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, ℂDuce, XQuery, OcamlDuce, XHaskell, XAct, are statically-typed functional languages. However, none of them provides full-fledged parametric polymorphism even though this feature has been repeatedly requested in different standardization groups. A major stumbling block to such an extension —*ie*, the definition of a subtyping relation for regular tree types with type variables— 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:

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

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

The first function is the classic `map` function defined in Haskell (we 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 "\"); 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 included in the electronic appendix available on-line). 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

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

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

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

POPL '15, January 15–17 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676991

[1] For `map even` we need to infer just two instantiations, namely, $\{(\gamma\backslash\texttt{Int})/\alpha, (\gamma\backslash\texttt{Int})/\beta\}$ and $\{(\gamma\vee\texttt{Int})/\alpha, (\gamma\backslash\texttt{Int})\vee\texttt{Bool}/\beta\}$. The type in (1) is redundant since the first type of the intersection is an instance (*eg*, for γ=`Int`) of the third. We included it just for the sake of the presentation.

of [3] to make it well typed. In other words, we define "local type inference"[2] 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 (1). 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 $\{\sigma'_j \mid j{\in}J\}$ such that for two given types s and t the relation $\bigwedge_{i\in I} s\sigma_i \leq \bigwedge_{j\in J} t\sigma'_j$ holds, 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 precise than those inferred by the type systems of the ML family. The paper is accompanied by an electronic appendix available at `http://dl.acm.org` which contains all detailed proofs and complete definitions. 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:*

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

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

In the definition, b ranges over basic types (*eg*, `Int`, `Bool`), α ranges over type variables, and $\mathbb{0}$ and $\mathbb{1}$ respectively denote the empty (that types no value) and top (that types all values) types. Coinduction accounts for recursive types and the condition on infinite branches bars out ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). It also ensures that the binary relation $\rhd \subseteq \mathscr{T}^2$ defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian. This gives an induction principle on $\mathscr{T}$ that we will use without any further explicit reference to the relation. We use $\mathsf{var}(t)$ to denote the *set of type variables* occurring in a type t. A type t is said to be *ground* or *closed* if and only if $\mathsf{var}(t)$ is empty. The subtyping relation for these types 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* (*ie*, 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, $s{\to}t$ 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*, $\mathbb{0}{\to}\mathbb{1}$ is the set of all functions[3] and $\mathbb{1}{\to}\mathbb{0}$ is the set of functions that diverge on every argument). Type connectives (*ie*, union, intersection, negation) are interpreted as the corresponding set-theoretic operators (*eg*, $s \vee t$ is the union of the values of the two types). For what concerns non-ground types (*ie*, types with variables occurring in them) all the reader needs to know for this work is that the subtyping relation of Castagna and Xu is preserved by substitution of the type variables. Namely, if $s \leq t$, then $s\sigma \leq t\sigma$ for every type-substitution σ (the converse does not hold in general, while it holds for *semantic* type-substitutions in convex models: see [4]). Two types are equivalent if they are subtype one of each other (type equivalence is denoted by $\simeq$). Finally, notice that in this system $s \leq t$ if and only if $s{\wedge}{\neg}t \leq \mathbb{0}$.

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

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

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

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

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 $\lambda^{\alpha\to\alpha} x.x$, while the classic "auto-application" term is written as $\lambda^{((\alpha\to\beta)\wedge\alpha)\to\beta} x.xx$. The intended meaning of using a type variable, such as α, is that a (well-typed) λ-abstraction not only has the type specified in its label (and by subsumption all its super-types) but also all types obtained by instantiating the type variables occurring in its label. So $\lambda^{\alpha\to\alpha} x.x$ has not only type $\alpha \to \alpha$ but by subsumption also, for instance, the types $\mathbb{0}{\to}\mathbb{1}$ (the type of all functions, which is a super-type of $\alpha{\to}\alpha$) and $\neg$`Int` (the type of all non integer values), and by instantiation the

[2] There are different definitions for *local type inference*. Here we use it with the meaning of finding the type of an expression in which not all type annotations are specified. This is the acceptation used in Scala where, 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 [14].

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

types $\texttt{Int}\to\texttt{Int}$, $\texttt{Bool}\to\texttt{Bool}$, 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 $\lambda^{\alpha\to\alpha}x.x$ has type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})\wedge\neg\texttt{Int}$.

The key problem to be solved, then, is to define an *explicitly-typed* λ-calculus 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 an annotation must be explicitly propagated. That is, to apply $\lambda^{\alpha\to\alpha}x.x$ to $\texttt{42}$ we must first apply the type-substitution $\{\texttt{Int}/\alpha\}$ to it, yielding $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$, and only then we can apply the function to $\texttt{42}$.

(ii) sets of type-substitutions are needed because of intersection types. A function that expects arguments of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ can be safely applied to $\lambda^{\alpha\to\alpha}x.x$, but the latter must be previously instantiated by a *set* of two type-substitutions $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$, yielding $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$ (the application of a set of substitution to a type t returns the intersection of all types obtained by applying each substitutions in the set to t).

(iii) type-substitutions cannot be immediately applied to the body of a function since this may yield ill-typed terms. For instance, consider the following "daffy" definition of the identity function

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

and apply it to the same set of substitutions as before, namely, $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$. This yields the following term

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

which is not well typed: to type it one should prove that under the hypothesis $x:\texttt{Int}$ the term $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$ has type $\texttt{Int}$, and that under the hypothesis $x:\texttt{Bool}$ this same term has type $\texttt{Bool}$, but both checks fail because, in both cases, $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is ill-typed (it neither has type $\texttt{Int}\to\texttt{Int}$ when $x:\texttt{Bool}$, nor has it type $\texttt{Bool}\to\texttt{Bool}$ when $x:\texttt{Int}$).

To cope with these three problems we proposed in Part 1 [3] that the instantiation of the body of a function changes according to the type of the argument of the function. For instance, when we apply the daffy identity function to an integer we must instantiate its body by the type-substitution $\{\texttt{Int}/\alpha\}$, while the type-substitution $\{\texttt{Bool}/\alpha\}$ must be used when the function argument is a Boolean value. To obtain this behavior, in Part 1 [3] we introduced and studied 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 λ-abstractions by (sets of) type-substitutions. For example, in order to pass our daffy identity function (4) to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ we first "lazily" instantiate it as follows:

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

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

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

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

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

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

As customary, we assume bound expression variables to be pairwise distinct and distinct from any free expression variable occurring in the expressions under consideration. Polymorphic variables can be bound by interfaces, but also by decorations: for example, in $\lambda^{\beta\to\beta}_{[\{\alpha/\beta\}]}x.(\lambda^{\alpha\to\alpha}y.y)x$, the α occurring in the interface of the inner abstraction is "bound" by the decoration $[\{\alpha/\beta\}]$, and the whole expression is α-equivalent to $(\lambda^{\beta\to\beta}_{[\{\gamma/\beta\}]}x.(\lambda^{\gamma\to\gamma}y.y)x)$. If a type variable is bound by an outer abstraction, