\sigma(\alpha) \wedge \neg t_2 \sigma \simeq_n (t_1 \lor \alpha') \land t_2 \land \neg t_2 \sigma \simeq_n 0$.

  Therefore, $\sigma(\alpha) \leq_n t_2 \sigma$.

  Consider the constraint $(t_1, \leq, \alpha)$. We have

  $$
  t_1 \sigma \wedge \neg \sigma(\alpha) \simeq_n t_1 \sigma \wedge \neg ((t_1 \lor \alpha') \land t_2) \sigma \simeq_n t_1 \sigma \wedge \neg t_2 \sigma
  $$

  What remains to do is to check the subtyping relation $t_1 \sigma \wedge \neg t_2 \sigma \leq_n 0$, that is, to check that the judgement $\sigma \models_n \{(t_1 \leq t_2)\}$ holds. Since the whole constraint-set $C$ is saturated, according to Definition C.25, we have $\emptyset \models_n \{(t_1 \leq t_2)\}$ and there exists $C' \subseteq \mathcal{I}$ such that $C' \prec C$, that is $C' \prec C_{\Sigma} \cup C \setminus C_{\Sigma}$. Moreover, as $C$ is well-ordered, $\mathcal{O}(\alpha) = \mathcal{O}(\beta) < \mathcal{O}(\mathcal{tv}(t_1) \cup \mathcal{tv}(t_2))$ and thus $\mathcal{O}(C \setminus C_{\Sigma}) < \mathcal{O}(\mathcal{tv}(t_1) \cup \mathcal{tv}(t_2))$. Therefore, we can deduce that $C' \setminus \mathcal{tv}(t_1) \cup \mathcal{tv}(t_2) < C_{\Sigma}$ and $C' \setminus \mathcal{tv}(t_1) \cup \mathcal{tv}(t_2) \prec C \setminus C_{\Sigma}$. From the premise and Lemma C.24, we have $\sigma \models n C' \setminus \mathcal{tv}(t_1) \cup \mathcal{tv}(t_2)$ and $\sigma \models_{n-1} C' \setminus \mathcal{tv}(t_1) \cup \mathcal{tv}(t_2)$. Then, by Lemma C.11, we get $\sigma \models_n \{(t_1 \leq t_2)\}$. Finally, consider the constraint-set $C \setminus (C_{\Sigma} \cup \{(t_1 \leq \alpha \leq t_2)\})$. By induction, we have $\sigma \models_n C \setminus (C_{\Sigma} \cup \{(t_1 \leq \alpha \leq t_2)\})$. Thus the result follows.

Finally, we explain how to prove the lemma with the statement (*). Take $C_{\Sigma} = \emptyset$. Since $\sigma \models E$, we have $\sigma \models_n E$ for all $n$. Trivially, we have $\sigma \models_0 C$. This can be used to prove $\sigma \models_1 C$. Since $\sigma \models_1 E$, by (*), we get $\sigma \models_1 C$, which will be used to prove $\sigma \models_2 C$. Consequently, we can get $\sigma \models_n C$ for all $n$, which clearly implies the lemma.

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

**Proof.** Let $\sigma' = \{\sigma(\alpha)/\alpha' \mid \alpha \in \operatorname{dom}(C)\}$. Consider each equation $\alpha = (t_1 \lor \alpha') \land t_2 \in E$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. As $\sigma \models E$, then $t_1 \sigma \leq \sigma(\alpha) \land \sigma(\alpha) \leq t_2 \sigma$.

$$
\begin{aligned}
(t_1 \lor \alpha') \land t_2 (\sigma \cup \sigma') &= (t_1(\sigma \cup \sigma') \lor \alpha'(\sigma \cup \sigma')) \land t_2 (\sigma \cup \sigma') \\
&= (t_1 \sigma \lor \sigma(\alpha)) \land t_2 \sigma \\
&\simeq (\sigma(\alpha) \land t_2 \sigma) \\
&\simeq (\sigma(\alpha) \land (\sigma(\alpha) \leq t_2 \sigma)) \\
&\simeq (\sigma(\alpha)) (\sigma(\alpha) \leq t_2 \sigma)
\end{aligned}
$$

\[12\] Or, equivalently, $\alpha = t_1 \lor (\alpha' \land t_2)$. Besides, in practice, if only $\alpha \geq t_1$ ($\alpha \leq t_2$ resp.) and all the occurrences of $\alpha$ 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.
Definition C.35. Let $E$ be an equation system and $O$ an ordering on $\text{dom}(E)$. We say that $E$ is well ordered if for all $\alpha = t_\alpha \in E$, we have $O(\alpha) < O(\beta)$ for all $\beta \in \text{tlv}(t_\alpha) \cap \text{dom}(E)$.

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

Proof. Clearly, $\text{dom}(E) = \text{dom}(C)$. Consider an equation $\alpha = (t_1 \cup \alpha') \land t_2$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha' \leq t_2 \in C$. By Definition C.16, for all $\beta \in (\text{tlv}(t_1) \cup \text{tlv}(t_2)) \cap \text{dom}(C)$, $O(\alpha) < O(\beta)$. Moreover, $\alpha'$ is a fresh type variable in $C$, that is $\alpha' \notin \text{dom}(C)$. And then $\alpha' \notin \text{dom}(E)$. Therefore, $\text{tlv}(t_1 \cup \alpha') \land t_2 \cap \text{dom}(E) = (\text{tlv}(t_1) \cup \text{tlv}(t_2)) \cap \text{dom}(C)$. 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 $\alpha = t_\alpha$, $\alpha$ may also appear in the type $t_\alpha$, such an equality reminds the definition of a recursive type. As a first step, we introduce a recursion operator $\mu$ 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 $x_\alpha \notin \text{fv}(t_\alpha)$), 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 $\mu$ as explained above. The next step is to substitute the content expression $\mu x_\alpha, t_\alpha$ for all the occurrences of $\alpha$ in equations in $E$. In detail, let $\beta = \mu x_\beta, t_\beta \in E$. Since $t_\alpha$ may contain some occurrences of $\beta$ and these occurrences are clearly bounded by $\mu x_\beta$, we in fact replace the equation $\beta = \mu x_\beta, t_\beta$ with $\beta = \mu x_\beta, t_\beta \{\mu x_\alpha, t_\alpha\} \{x_\beta/\beta\}$, yielding a new equation system $E'$. Finally, assume that the equation system $E'$ (which has fewer equations) has a solution $\sigma'$. 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.

![Figure 10. Equation system solving algorithm Unify()](image)

Definition C.37 (General solution). Let $E$ be an equation system. A general solution to $E$ is a substitution $\sigma$ from $\text{dom}(E)$ to $\mathcal{T}$ such that

\[
\forall \alpha \in \text{dom}(\sigma), \text{var}(\sigma(\alpha)) \cap \text{dom}(\sigma) = \emptyset
\]

and

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

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

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 $\alpha \notin \text{var}(t_\alpha)$. Consider the second step. Let $\text{var}(E) = \bigcup_{(\alpha = t_\alpha) \in E} \text{var}(t_\alpha)$ and $\mathcal{S} = \mathcal{V} \setminus \mathcal{S}$, where $\mathcal{S}$ is a set of type variables. We prove a stronger statement:

\[
\forall \alpha \in \text{dom}(\sigma), \text{var}(\sigma(\alpha)) \cap (\text{dom}(\sigma) \cup \text{var}(E)) = \emptyset
\]

and $\text{dom}(\sigma) = \text{dom}(E)$.

The proof proceeds by induction on $E$:
$E = \emptyset$: straightforward.
$E = \{ (\alpha = t_{\alpha}) \} \cup E'$: let $E'' = \{ (\beta = t_{\beta}\{x_{\beta}/\beta\}) \mid (\beta = t_{\beta}) \in E' \}$. Then there exists a substitution $\sigma''$ such that $\sigma'' = \text{Unify}(E'')$ and $\sigma = \{ t_{\alpha}\sigma''/\alpha \} \oplus \sigma''$. By induction, we have $\forall \beta \in \text{dom}(\sigma'')$, $\text{var}(\sigma'(\beta)) \cap (\text{dom}(\sigma) \cup \text{var}(E'')) = \emptyset$ and $\text{dom}(\sigma'') = \text{dom}(E)$. As $\alpha \notin \text{dom}(\sigma'')$, we have $\alpha \notin \text{dom}(\sigma) \cup \{ \alpha \} = \text{dom}(E)$.

Moreover, $\alpha \notin \text{var}(E'')$, then $\text{dom}(\sigma) \subset \text{dom}(\sigma'') \cup \text{var}(E'')$. Thus, for all $\beta \in \text{dom}(\sigma'')$, we have $\text{var}(\sigma'(\beta)) \cap \text{dom}(\sigma) = \emptyset$. Consider $t_{\alpha}\sigma''$. It is clear that $\text{var}(t_{\alpha}\sigma'') \cap \text{dom}(\sigma) = \emptyset$. Besides, the algorithm does not introduce any fresh variable, then for all $\beta \in \text{dom}(\sigma)$, we have $\text{var}(t_{\beta}) \cap \text{var}(E) = \emptyset$.

Therefore, the result follows.

\[\square\]

Lemma C.39 (Soundness). Let $E$ be an equation system. If $\sigma = \text{Unify}(E)$, then $\sigma \vDash E$.

**Proof.** By induction on $E$.

$E = \emptyset$: straightforward.
$E = \{ (\alpha = t_{\alpha}) \} \cup E'$: let $E'' = \{ (\beta = t_{\beta}\{x_{\beta}/\beta\}) \mid (\beta = t_{\beta}) \in E' \}$. Then there exists a substitution $\sigma''$ such that $\sigma'' = \text{Unify}(E'')$ and $\sigma = \{ t_{\alpha}\sigma''/\alpha \} \oplus \sigma''$. By induction, we have $\sigma'' \vDash E''$. According to Lemma C.38, we have $\text{dom}(\sigma'') = \text{dom}(E'')$. So $\text{dom}(\sigma) = \text{dom}(\sigma'') \cup \{ \alpha \}$.

Considering any equation $(\beta = t_{\beta}) \in E$ where $\beta \in \text{dom}(\sigma'')$. Then

$$\sigma(\beta) = \begin{cases} \sigma''(\beta) & \text{(apply } \sigma) \\ t_{\beta}\{x_{\beta}/\beta\}\sigma'' & \text{(as } \sigma'' \vDash E'') \\ = t_{\beta}\{x_{\beta}/\beta\}\sigma''/\alpha \sigma'' & \text{(expand } \sigma'') \\ = t_{\beta}\{x_{\beta}\sigma''/\beta\} \oplus \sigma'' & \text{(as } \sigma'' \vDash E'') \\ \end{cases}$$

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

$$\sigma(\alpha) = \begin{cases} t_{\alpha}\sigma'' & \text{(apply } \sigma) \\ t_{\alpha}\{x_{\beta}\sigma''/\beta\} \mid \beta \in \text{dom}(\sigma'') & \text{(expand } \sigma'') \\ t_{\alpha}\{x_{\beta}\sigma''/\beta\} \mid \beta \in \text{dom}(\sigma'') & \text{(as } \beta \sigma'' = \beta \sigma) \\ t_{\alpha}\{x_{\beta}\sigma''/\beta\} \mid \beta \in \text{dom}(\sigma) \cup \{ \alpha \} & \text{(as } \alpha \notin \text{var}(t_{\alpha})) \\ t_{\alpha}\beta \sigma/\beta \mid \beta \in \text{dom}(\sigma) & \text{(as } \text{dom}(\sigma) = \text{dom}(\sigma'') \cup \{ \alpha \}) \\ \end{cases}$$

Thus, the result follows.

\[\square\]

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

**Proof.** Immediate consequence of Lemmas C.38 and C.39.

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

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

**Proof.** By induction on the number of equations in $E$.

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

Lemma C.43 (Completeness). Let $E$ be an equation system. For all substitution $\sigma$, if $\sigma \vDash E$, then there exist $\sigma_0$ and $\sigma'$ such that $\sigma_0 = \text{Unify}(E)$ and $\sigma \simeq \sigma' \circ \sigma_0$.

**Proof.** According to Lemma C.41, there exists $\sigma_0$ such that $\sigma_0 = \text{Unify}(E)$. For any $\alpha \notin \text{dom}(\sigma_0)$, 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 $\sigma_0 = \text{Unify}(E)$ then $\forall \alpha \in \text{dom}(\sigma_0), \alpha \sigma_0 \sigma \simeq \alpha \sigma$. The proof proceeds by induction on $E$.

$E = \emptyset$: straightforward.
Assume that there exists an ill-formed is well-formed. Let $t_\beta\{t_\alpha\{x_{\beta}\}\} \in E'$. Then there exists a substitution $\sigma''$ such that $\sigma'' = \text{Unify}(E'')$ and $\sigma_0 = \{t_\alpha \sigma'' \sigma_\alpha\} \oplus \sigma''$. Considering each equation $(\beta = t_\beta\{t_\alpha\{x_{\beta}\}\}) \in E''$, we have

$$t_\beta\{t_\alpha\{x_{\beta}\}\} = t_\beta\{t_\alpha\{x_{\beta}\}\} \oplus \sigma$$

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

$$\alpha \sigma_0 \sigma \simeq \alpha \sigma_\alpha \sigma.$$ 

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 $\mu x. x \forall t$ is not well-formed. To make the substitutions usable, we should avoid these substitutions with ill-formed types. Fortunately, this can be done by giving an ordering on the domain of an equation system to make sure that the equation system is well-ordered.

**Lemma C.44.** Let $E$ be a well-ordered equation system. If $\sigma = \text{Unify}(E)$, then for all $\alpha \in \text{dom}(\sigma)$, $\sigma(\alpha)$ 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 $\text{Unify}()$, there exists a sequence of equations $\{\alpha = \alpha_0 \simeq t_\alpha, \alpha_1 = t_\alpha, \ldots, \alpha_n = t_\alpha\}$ such that $\alpha_i \in \text{tvl}(t_{\alpha_{i-1}})$ and $\alpha_n \in \text{tvl}(t_{\alpha_n})$ for $i \in \{1, \ldots, n\}$ and $n \geq 0$. According to Definition C.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 $\mu$. 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 $\emptyset$.

**C.1.4 The complete algorithm**

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

**Theorem C.45 (Soundness).** Let $C$ be a constraint-set. If $\text{Sol}_C(C) \rightsquigarrow \Theta$, then for all $\alpha \in \Theta$, $\sigma \vdash C$.


**Theorem C.46 (Completeness).** Let $C$ be a constraint-set and $\text{Sol}_C(C) \rightsquigarrow \Theta$. Then for all substitution $\sigma$, if $\sigma \vdash C$, then there exists $\sigma' \in \Theta$ and $\sigma''$ such that $\sigma \simeq \sigma' \circ \sigma''$.

**Proof.** Consequence of Lemmas C.12, C.21, C.34 and C.43.

**Theorem C.47 (Termination).** Let $C$ be a constraint-set. Then $\text{Sol}_C(C)$ terminates.

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

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


□

### C.2 Type-Substitution Inference Algorithm

In Section B, we presented a sound and complete inference system, which is parametric in the decision procedures for $\mathcal{F}_{\Delta}, \Pi_{\Delta}(\cdot)$, and $\bullet_{\Delta}$. In this section we tackle the problem of computing these operators. We focus on the application problem $\bullet_{\Delta}s$, 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 $\forall h \in I \cup J. \sigma_h \nleq \Delta$ and

$$\bigwedge_{i \in I} t\sigma_i \leq 0 \rightarrow 1 \quad (18)$$

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

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 $s' \subseteq_{\Delta} t'$ for any $s'$ and $t'$, just by considering $t'$ ground. Therefore we can decide $\subseteq_{\Delta}$. We can also decide $[\sigma_i]_{i \in I} \vdash s \subseteq_{\Delta} \Delta \times 1$ for all $s$, and therefore compute $\Pi_{\Delta}(s)$.

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 $q$.

#### C.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 $\sigma_k (k \in I \cup J)$ into two substitutions: a renaming substitution $\rho_k$ that maps each variable in the domain of $\sigma_k$ into a fresh variable and a second substitution $\sigma'_k$ such that $\sigma_k = \sigma'_k \circ \rho_k$. The two inequalities then can be rewritten as

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

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

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

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

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

which are equivalent to

$$t'\sigma \leq 0 \rightarrow \bot$$

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

where $t' = \bigwedge_{i \in I} (t\rho_i)$ and $s' = \bigwedge_{j \in J} (s\rho_j)$. As $t'\sigma \leq 0 \rightarrow \bot$, 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$^{13}$:

$$C = \{(t', \leq, 0 \rightarrow \bot), (t', \leq, s' \rightarrow \gamma)\}$$

where $\gamma$ 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 C.1. We write $\text{AppFix}_{\Delta}(t, s)$ for the algorithm of the application problem with fixed cardinalities $t \bullet_{\Delta}s$ and $\text{AppFix}_{\Delta}(t, s) \rightsquigarrow \Theta$ if $\text{AppFix}_{\Delta}(t, s)$ terminates with $\Theta$.

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

$^{13}$ The first constraint $(t', \leq, 0 \rightarrow \bot)$ can be eliminated since it is implied by the second one.
Proof. Consider any substitution $\sigma$. As $t \sigma \leq s \sigma \rightarrow \gamma$, by Lemma C.12 in the companion paper [3], we have $s \sigma \leq \text{dom}(t \sigma)$. Then by Lemma C.13 in the companion paper [3], we get $\sigma(\gamma) \geq t \sigma \cdot s \sigma$.

Lemma C.50. Let $t, s$ be two types and $\gamma$ a type variable such that $\gamma \notin \text{var}(t) \cup \text{var}(s)$. Then for all substitution $\sigma$, if $s \sigma \leq \text{dom}(t \sigma)$ and $\gamma \notin \text{dom}(\sigma)$, then there exists $\sigma'$ such that $\sigma' \not\simeq \sigma$ and $t(\sigma \cup \sigma') \leq (s \rightarrow \gamma)(\sigma \cup \sigma')$.

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

\[
\begin{align*}
(t(\sigma \cup \sigma')) &= t \sigma \\
&\leq s \sigma \rightarrow ((t \sigma) \cdot (s \sigma)) \\
&= s \sigma \rightarrow \gamma \sigma' \\
&\Rightarrow (s \rightarrow \gamma)(\sigma \cup \sigma')
\end{align*}
\]

Note that the solution of the $\gamma$ introduced in the constraint $(t, \leq, s \rightarrow \gamma)$ 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 $\gamma$ (which occurs in a covariant position) the minimum type for that solution. So the minimum solutions for $\gamma$ are in $t \bullet s$ (see the substitution $\sigma'(\gamma) = (t \sigma) \cdot (s \sigma)$ in the proof of Lemma C.50).

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


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


C.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 dovetail 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 C.2.1 and apply the tallying solving algorithm described in Section C.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 C.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 dovetail 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 C.53). For example, the function $\text{map}$ can not be applied to any integer, as the normalization of $\{(\text{Int}, \leq, \alpha \rightarrow \beta)\}$ is empty (and even for any expansion of $\alpha \rightarrow \beta$). 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 $((\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool})) \rightarrow t$ to an argument of type $\alpha \rightarrow \alpha$. We start with the constraint

\[
(\alpha \rightarrow \alpha, \leq, (\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool})).
\]

The tallying algorithm first normalizes it into the set

\[
\{(\alpha, \leq, \text{Int}), (\alpha, \geq, \text{Int}), (\alpha, \leq, \text{Bool}), (\alpha, \geq, \text{Bool})\}\ (\text{Step 1}).
\]

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

\[
((\alpha_1 \rightarrow \alpha_1) \land (\alpha_2 \rightarrow \alpha_2), \leq, (\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool})),
\]

and one of the constraint-set of its normalization is

\[
\{(\alpha_1, \leq, \text{Int}), (\alpha_1, \geq, \text{Int}), (\alpha_2, \leq, \text{Bool}), (\alpha_2, \geq, \text{Bool})\}
\]

The conflict between $\text{Int}$ and $\text{Bool}$ 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 C.53. Let \( t, s \) be two types, \( \gamma \) a fresh type variable and \( \{ \rho_i \}_{i \in I}, \{ \rho_j \}_{j \in J} \) two sets of general renamings. If \( \emptyset \vdash \{(t, \leq 0 \rightarrow \top), (t, \leq s \rightarrow \gamma)\} \rightarrow \emptyset \), then \( \emptyset \vdash \{(\bigwedge_{i \in I} t \rho_i, \leq 0 \rightarrow \top), (\bigwedge_{j \in J} t \rho_j, \leq (\bigwedge_{j \in J} s \rho_j) \rightarrow \gamma)\} \rightarrow \emptyset \).

Proof. As \( \emptyset \vdash \{(t, \leq 0 \rightarrow \top), (t, \leq s \rightarrow \gamma)\} \rightarrow \emptyset \), then either \( \emptyset \vdash \{(t, \leq 0 \rightarrow \top)\} \rightarrow \emptyset \) or \( \emptyset \vdash \{(t, \leq s \rightarrow \gamma)\} \rightarrow \emptyset \). If the first one holds, then according to Lemma C.19, we have \( \emptyset \vdash \{(\bigwedge_{i \in I} t \rho_i, \leq 0 \rightarrow \top)\} \rightarrow \emptyset \), and a fortiori
\[
\emptyset \vdash \{(\bigwedge_{i \in I} t \rho_i, \leq 0 \rightarrow \top), (\bigwedge_{i \in I} \rho_i, \leq (\bigwedge_{j \in J} s \rho_j) \rightarrow \gamma)\} \rightarrow \emptyset
\]
Assume that \( \emptyset \vdash \{(t, \leq s \rightarrow \gamma)\} \rightarrow \emptyset \). Without loss of generality, we consider the disjunctive normal form \( \tau \) of \( t \):
\[
\tau = \bigvee_{k \in K_k} \tau_{k_b} \vee \bigvee_{k \in K_p} \tau_{k_p} \vee \bigvee_{k \in K_a} \tau_{k_a}
\]
where \( \tau_{k_b} \) (\( \tau_{k_p} \) and \( \tau_{k_a} \) resp.) is an intersection of basic types (products and arrows resp.) and type variables. Then there must exist \( k \in K_k \cup K_p \cup K_a \) such that \( \emptyset \vdash \{(\tau_{k}, \leq 0 \rightarrow \top)\} \rightarrow \emptyset \). If \( k \in K_k \cup K_p \), then the constraint \( (\tau_{k}, \leq s \rightarrow \gamma) \) is equivalent to \( (\tau_{k_a}, \leq 0) \). By Lemma C.19, we get
\[
\emptyset \vdash \{(\bigwedge_{i \in I} \tau_{k_i} \rho_i, \leq 0)\} \rightarrow \emptyset,
\]
that is, \( \emptyset \vdash \{(\bigwedge_{i \in I} \tau_{k_i} \rho_i, \leq (\bigwedge_{j \in J} s \rho_j) \rightarrow \gamma)\} \rightarrow \emptyset \). So the result follows.

Otherwise, it must be that \( k \in K_a \) and \( \tau_k = \bigwedge_{p \in P}(w_p \rightarrow v_p) \land \bigwedge_{n \in N} \neg(w_n \rightarrow v_n) \). We claim that \( \emptyset \vdash \{(\tau_k, \leq 0)\} \rightarrow \emptyset \) (otherwise, \( \emptyset \vdash \{(\tau_{k_a}, \leq s \rightarrow \gamma)\} \rightarrow \emptyset \) does not hold). Applying Lemma C.19 again, we get \( \emptyset \vdash \{(\bigwedge_{i \in I} \tau_{k_i} \rho_i, \leq 0)\} \rightarrow \emptyset \). Moreover, following the rule (NARROW), there exists a set \( P' \subseteq P \) such that
\[
\emptyset \vdash \{(\bigwedge_{p \in P'} (\neg w_p \land s, \leq 0)\} \rightarrow \emptyset
\]
Applying C.19, we get
\[
\emptyset \vdash \{(\bigwedge_{i \in I} \bigvee_{p \in P'} (\neg w_p \land s, \leq 0)\} \rightarrow \emptyset
\]
By the rule (NARROW), we have
\[
\emptyset \vdash \{(\bigwedge_{i \in I} \bigvee_{p \in P'} (w_p \land \neg \gamma, \leq 0)\} \rightarrow \emptyset
\]
Therefore, we have \( \emptyset \vdash \{(\bigwedge_{i \in I} (w_i \rightarrow v_i) \rho_i, \leq 0, (\bigwedge_{j \in J} s \rho_j) \rightarrow \gamma)\} \rightarrow \emptyset \). So the result follows.

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

Theorem C.54. Let \( t, s \) be two types and \( \gamma \) the special fresh type variable introduced in \( (\bigwedge_{i \in I} t \sigma_i, \leq (\bigwedge_{j \in J} s \sigma_j) \rightarrow \gamma) \). If \( \text{App}_\Delta(t, s) \) terminates with \( \Theta \), then
(1) (Soundness) if \( \Theta \neq \emptyset \), then for each \( \sigma \in \Theta \), \( \sigma(\gamma) \in t \bullet \Delta \).
(2) (Weak completeness) if \( \Theta = \emptyset \), then \( t \bullet \Delta \mathbf{s} = \emptyset \).

(2): consequence of Lemma C.53.
where $\gamma$ is a fresh type variable (and where we $\alpha$-converted the type of map). Then the algorithm $\text{Sol}_\Delta(C_1)$ generates a set of eight constraint-sets at Step 2:

$$
\begin{align*}
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0 \} \\
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \} \\
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0, \beta_1 \geq \text{Int} \} \\
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \vee (\alpha \setminus \text{Int}) \} \\
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0, \beta_1 \geq \text{Int} \cap, \beta_1 \geq \text{Int} \} \\
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \cap (\alpha \setminus \text{Int}) \} \\
\{ \gamma \geq \alpha_1 \rightarrow \beta_1, \alpha_1 \leq 0, \beta_1 \geq \text{Int} \cap, \beta_1 \geq (\alpha \setminus \text{Int}) \} \\
\end{align*}
$$

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 $\gamma$ (i.e., the result types of $\text{map even}$: $[] \rightarrow [],$ or $\text{Int} \rightarrow \text{Bool},$ or $\text{Int} \setminus \text{Int} \rightarrow \text{Int},$ or $\text{Int} \rightarrow \text{Bool} \vee (\alpha \setminus \text{Int})$). 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

$$
(\text{Int} \setminus \text{Int}) \rightarrow (\text{Int} \cap \text{Int}) \wedge ((\alpha \setminus \text{Int}) \rightarrow (\text{Int} \cap \text{Int}))
$$

(20)

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

$$
\begin{align*}
\{(\alpha_1 \rightarrow \beta_1) \rightarrow [\alpha_1 \rightarrow \beta_1] \wedge ((\alpha_2 \rightarrow \beta_2) \rightarrow [\alpha_2 \rightarrow \beta_2]) \\
\leq ((\text{Int} \rightarrow \text{Bool}) \wedge ((\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int})) \rightarrow \gamma)
\end{align*}
$$

from which we get the type (20) directly.

As stated in Section C.1, we chose an arbitrary ordering on type variables, which affects the generated substitutions and then the resulting types. Assume that $\sigma_1$ and $\sigma_2$ are two type substitutions generated by different orders. Thanks to the completeness of the tallying problem, there exist $\sigma_1$ and $\sigma_2$ such that $\sigma_2 \succeq \sigma_1 \circ \sigma_1$ and $\sigma_1 \succeq \sigma_2 \circ \sigma_2$. Therefore, the result types corresponding to $\sigma_1$ and $\sigma_2$ are equivalent under $\mathbb{C}$, that is $\sigma_1(\gamma) \subseteq \mathbb{C} \sigma_2(\gamma)$ and $\sigma_2(\gamma) \subseteq \mathbb{C} \sigma_1(\gamma)$. However, this does not imply that $\sigma_1(\gamma) \succeq \sigma_2(\gamma)$. For example, $\alpha \subseteq \mathbb{C} \beta$ and $\beta \subseteq \mathbb{C} \alpha$, but $\alpha \not\subseteq \mathbb{C} \beta$. 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. For example, consider the $\text{map even}$ again. The type (20) is obtained under the ordering $\alpha(\alpha) < \alpha(\beta_1) < \alpha(\alpha)$. While under the ordering $\alpha(\alpha) < \alpha(\alpha) < \alpha(\beta_1)$, we would instead get

$$
(\beta \setminus \text{Int}) \rightarrow [\beta] \wedge ((\text{Int} \rightarrow \text{Bool} \vee \beta) \rightarrow [\text{Bool} \vee \beta])
$$

(21)

It is clear that (20) $\subseteq$ (21) and (21) $\subseteq$ (20). However, compared with (20), (21) is less precise and less comprehensible, if we look at the type $\text{Int} \rightarrow \text{Bool} \vee \beta$ : (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 $\text{map even}$, 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 an argument of type $(\text{Int} \times \text{Bool}) \vee (\text{Bool} \times \text{Int})$. For this application our algorithm returns after one iteration the type $(\text{Int} \rightarrow \text{Bool}) \times (\text{Int} \rightarrow \text{Bool})$ (since it unifies $\alpha$ with $\beta$) while one further iteration allows the system to deduce the more precise type $(\text{Int} \times \text{Bool}) \vee (\text{Bool} \times \text{Int})$. 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).

### C.2.3 Heuristics to stop type-substitution inference

We only have a semi-algorithm for $\star s$ because, as long as we do not find a solution, we may increase the polarities 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 polarities 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: no solution is found when the polarities of $I$ and $J$ have reached these heuristic numbers, then the procedure stops returning failure. This yields a terminating algorithm for $\star s$ 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.

---

14 In the current implementation we assume that the type variables in the function type always have smaller orders than those in the argument type.
Through some examples, we first analyze the reasons why one needs to expand the function type and/or the argument type: 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., \( \emptyset \rightarrow \emptyset \)) 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 \( \subseteq \)), 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) unisons in \( s \). Generally, it is better to have as many copies of \( t \) as there are disjunctions in \( s \). Consider the example,

\[
\begin{align*}
t &= (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\
&= (\text{Int} \rightarrow \text{Int}) \lor (\text{Bool} \rightarrow \text{Bool})
\end{align*}
\]  

(22)

If we do not expand \( t \) (i.e., if \( p = 1 \)), then the result type computed for the application of \( t \) to \( s \) is \( \emptyset \rightarrow \emptyset \). However, this result type cannot be applied hereafter, since its domain is \( \emptyset \), and is therefore useless (more precisely, it can be applied only to expressions that are provably diverging). When \( p = 2 \), we get an extra result type, \((\text{Int} \rightarrow \text{Int}) \lor (\text{Bool} \rightarrow \text{Bool})\), which is obtained by instantiating \( t \) twice, by \( \text{Int} \) and \( \text{Bool} \) 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:

\[
\begin{align*}
t &= (\alpha \times \beta) \rightarrow (\beta \times \alpha) \\
&= (\text{Int} \times \text{Bool}) \lor (\text{Bool} \times \text{Int})
\end{align*}
\]  

(23)

Without expansion, the result type is \(((\text{Int} \lor \text{Bool}) \times (\text{Bool} \lor \text{Int})) \lor \alpha \) unifies \( \text{Int} \) and \( \text{Bool} \). If we expand \( t \), there exists a more precise result type \((\text{Int} \times \text{Bool}) \lor (\text{Bool} \times \text{Int}) \), each summand of which 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 \(((\text{Int} \lor \text{Bool}) \times \beta) \lor ((\text{Int} \lor \text{Bool}) \times \alpha) \lor ((\text{Int} \lor \text{Bool}) \times \alpha) \lor ((\text{Int} \lor \text{Bool}) \lor \alpha) \). Further expanding \( t \) makes more product decompositions possible, which may in turn generate new result types. However, the type \((\text{Int} \times \text{Bool}) \lor (\text{Bool} \times \text{Int})\) 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

\[
\begin{align*}
t &= \alpha \rightarrow (\alpha \times \alpha) \\
&= \text{Int}
\end{align*}
\]  

(24)

Without expansion, the result type is \( \gamma_1 = (\text{Int} \times \text{Int}) \). With two copies of \( t \), besides \( \gamma_1 \), we get another result type \( \gamma_2 = (\beta \times \beta) \lor (\text{Int} \lor \beta \lor \text{Int} \lor \beta) \), which is more general than \( \gamma_1 \) (e.g., \( \gamma_1 = \gamma_2[0/\beta] \)). Generally, with \( k \) copies, we get \( k \) result types of the form

\[
\gamma_k = (\beta_1 \times \beta_1) \lor \ldots \lor (\beta_{k-1} \times \beta_{k-1}) \lor (\text{Int} \lor \beta_1) \lor (\text{Int} \lor \beta_2) \lor (\text{Int} \lor \beta_3) \lor \ldots \lor (\text{Int} \lor \beta_{k-1})
\]

It is clear that \( \gamma_{k+1} \supseteq \gamma_k \). Moreover, it is easy to find two substitutions \([\sigma_1, \sigma_2]\) such that \([\sigma_1, \sigma_2] \models \gamma_k \supseteq \gamma_{k+1} (k \geq 2)\). Therefore, \( \gamma_2 \) is the minimum (with respect to \( \supseteq \)) of \( \{\gamma_k, k \geq 1\} \), so expanding \( t \) more than once is useless (we do not get a type more precise than \( \gamma_2 \)). However, we think the programmer expects \((\text{Int} \times \text{Int})\) as a result type instead of \( \gamma_2 \). So we take the heuristic number here as 1.

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

\[
\begin{align*}
t &= (\alpha \rightarrow \alpha) \times (\beta \rightarrow \beta) \\
&= ((\text{Even} \rightarrow \text{Even}) \lor (\text{Odd} \rightarrow \text{Odd})) \lor (\text{Bool} \rightarrow \text{Bool})
\end{align*}
\]  

(25)

For the application to succeed, we have a constraint generated for each component of the product type, namely \( (\alpha \rightarrow \alpha \geq (\text{Even} \rightarrow \text{Even}) \lor (\text{Odd} \rightarrow \text{Odd})) \) and \( (\beta \rightarrow \beta \geq \text{Bool} \lor \text{Bool}) \). As with Example (22), it is better to expand \( \alpha \rightarrow \alpha \) once for the first constraint, while there is no need to expand \( \beta \rightarrow \beta \) for the second one. As a result, we expand the whole type \( t \) once, and get the result type \(((\text{Bool} \lor \text{Bool}) \times ((\text{Even} \rightarrow \text{Even}) \lor (\text{Odd} \rightarrow \text{Odd})))\) 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.

\[
\begin{align*}
t &= (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
&= (\text{Int} \rightarrow \text{Bool}) \lor (\text{Int} \rightarrow \text{Int})
\end{align*}
\]  

(26)

When \( p = 1 \), the constraint to solve is \( \alpha \rightarrow \beta \geq s \). As stated in Subsection C.2.2, we get four possible result types: \([\ ] \rightarrow [\ ]\), \([\text{Int}] \rightarrow [\text{Bool}]\), \([\alpha \rightarrow \text{Int}] \rightarrow [\alpha \lor \text{Int}]\), or \([\text{Int} \lor \alpha] \rightarrow [\text{Bool} \lor (\alpha \lor \text{Int})]\), 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 it. Then we would have to solve either \((\bigvee_{i=1..q} \alpha_i \rightarrow \beta_i \geq s)\) or \((\bigwedge_{i=1..p} \alpha_i \rightarrow \beta_i \geq s)\). For the first constraint to hold, by the decomposition rule of arrows, there exists \(i_0\) such that \(s \leq \alpha_{i_0} \rightarrow \beta_{i_0}\), which is the same constraint as for \(p = 1\). The second constraint implies \(s \leq \alpha_i \rightarrow \beta_i\) 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

\[
\begin{align*}
t & = (\text{true} \times (\text{Int} \rightarrow \alpha)) \rightarrow t_1) \land ((\text{false} \times (\alpha \rightarrow \text{Bool})) \rightarrow t_2) \\
& = (\text{Bool} \times (\text{Int} \rightarrow \text{Bool}))
\end{align*}
\]

Here \(\text{dom}(t)\) is \(\text{true} \times (\text{Int} \rightarrow \alpha) \lor (\text{false} \times (\alpha \rightarrow \text{Bool}))\). The type \(s\) cannot be completely contained in either summand of \(\text{dom}(t)\), but it can be contained in \(\text{dom}(t)\). Indeed, the first summand requires the substitution of \(\alpha\) to be a supertype of \(\text{Bool}\) while the second one requires it to be a subtype of \(\text{Int}\). As \(\text{Bool}\) is not a subtype of \(\text{Int}\), 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 \(\text{dom}(\bigwedge_{i \in I} t_i) = \bigvee_{i \in I} \text{dom}(t_i)\), so that the tallying problem for the application becomes \(\bigwedge_{i \in I} s\sigma_i^j \leq \bigvee_{i \in I} \text{dom}(t_i)\). We now give some heuristic numbers for \(|/|\) depending on \(\text{dom}(t)\).

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

\[
\begin{align*}
\text{dom}(t) & = (\text{Int} \rightarrow ((\text{Bool} \rightarrow \text{Bool}) \land (\text{Int} \rightarrow \text{Int}))) \\
& \lor (\text{Bool} \rightarrow ((\text{Bool} \rightarrow \text{Bool}) \land (\text{Int} \rightarrow \text{Int}) \land (\text{Real} \rightarrow \text{Real})))
\end{align*}
\]

For the application to succeed, we need to expand \(\text{Int} \rightarrow (\alpha \rightarrow \alpha)\) with two copies (so that we can make two distinct instantiations \(\alpha = \text{Bool}\) and \(\alpha = \text{Int}\)) and \(\text{Bool} \rightarrow (\beta \rightarrow \beta)\) with three copies (for three instantiations \(\beta = \text{Bool}, \beta = \text{Int}, \text{and} \beta = \text{Real}\), corresponding to the first and the second summand in \(\text{dom}(t)\)) 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: \{\(\alpha = \text{Bool}, \beta = \text{Bool}\), \(\{\alpha = \text{Bool}, \beta = \text{Int}\), \(\{\alpha = \text{Bool}, \beta = \text{Real}\), \(\{\alpha = \text{Int}, \beta = \text{Bool}\), \(\{\alpha = \text{Int}, \beta = \text{Int}\), and \(\{\alpha = \text{Int}, \beta = \text{Real}\}\}, which are the Cartesian products of the substitutions for \(\alpha\) and \(\beta\).

If \(\text{dom}(t)\) 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,

\[
\begin{align*}
\text{dom}(t) & = (\text{Int} \rightarrow \text{Int}) \times (\text{Bool} \rightarrow \text{Bool}) \\
& = (\alpha \rightarrow \alpha) \times (\alpha \rightarrow \alpha)
\end{align*}
\]

It is easy to infer that the substitution required by the left component needs \(\alpha\) to be \(\text{Int}\), while the one required by the right component needs \(\alpha\) to be \(\text{Bool}\). Thus, we need to expand \(s\) at least once. Assume that \(s = (s_1 \times s_2)\) and we need \(q_1\) copies of \(s_1\) with the type substitutions: \(\sigma_1^1, \ldots, \sigma_1^{q_1}\). Generally, we can expand the whole product type so that we get \(s_1 \times s_2\) copies as follows:

\[
\begin{align*}
\lambda_{j=1..q_1} (s_1 \times s_2) & \sigma_j^1 \land \lambda_{j=1..q_2} (s_1 \times s_2) \sigma_j^2 \\
& = ((\lambda_{j=1..q_1} s_1 \sigma_j^1) \land (\lambda_{j=1..q_2} s_1 \sigma_j^2)) \times (\lambda_{j=1..q_1} s_2 \sigma_j^1) \land (\lambda_{j=1..q_2} s_2 \sigma_j^2)
\end{align*}
\]

Clearly, this expansion type is a subtype of \((\lambda_{j=1..q_1} s_1 \sigma_j^1 \times \lambda_{j=1..q_2} s_2 \sigma_j^2)\) and so the type tallying succeeds.

Next, consider the case where \(\text{dom}(t)\) is an intersection of arrows:

\[
\begin{align*}
\text{dom}(t) & = (\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool}) \\
& = \alpha \rightarrow \alpha
\end{align*}
\]
Numbers succeed. Because (arrow type and contains a union in its domain. Therefore, we have to look into and then use subsumption to obtain the desired result. Likewise, we may have to expand $\alpha$ case that the expansions of $t$ sharp of $\sigma$ and thus a supertype of $t$ by the expansion $(\sigma t_1 \land \sigma t_2)$. Following these intuitions, we define in Table 2 a heuristic number $H_q(\text{dom}(t))$ so far.

Let us consider each summand in $t$ separately. Here we informally discuss the relationship between the two, and hint as why decidability is difficult to prove.

Let $\text{dom}(t) = t_1 \lor t_2$, $s = s_1 \lor s_2$, and suppose the type tallying between $\text{dom}(t)$ and $s$ requires that $t_i \sigma_i \geq s_i$, where $\sigma_1$ and $\sigma_2$ are two conflicting type substitutions. Then we can simply expand $\text{dom}(t)$ with $\sigma_1$ and $\sigma_2$, yielding $t_1 \sigma_1 \lor t_2 \sigma_1 \lor t_1 \sigma_2 \lor t_2 \sigma_2$. Clearly, this expansion type is a supertype of $t_1 \sigma_1 \lor t_2 \sigma_2$ 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., $t_2 \sigma_1 \lor t_1 \sigma_2$) 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 $s = s_1 \lor s_2$ and $s_i$ requires a different substitution $\sigma_i$ (i.e., $s_i \sigma_i \leq \text{dom}(t)$) and $\sigma_1$ is in conflict with $\sigma_2$. If we expand $s$ with $\sigma_1$ and $\sigma_2$, then we have

$$(s_1 \lor s_2) \sigma_1 \land (s_1 \lor s_2) \sigma_2 = (s_1 \sigma_1 \lor s_1 \sigma_2) \lor (s_2 \sigma_1 \lor s_1 \sigma_2) \lor (s_2 \sigma_1 \lor s_2 \sigma_2)$$

It is clear that $s_1 \sigma_1 \land s_1 \sigma_2$, $s_1 \sigma_1 \land s_2 \sigma_2$ and $s_2 \sigma_1 \land s_2 \sigma_2$ are subtypes of $\text{dom}(t)$. Consider the extra type $s_1 \sigma_2 \land s_2 \sigma_1$. 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 $\text{dom}(t)$, then the type tallying succeeds. Otherwise, in some sense, we need to solve another type tallying between $s \land (s_2 \sigma_1 \lor s_1 \sigma_2)$ and $\text{dom}(t)$, 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 $\bullet \alpha$) so far.

To illustrate this phenomenon, consider the following example:

$$\text{dom}(t) = ((\text{Bool} \rightarrow \text{Bool}) \rightarrow (\text{Int} \rightarrow \text{Int}))$$

$$\lor ((\text{Bool} \rightarrow \text{Bool}) \lor (\text{Int} \rightarrow \text{Int})) \rightarrow ((\beta \lor \beta) \lor (\text{Finite} \lor \text{Fin}))$$

$$\lor (\beta \lor \beta)$$

$$\lor (\alpha \rightarrow (\text{Int} \lor \text{Int}))$$

$$\lor (\text{Finite} \rightarrow (\text{Finite} \lor \text{Finite}))$$

$$\lor (\text{Finite} \rightarrow (\text{Finite} \lor \text{Finite}))$$

$$\lor (\text{Finite} \rightarrow (\text{Finite} \lor \text{Finite}))$$

$$\lor (\text{Finite} \rightarrow (\text{Finite} \lor \text{Finite}))$$

Let us consider each summand in $t$ respectively. A solution for the first summand is $\alpha \geq \text{Boolean} \lor \text{Boolean}$, which corresponds to the first summand in $\text{dom}(t)$. The second one requires $\alpha \leq \text{Int} \lor \text{Int}$ and the third one $\beta \geq \text{Boolean}$. Since $\text{Boolean} \lor \text{Boolean}$ is not subtype of $(\text{Int} \lor \text{Int})$, we need to expand $s$ once, that is,

$$\text{dom}(t) = ((\text{Bool} \lor \text{Bool}) \lor (\text{Int} \lor \text{Int})) \lor ((\text{Int} \lor \text{Int}) \lor (\text{Int} \lor \text{Int}))$$

Thus $\text{dom}(t)$, which is the desired result.

Almost all the summands of $s'$ are contained in $\text{dom}(t)$ except the extra type

$$(\text{Bool} \lor \text{Bool}) \lor (\text{Int} \lor \text{Int})$$

That is to say, the expansion of $s$ would not cause the expansion of $t$.

To illustrate this phenomenon, consider the following example:

$$\text{dom}(t) = ((\text{Bool} \lor \text{Bool}) \lor (\text{Int} \lor \text{Int}))$$

$$\lor ((\text{Bool} \lor \text{Bool}) \lor (\text{Int} \lor \text{Int})) \lor ((\text{Int} \lor \text{Int}) \lor (\text{Int} \lor \text{Int}))$$

$$\lor ((\text{Int} \lor \text{Int}) \lor (\text{Int} \lor \text{Int}))$$

Thus $\text{dom}(t)$, which is the desired result.
Therefore, we need to consider another type tallying involving this extra type and \( \text{dom}(t) \). By doing so, we obtain \( \beta = \text{Int} \); however we have inferred before that \( \beta \) should be a supertype of \( \text{Bool} \). Consequently, we need to expand \( \text{dom}(t) \); the expansion of \( \text{dom}(t) \) with \( \text{Int}/\beta \) 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) \( \text{dom}(t) \), and we do not have to expand \( \text{dom}(t) \) or \( s \) again. The heuristic numbers we gave seem to be enough in practice.

### D. Type reconstruction

We define an implicit calculus without interfaces, for which we define a reconstruction system.

**Definition D.1.** An implicit expression \( m \) is an expression without any interfaces (or type substitutions). It is inductively generated by the following grammar:

\[
\begin{align*}
  m & ::= \text{c} \mid x \mid (m, m) \mid \pi_i(m) \mid m m \mid \lambda x.m \mid m \in \text{t} \ ? m : m
\end{align*}
\]

The type reconstruction for expressions has the form \( \Gamma \vdash \_ c : t \rightarrow \mathcal{S} \), which states that under the typing environment \( \Gamma \), \( c \) has type \( t \) if there exists at least one constraint-set \( C \) in the set of constraint-sets \( \mathcal{S} \) such that \( C \) are satisfied. The type reconstruction rules are given in Figure 11.

**Figure 11.** Type reconstruction rules

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 ((\text{RECON-CASE})): (i) if no branch is selected, then the type \( t_0 \) inferred for the argument \( m_0 \) should be \( \emptyset \) (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 \( \alpha \) 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 \( \alpha \) 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 
\( \alpha \) 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].

Let $m$ be an implicit expression such that $\Gamma \vdash m : t \leadsto \mathcal{S}$. By inserting $m$ those types form of 
\( \alpha \to \beta \) introduced by the derivation of $\Gamma \vdash m : t \leadsto \mathcal{S}$ for the \( \lambda \)-abstractions in $m$ correspondingly, we 
achieve an explicit expression $e$ for $m$, denoted as $\text{insert}(m)$. In particular, for a \( \lambda \)-abstraction $\lambda x. \ m$, we have

\[
\text{insert}(\lambda x. \ m) = \lambda^{\alpha \to \beta} x. \text{insert}(m)
\]

where $\alpha \to \beta$ is a fresh type introduced for $\lambda x. \ m$.

**Theorem D.2 (Soundness).** Let $m$ be an implicit expression such that $\Gamma \vdash m : t \leadsto \mathcal{S}$. Then for all 
$C \in \mathcal{S}$ and for all $\sigma$, if $\sigma \vdash C$, then $\exists \Gamma \sigma \vdash \text{insert}(m)[\sigma] : t\sigma$.

**Proof.** By induction on the derivation of $\Gamma \vdash m : t \leadsto \mathcal{S}$. We proceed by a case analysis of the last rule 
used in the derivation.

(\text{RECON-CONST}): straightforward .

(\text{RECON-PAIR}): straightforward.

(\text{RECON-PROJ}): consider the following derivation:

\[
\Gamma \vdash m_1 : t_1 \leadsto \mathcal{S}_1 \quad \Gamma \vdash m_2 : t_2 \leadsto \mathcal{S}_2 \\
\Gamma \vdash (m_1, m_2) : t_1 \times t_2 \leadsto \mathcal{S}_1 \cap \mathcal{S}_2
\]

Since $C \in \mathcal{S}_1 \cap \mathcal{S}_2$, according to Definition C.4, there exists $C_1 \in \mathcal{S}_1$ and $C_2 \in \mathcal{S}_2$ such that $C = C_1 \cup C_2$. Thus, we have $\sigma \vdash C_1$ and $\sigma \vdash C_2$. By induction, we have $\exists \Gamma \sigma \vdash \text{insert}(m_1)[\sigma] : t_1\sigma$ and $\exists \Gamma \sigma \vdash \text{insert}(m_2)[\sigma] : t_2\sigma$. By (pair), we get $\exists \Gamma \sigma \vdash (\text{insert}(m_1)[\sigma], \text{insert}(m_2)[\sigma]) : (t_1\sigma \times t_2\sigma)$, that is $\exists \Gamma \sigma \vdash \text{insert}((m_1, m_2))[\sigma] : (t_1 \times t_2)\sigma$.

(\text{RECON-APPL}): consider the following derivation:

\[
\Gamma \vdash m_1 : t_1 \leadsto \mathcal{S}_1 \\
\Gamma \vdash m_2 : t_2 \leadsto \mathcal{S}_2 \\
\Gamma \vdash \lambda x. m : \alpha \to \beta \leadsto \mathcal{S}_1 \cap \mathcal{S}_2 \cap \{ \{ (t_1, \leq, t_2 \to \alpha) \} \}
\]

According to Definition C.4, there exists $C \in \mathcal{S}$ such that $C = C \cap \{ (t_1, \leq, t_2 \to \alpha) \}$. Thus, we have $\sigma \vdash C$ and $t' \sigma \leq (t_1 \sigma \times t_2 \sigma)$. By induction, we have $\exists \Gamma \sigma \vdash \text{insert}(m_1)[\sigma] : t_1\sigma$ and $\exists \Gamma \sigma \vdash \text{insert}(m_2)[\sigma] : t_2\sigma$. By (app), we get $\exists \Gamma \sigma \vdash \text{insert}((m_1, m_2))[\sigma] : \alpha\sigma$, that is $\exists \Gamma \sigma \vdash \text{insert}(\text{insert}(m_1), \text{insert}(m_2))[\sigma] : \alpha\sigma$.

(\text{RECON-ABSTR}): consider the following derivation:

\[
\Gamma, (x : \alpha) \vdash m : t' \leadsto \mathcal{S}' \\
\Gamma \vdash \lambda x. m : \alpha \to \beta \leadsto \mathcal{S} \cap \{ \{ (t', \leq, \beta) \} \}
\]

According to Definition C.4, there exists $C \in \mathcal{S}$ such that $C \cap \{ (t', \leq, \beta) \}$. Thus, we 
have $\sigma \vdash C$ and $t' \sigma \leq \beta\sigma$. By induction, we have $\exists \Gamma \sigma, (x : \alpha) \vdash \text{insert}(m)[\sigma] : t'\sigma$. By subduction, 
we can get $\exists \Gamma \sigma \vdash \text{insert}(m)[\sigma] : t_1\sigma$ and $\exists \Gamma \sigma \vdash \text{insert}(m)[\sigma] : t_2\sigma$. By subduction, we can get $\exists \Gamma \sigma \vdash \text{insert}(m_1)[\sigma] : t_1\sigma \to \alpha\sigma$. Then by (app), we get $\exists \Gamma \sigma \vdash \text{insert}(m_1)[\sigma] \text{ insert}(m_2)[\sigma] : \alpha\sigma$, 
that is $\exists \Gamma \sigma \vdash \text{insert}(m_1, m_2)[\sigma] : \alpha\sigma$.

(\text{RECON-CASE}): consider the following derivation:

\[
\Gamma \vdash m_0 : t_0 \leadsto \mathcal{S}_0 \quad (m_0 \notin \mathcal{S}) \quad (m_0 \notin \mathcal{S}) \\
\Gamma \vdash m_1 : t_1 \leadsto \mathcal{S}_1 \\
\Gamma \vdash m_2 : t_2 \leadsto \mathcal{S}_2 \\
\mathcal{S} = (\mathcal{S}_0 \cap \{ \{ (t_0, \leq, \emptyset), (\emptyset, \leq, \alpha) \} \}) \\
\cup (\mathcal{S}_0 \cap \mathcal{S}_1 \cap \{ \{ (t_0, \leq, t'), (t_1, \leq, \alpha) \} \}) \\
\cup (\mathcal{S}_0 \cap \mathcal{S}_2 \cap \{ \{ (t_0, \leq, \emptyset), (t_2, \leq, \alpha) \} \}) \\
\cup (\mathcal{S}_0 \cap \mathcal{S}_1 \cap \mathcal{S}_2 \cap \{ \{ (t_0, \leq, \emptyset), (t_1 \lor t_2, \leq, \alpha) \} \})
\]

\[
\Gamma \vdash \text{insert}(m_0 \in t' \ ? m_1 : m_2) : \alpha \to \beta
\]
Since $C \in \mathcal{H}$, according to Definition C.4, there are four possible cases for $C$: (i) $C \in \mathcal{H}_0 \cap \{\{t_0, \leq, 0\}, \{(0, \leq, \alpha)\}\}$, (ii) $C \in \mathcal{H}_0 \cap \mathcal{H}_1 \cap \{\{(t_0, \leq, t'), (t_1, \leq, \alpha)\}\}$, (iii) $C \in \mathcal{H}_0 \cap \mathcal{H}_1 \cap \{\{(t_0, \leq, -t'), (t_2, \leq, \alpha)\}\}$, and (iv) $C \in \mathcal{H}_0 \cap \mathcal{H}_1 \cap \mathcal{H}_2 \cap \{\{(t_0, \leq, 1), (t_1 \lor t_2, \leq, \alpha)\}\}$.

**Case (i):** there exists $C_0 \in \mathcal{H}_0$ such that $\sigma \vdash C_0$, $t_0 \sigma \leq 0$ and $0 \leq \alpha \sigma$. By induction, we have $0 \vdash \Gamma \vdash \text{insert}(m_0)@\sigma : t_0 \sigma$. Since $t_0 \sigma \leq 0$, we have $t_0 \sigma \leq -t'$ and $t_0 \sigma \leq t'$. Then applying the rule (case), we have $0 \vdash \Gamma \vdash \text{insert}(m')@\sigma@\sigma \vdash \text{insert}(m_1)@\sigma : t_1 \sigma$, that is, $0 \vdash \Gamma \vdash \text{insert}(m')@\sigma : t_1 \sigma$. Finally, by subsumption, the result follows.

**Case (ii):** there exists $C_0 \in \mathcal{H}_0$ and $C_1 \in \mathcal{H}_1$ 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 $0 \vdash \Gamma \vdash \text{insert}(m_0)@\sigma : t_0 \sigma$ and $0 \vdash \Gamma \vdash \text{insert}(m_1)@\sigma : t_1 \sigma$. If $t_0 \sigma \leq t'$, then $t_0 \sigma \leq t' \land (-t') \approx 0$ (i.e., Case (i)), and thus the result follows by subsumption. Otherwise, we have $t_0 \sigma \leq -t'$. Then applying the rule (case), we have $0 \vdash \Gamma \vdash \text{insert}(m')@\sigma : t_1 \sigma$. Finally, by subsumption, the result follows.

**Case (iii):** similar to Case (ii).

**Case (iv):** there exists $C_0 \in \mathcal{H}_0, C_1 \in \mathcal{H}_1$ and $C_2 \in \mathcal{H}_2$ such that $\sigma \vdash C_0$, $\sigma \vdash C_1$, $\sigma \vdash C_2$ and $t_0 \sigma \lor t_1 \sigma \lor t_2 \sigma \leq \alpha \sigma$. By induction, we have $0 \vdash \Gamma \vdash \text{insert}(m_0)@\sigma : t_0 \sigma$, $0 \vdash \Gamma \vdash \text{insert}(m_1)@\sigma : t_1 \sigma$ and $0 \vdash \Gamma \vdash \text{insert}(m_2)@\sigma : t_2 \sigma$. By subsumption, we have $0 \vdash \Gamma \vdash \text{insert}(m_1)@\sigma : t_1 \sigma \lor t_2 \sigma$ and $0 \vdash \Gamma \vdash \text{insert}(m_2)@\sigma : t_1 \sigma \lor t_2 \sigma$. If $t_0 \sigma \leq t'$ or $t_0 \sigma \leq -t'$, then we are in Case (i) – (iii), thus the result follows by subsumption. Otherwise, applying the rule (case), we have $0 \vdash \Gamma \vdash \text{insert}(m')@\sigma@\sigma \vdash \text{insert}(m_1)@\sigma : t_1 \sigma \lor t_2 \sigma$.

(Reconc-case-var): similar to (Reconc-case). \(\square\)

Consider the implicit version of map, which can be defined as:

\[
\mu m \lambda f . \lambda l . l \in \text{nil} ? \text{nil} : (f(\pi_1 l), m f(\pi_2 l))
\]

The type inferred for map by the type reconstruction system is $\alpha_1 \rightarrow \alpha_2$ and the generated set $\mathcal{H}$ of constraint-sets is:

\[
\left\{\begin{array}{ll}
\{ \alpha_3 \rightarrow \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_1, \alpha_3 \leq 0, 0 \leq \alpha_5 \}, \\
\{ \alpha_3 \rightarrow \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_1, \alpha_3 \leq \text{nil}, \alpha_5 \leq \alpha_3 \}, \\
\{ \alpha_3 \rightarrow \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_1, \alpha_4 \leq \text{nil}, \alpha_6 \leq \alpha_0 \}, \\
\{ \alpha_3 \rightarrow \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_1, \alpha_4 \leq 1, \alpha_6 \leq \alpha_0 \} \cup C,
\end{array}\right.
\]

where $C$ is $\{\alpha_1 \leq \alpha_7 \rightarrow \alpha_6, \alpha_3 \leq \{\alpha_7 \times \alpha_8\}, \alpha_2 \ldots \}$.

Then applying the tallying algorithm to the sets, we get the following types for map:

\[
\begin{align*}
0 & \rightarrow (0 \rightarrow \alpha_3) \\
\text{nil} & \rightarrow (\text{nil} \rightarrow \text{nil}) \\
(0 \rightarrow 1) & \rightarrow (0 \rightarrow 0) \\
(0 \rightarrow 0) & \rightarrow (0 \rightarrow \alpha_5) \\
0 & \rightarrow (0 \rightarrow \alpha_5) \\
(0 \rightarrow \text{nil} \lor (1 \times 1)) & \rightarrow \text{nil} \\
(0 \rightarrow 0) & \rightarrow (\text{nil} \rightarrow \text{nil}) \\
(\alpha \rightarrow \beta) & \rightarrow (\alpha \rightarrow [\beta])
\end{align*}
\]

By replacing type variables that only occur positively by $0$ and those only occurring negatively by $1$, we obtain

\[
\begin{align*}
\text{id} & \rightarrow (0 \rightarrow 0) \\
\text{id} & \rightarrow (\text{nil} \rightarrow \text{nil}) \\
0 & \rightarrow ((1 \times 1) \rightarrow 0) \\
(0 \rightarrow 1) & \rightarrow (0 \rightarrow 0) \\
(1 \rightarrow \beta) & \rightarrow (0 \rightarrow [\beta]) \\
0 & \rightarrow (\text{nil} \lor (1 \times 1)) \rightarrow \text{nil} \\
(0 \rightarrow 1) & \rightarrow (\text{nil} \rightarrow \text{nil}) \\
(\alpha \rightarrow \beta) & \rightarrow ([\alpha] \rightarrow [\beta])
\end{align*}
\]

All the types, except the last two, are useless, as they provide no further information. Thus we deduce the following type for map:

\[
((\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta])) \land ((0 \rightarrow 1) \rightarrow (\text{nil} \rightarrow \text{nil}))
\]

15 These useless types are generated from the fact that $0 \rightarrow t$ contains all the functions, or the fact that $(0 \times t)$ or $(t \times 0)$ is a subtype of any type, or the fact that Case (i) in type-cases is useless in practice.
which is more precise than \((\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta])\) since it states that the application of map to any function and the empty list returns the empty list.

## E. Application to CDuce

We give a rough overview of the modifications that are necessary in order to transpose the algorithms and the results of this work to the implementation of the polymorphic extension of CDuce. In particular, we show how to generalize the static and dynamic semantics of explicit type-case expressions of this work to CDuce’s pattern matching expressions. Details about the syntax and semantics of CDuce can be found in [2] or, better, in the online documentation available at [www.cduce.org](http://www.cduce.org).

### E.1 Intermediate language

The CDuce compiler includes three different languages (source, typed, and lambda) that are refined one into the other in different passes of the compiler. The first language corresponds to parsed CDuce expressions and the last is closer to CDuce bytecode.

The source language is defined in the module `ast.ml` of the CDuce’s source distribution. It is the representation of the source code.

\[
a ::= c | x | aa | (\alpha, a) | \lambda x.p.a \text{ match } a \text{ with } p \rightarrow a | p \rightarrow a
\]  

The source language is composed of variables, constants, tuples, application of two expressions, lambda abstractions where \(p\) is a pattern, and match expressions. Patterns are defined as follows

\[
p ::= t | (p, p) | p\&p | p | x ::= c | x
\]

with types, tuples of patterns, intersection, union, constants, and capture variables (plus recursive patterns here omitted).

The typed language is the result of the type inference performed on the source language and it is defined in the module `typed.ml` in the source distribution of CDuce.

\[
e ::= c | x | x \in | ee | (e, e) | \lambda x.p.e \text{ match } e \text{ with } p_1 : \Xi_1 \rightarrow e_1 | p_2 : \Xi_2 \rightarrow e_2
\]  

The typed language is similar to the source language. The notable differences are the presence of polymorphic variables \(\Xi\) (cf. Part 1, §5.3 [3]), the application of substitutions to expressions \(\sigma\) and the \(\Xi\)'s associated to patterns in the match expressions. Each \(\Xi\) is a mapping from the capture variables of the pattern \(p_i\) to sets of type variable and will be used to compile away \(\lambda\)-polymorphic expression variables (compiling a source expression variable into a monomorphic variable is much less expensive in terms of run-time performance). Also since this language is the target of type-inference and we do not infer the decorations of lambdas, then \(\lambda x.p.e\) stands for \(\lambda x'p.e\).

The typed language is transformed in the intermediate language as result of the compilation step. The evaluation language is defined in the module `eval.ml`

\[
e ::= c | x | x \in | ee | (e, e) | \lambda x\&x.p.e \text{ match } e \text{ with } p_1 : \Xi_1 \rightarrow e_1 | p_2 : \Xi_2 \rightarrow e_2 | p_3 : \Xi_3 \rightarrow e_3
\]

In the intermediate (or compiled) language, (lazy) type-substitutions “\(\Sigma\)” are now associated to polymorphic variables and to polymorphic lambda expressions. The \(\Xi\) annotations present in the patterns of the match expressions are now removed since they were used to determine whether a variable \(x\) has to be compiled as \(x\); just as \(x\). The symbol \(\lambda\) is a compiler optimization that is explained at the end of Section 5.3 in Part I [3]. Notice that we added in lambda abstractions an explicit variable to capture the argument of the function. This is used to compile lambda-expressions with lazy type-substitutions (in particular \(\Xi1(x, t, \Sigma)\) defined right below); in the actual implementation these variables are nameless and compiled by reserving a special slot in the type-environment of closures. \(\Sigma\) ranges over expressions that denote sets of type substitutions.

\[
\Sigma ::= \sigma_1 | \text{comp}(\Sigma, \Sigma') | \text{sel}(x, t, \Sigma)
\]

we use \(\Sigma\) to denote the identity of these expressions that is the empty set of type substitutions. We use \(\text{dom}(\Sigma)\) to denote the domain \(\Sigma\). It is inductively defined as follows:

\[
\text{dom}(\sigma_1) \subseteq \sigma_i \rightarrow \bigcup_{\sigma_i \subseteq \sigma_1} \text{dom}(\sigma_i)
\]

\[
\text{dom}(\text{comp}(\Sigma, \Sigma')) = \text{dom}(\Sigma) \cup \text{dom}(\Sigma') \text{ (note: this is a rough approximation)}
\]

\[
\text{dom}(\text{sel}(x, t, \Sigma)) = \text{dom}(\Sigma)
\]

Note that \(\text{dom}(i) = \emptyset\). We use \(\text{var}(t)\) to denote the set of all type variables occurring in the type \(t\).

We use two containment relations. The first \(s \subseteq t\) is the semantic subtyping relation that states that for all substitutions \(s\) is a subtype of \(t\). The second \(s \subseteq \Delta\) \(t\) specifies that there exists a substitution for the variables not in \(\Delta\) (\(\Delta\) is a set containing all monomorphic variables) such that \(s \subseteq t\). We denote the set of all polymorphic variables as \(\Xi\).

### E.2 Type-directed translation

The CDuce compiler translates one internal language into another.
The translation from the language in (33) to the language in (34) is given contextually to the typing relation. In particular we extend the typing rules in order to prove judgments of the form $\Delta, \Gamma \vdash x : \tau$, where $\tau$ is a term of the source language (33) and $e$ its translation in the intermediate language (34).

The type inference rules that perform the translation from (33) to the language in (34) are specified in Figure 12. The rules (INF-VAR-MONO) translate variables into polymorphic or monomorphic ones according to whether their type contains polymorphic type variables or not. The rule for application, (INF-APPL) simply applies the sets of type-substitutions inferred for the function and for its argument to them. The rule for match (INF-MATCH) is the standard C# rule (see [11]) except that it stores in $\Xi_j$ the type variables occurring in the types of each capture variable of $p_j$. The rule (INF-ABSTR) is standard too, except that it merges the different $\Xi_j$’s and $\sigma_j$’s found for the same branch while checking the type for different arrows of the interface. Notice that these last two rules use the standard C# meta-operator “//” to compute the type environment for pattern’s capture variables (see [11]). Formally, let $t$ and $p$ be a type and a parameter such that $t \leq [p]$. We define $(t / p)(x) = \{(v/p)(x) \mid v \in t\}$, that is:

\[
\begin{align*}
    t / x & = \{x \mapsto t\}, \\
    t / t_0 & = \{\}\end{align*}
\]

\[
\begin{align*}
    t / (p_1 & \& p_2) = \pi_1(t) / p_1 \cup \pi_2(t) / p_2, \\
    (t / (p_1, p_2))(x) & = \left\{ \begin{array}{ll}
    (\pi_1(t) / p_1)(x) & \text{if } x \in \text{var}(p_1) \setminus \text{var}(p_2) \\
    (\pi_2(t) / p_2)(x) & \text{if } x \in \text{var}(p_2) \setminus \text{var}(p_1) \\
    \bigcup_{(t_1, t_2) \in \sigma_1} ((t_1 / p_1)(x), (t_2 / p_2)(x)) & \text{if } x \in \text{var}(p_1) \cap \text{var}(p_2) \\
    \end{array}\right. \\
    t / p_1 | p_2 & = (t \land \{p_1\}) / p_1 \cup (t \setminus \{p_1\}) / p_2, \\
    t / (x := e) & = \left\{ \begin{array}{ll}
    \{x \mapsto b\} & \text{if } t \not\leq 0 \\
    \{\} & \text{otherwise} \end{array}\right.
\]
\]

where the pairwise union of mappings assumes that the domains are distinct

\[
(\Gamma_1 \cup \Gamma_2)(x) = \left\{ \begin{array}{ll}
    \Gamma_1(x) & \text{if } x \in \text{dom}(\Gamma_1) \\
    \Gamma_2(x) & \text{if } x \in \text{dom}(\Gamma_2) \end{array}\right.
\]

\[
\begin{align*}
    \Delta_1 \vdash x : \Gamma(x) \triangleright \Delta & = \emptyset, \\
    \Delta_2 \vdash x : \Gamma(x) \triangleright \Delta & = \emptyset \\
    \Delta_1 \vdash x : \Gamma(x) \triangleright \Delta & = \emptyset \\
    \Delta_2 \vdash x : \Gamma(x) \triangleright \Delta & = \emptyset \\
    \Delta \vdash \cdot : \tau \triangleright \Delta & = \emptyset \\
    \Delta \vdash \cdot : \tau \triangleright \Delta & = \emptyset \\
\end{align*}
\]
Finally, the compilation of the explicitly-typed language (34) into the intermediate language (35) is given by the following rules:

\[ [x]_{\Sigma,E} = x \]
\[ [\xi]_{\Sigma,E} = \begin{cases} x & \text{if } \Xi(x) \cap \text{dom}(\Sigma) = \emptyset \6 \ x_{\Sigma} & \text{otherwise} \end{cases} \]
\[ [\lambda^p.e]_{\Sigma,E} = \begin{cases} \lambda^{x\in\text{dom}(\Sigma)}_{\xi_{\lambda^p.e}(x),\Sigma,E} & \text{if } \text{var}(t) \cap \text{dom}(\Sigma) = \emptyset \ (x \text{ fresh}) \6 \ \lambda^{x\in\text{dom}(\Sigma)}_{\xi_{\lambda^p.e}(x),\Sigma,E} & \text{otherwise} \end{cases} \]
\[ [(e_1,e_2)]_{\Sigma,E} = [(e_1)]_{\Sigma,E} \cdot [(e_2)]_{\Sigma,E} \]
\[ [e_1e_2]_{\Sigma,E} = [e_1]_{\Sigma,E} \cdot [e_2]_{\Sigma,E} \]
\[ [(\sigma \tau)]_{\Sigma,E} = [\sigma]_{\text{comp}(\Sigma, \sigma \tau)} \cdot [\tau]_{\Sigma,E} \]

[match e with \(p_1 : \Xi_1 \rightarrow e_1 \mid p_2 : \Xi_2 \rightarrow e_2\)]_{\Sigma,E} = [match \{e\}_{\Sigma,E} with \(p_1 \rightarrow [e_1]_{\Sigma,(\Xi \cup \Xi_1)} \mid p_2 \rightarrow [e_2]_{\Sigma,(\Xi \cup \Xi_2)}\)]

These rules are mostly straightforward except that we try to compile into monomorphic expression variables as many capture variables as possible. In particular, we compile as monomorphic also those polymorphic expression variables for which we can statically determine that type substitutions will have no effect at run-time (ie, every variable \(x\) for which \(\Xi(x) \cap \text{dom}(\Sigma) = \emptyset\) holds).

### E.3 Evaluation Rules

The evaluation procedure transforms the evaluation language into values of the following form:

\[ v ::= c \mid (v, v) \mid \lambda^{x,p.e,x} \mid (\lambda^{x,p.e,x}, v) \]

Notice that closures now include a slot for a variable. This slot stores the fresh variables that were introduced in the translations of lambdas and it is used at the application (rule (OE-APPLY)).

The operational semantics must be modified to take into account new constructions and to lazily propagate type substitutions for all constructed values.

\[ \text{(OE-CONST)} \]
\[ \text{(OE-CLOSURE)} \]
\[ \text{(OE-PAIRVALUE)} \]
\[ \text{(OE-VAR)} \]
\[ \text{(OE-PV-C)} \]
\[ \text{(OE-PV-IF)} \]
\[ \text{(OE-PV-P)} \]
\[ \text{(OE-PAIR)} \]
\[ \text{(OE-APPLY)} \]
\[ \text{(OE-MATCH 1)} \]
\[ \text{(OE-MATCH 2)} \]

Pattern matching is defined as follows:

\[ v/x = \{ x \mapsto v \} \]
\[ v/t = \begin{cases} \{ \} & \text{if } v \in_t t \6 \ \emptyset & \text{otherwise} \end{cases} \]
\[ \{v_1, v_2\}/(p_1, p_2) = v_1/p_1 \oplus v_2/p_2 \]
\[ (v_1, v_2)_{\Sigma}/(p_1, p_2) = v_1@\Sigma/p_1 \oplus v_2@\Sigma/p_2 \]
\[ v/p_1 & p_2 = v/p_1 \oplus v/p_2 \]
\[ v/p_1 | p_2 = \begin{cases} v/p_1 & \text{if } v/p_1 \neq \Omega \6 v/p_2 & \text{otherwise} \end{cases} \]
\[ v/(x := c) = \{ x \mapsto c \} \]

\(^{16}\) or \(\Sigma = \emptyset\) which is a special case of the condition (since \(\text{dom}(\emptyset) = \emptyset\)) that can be checked more easily.
where the $\oplus$ operator has the following definition ($\gamma$ ranges over value substitutions, ie mappings from expression variables to values):

$$
(\gamma_1 \oplus \gamma_2)(x) = \begin{cases} 
  \gamma_1(x) & \text{if } x \in \text{dom}(\gamma_1) \setminus \text{dom}(\gamma_2) \\
  \gamma_2(x) & \text{if } x \in \text{dom}(\gamma_2) \setminus \text{dom}(\gamma_1) \\
  (\gamma_1(x), \gamma_2(x)) & \text{if } x \in \text{dom}(\gamma_1) \cap \text{dom}(\gamma_2)
\end{cases}
$$

Notice that in the fourth rule of the definition of pattern matching when we deconstruct a pair that is annotated by a lazy type-substitution we do not immediately propagate the substitution to the sub-components. Instead we delay it until this substitution is needed. This is implemented by the “delay substitution” operation “@” defined as

$$
(e@\Sigma) = \begin{cases} 
  c & \text{if } c \in t \\
  (\lambda^e_{\Sigma} p.e, x, \varepsilon') @ \Sigma = (\lambda^e_{\text{comp}(\Sigma, \Sigma')} p.e, x, \varepsilon') \\
  (v_1, v_2) @ \Sigma = (v_1, v_2)\Sigma \\
  (v_1, v_2) @ \Sigma = (v_1, v_2)\text{comp}(\Sigma, \Sigma')
\end{cases}
$$

This requires a modification of the rules used to check the type of a value:

$$
\begin{align*}
  c @_t & \iff b @_t \\
  (\lambda^e_{\Sigma} p.e, x, \varepsilon') @_t & \iff \text{eval}(\varepsilon', \Sigma) \leq t \\
  (v_1, v_2) @_t & \iff v_1 @_t \pi_i(t), i \in 1, 2 \\
  (v_1, v_2) @_t & \iff v_1 @_t \in \text{dom}(\Sigma, \pi_i(t), i \in 1, 2)
\end{align*}
$$

where, we recall, the evaluation of the symbolic set of type-substitutions is inductively defined as

$$
\begin{align*}
  \text{eval}(\varepsilon', \sigma_1) = \sigma_1 \\
  \text{eval}(\varepsilon', \text{comp}(\Sigma, \Sigma')) = \text{eval}(\varepsilon', \Sigma) \circ \text{eval}(\varepsilon', \Sigma') \\
  \text{eval}(\varepsilon', \text{sel}(x, \lambda_{i \in I} t_i \rightarrow s_i, \Sigma)) = [\sigma_j \in \text{eval}(\varepsilon', \Sigma) \mid \exists i \in I : \varepsilon'(x) \in t_i, \sigma_j]
\end{align*}
$$

F. Experiments

To gauge the practicality of our local type inference algorithm, we performed extensive experiments, applying higher-order polymorphic function. To that end, we automatically generated function applications from the List module of the OCaml standard distribution. More specifically we considered the following functions:

1. `val length : 'a list -> int`
2. `val hd : 'a list -> 'a`
3. `val tl : 'a list -> 'a list`
4. `val nth : 'a list -> int -> 'a`
5. `val rev : 'a list -> 'a list`
6. `val append : 'a list -> 'a list -> 'a list`
7. `val rev_append : 'a list -> 'a list -> 'a list`
8. `val concat : 'a list list -> 'a list`
9. `val flatten : 'a list list -> 'a list`
10. `val iter : ('a -> unit) -> 'a list -> unit`
11. `val iteri : ('int -> 'a -> unit) -> 'a list -> unit`
12. `val mapi : ('int -> 'a -> 'b) -> 'a list -> 'b list`
13. `val rev_mapi : ('a -> 'b) -> 'a list -> 'b list`
14. `val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
15. `val fold_right : ('a -> 'b -> 'b) -> 'a -> 'b list -> 'b`
16. `val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit`
17. `val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`
18. `val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`
19. `val fold_left2 : ('a -> 'b -> 'c) -> 'a -> 'b list -> 'c list -> 'a`
20. `val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a -> 'b list -> 'c list -> 'c`
21. `val for_all : ('a -> bool) -> 'a list -> bool`
22. `val exists : ('a -> bool) -> 'a list -> bool`
23. `val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`
24. `val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`
25. `val mem : 'a -> 'a list -> bool`
26. `val memq : 'a -> 'a list -> bool`
27. `val mem_assoc : 'a -> ('a * 'b) list -> bool`
val mem_assq : 'a -> ('a * 'b) list -> bool
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
val merge : ('a -> 'a -> int) -> 'a list -> 'a list

We then devised a series of tests as follows. First, we generated all the applications that were well typed in
OCaml from one function against all the others. This gave, for instance, applications such as "map length"
or "map hd", that is performing local type inference for the applications
\((\alpha \to \beta) \to [\alpha] \to [\beta] \cdot \alpha \to Int\)
and
\((\alpha \to \beta) \to [\alpha] \to [\beta] \cdot [\alpha] \to \alpha\)
Then, for each function \(f\) of type \(t\) and \(f_1, \ldots, f_n\) of type \(t_1, \ldots, t_n\) such that all the applications "\(ff_1\)",
\(\ldots, "ff_n"\) are well-typed, we performed the local type inference on
\(t \cdot t_1 \land \ldots \land t_k\) for all \(k \leq n\)
Remark that these applications are well typed since a function of type
\(\alpha \to Int\) has also type \(\alpha\), and any of these type is in the domain of
\(\alpha\). Notice also that intersection of arrow types are never empty (all arrow types contain the type \(0 \to 0\).
Conversely, for all triple of functions \(f, f_1, f_2\) such as "\(ff_1\)" and "\(ff_2\)" are well-typed, we also typed
performed local type inference for \(t_1 \lor t_2\) • \(t\).
Lastly, we added to our test suite some ill-typed applications (such as \((\alpha \to \beta) \to [\alpha] \to [\beta] \cdot Int\)) to ensure
our implementation indeed detects these as invalid applications. Our test machine is an average laptop with
64bit Intel Core i3-2367M, 1.4Ghz, 4 cores and 8GB of RAM.
The results of our experiments are summarized in the following table:

<table>
<thead>
<tr>
<th># of tests</th>
<th>Total Time</th>
<th>Average Time</th>
<th>Median Time</th>
<th>Min. Time</th>
<th>Max. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 859</td>
<td>27s</td>
<td>14ms</td>
<td>2.1ms</td>
<td>0.1ms</td>
<td>2.090s</td>
</tr>
</tbody>
</table>

The worst time (2.09s) is the one for the local type inference of

\[
\text{map (length & hd & tl & nth & rev &}
\text{append & rev_append & concat & flatten &}
\text{iter & iteri & map & mapi & rev_map & fold_left)}
\]

As expected, the behavior of our algorithm is exponential (subtyping is already EXPTIME-complete,
although our implementation performs very well even for large types). We illustrate the general behavior of
our algorithm on two kinds of application. First, given a function of type \(t \to s\) where \(t\) does not contain
any arrows (but may contain products, sequences and so forth), local type inference scales linearly with the
sum of the size of types \(t \to s\) and \(u\), when computing \((t \to s) \cdot u\).

However (and as expected) if we consider types \(t \to s\) where \(t\) contains one, two, or more arrows, then
local type inference becomes exponential with respect to the size of the argument \(u\).
While these tests already represent cases that are unlikely to happen in practice (the worst time case features more than 45 connectives/constructors, namely 15 intersections and 30 arrows), we conjecture that standard optimization techniques (hash-consing, memoization, laziness) will make our semi-naive implementation even more tractable. During the experiment, memory usage was negligible (few megabytes).

Finally we also tested the type inference for applications of curried functions to several arguments. We added (by hand) to our test suite a set of functions that accept up to $n$ arguments. More precisely for each arity $n$ we added functions with the following types

\[
\begin{align*}
\alpha_1 & \rightarrow \cdots \rightarrow \alpha_n \rightarrow \text{Int} \\
\beta_1 & \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \beta_n \rightarrow \alpha_n
\end{align*}
\]

each of these functions, if its arity is $k$, was then applied to $k$ other randomly selected functions of this set. The test showed that our implementation can smoothly handle inference for the application of up to 20 arguments (for $n = 20$ the 120 tests take less than one second of cpu on a desktop workstation), then the exponential blowup becomes too important (in particular because of the memory footprint). The following table reports a sample of the cpu times for different $n$'s

<table>
<thead>
<tr>
<th>arity $n$</th>
<th># of tests</th>
<th>Total Time for all the tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>60</td>
<td>0m0.03s</td>
</tr>
<tr>
<td>15</td>
<td>90</td>
<td>0m0.27s</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>0m0.76s</td>
</tr>
<tr>
<td>25</td>
<td>150</td>
<td>2m39.68s</td>
</tr>
</tbody>
</table>

Consider that in the standard library of OCaml export all functions have at most 5 arguments, and that there is margin for important improvement since we did not simplify the types of partial applications (whose intersection types are in general quite redundant).

Our implementation is already included in the development branch of the the C.Duce distribution which can be retrieved at https://www.cduce.org/redmine/projects/cduce. It currently is in alpha-testing therefore we recommend the user to check the bug-tracker for open issues.

Also available is a prototype which implements the work described in both papers: type inference/reconstruction for implicitly-typed expressions, constraint solving with basic simplification algorithms, evaluation. The implementation is naive, not optimized, and implements very naive simplification heuristics, but it permits a smoother and friendlier evaluation and testing of our system since it is stable, includes an interactive toplevel and contains, a different test suite based on the examples used in both papers. It is available at http://www.pps.univ-paris-diderot.fr/~gc/misc/polyduce.tar.gz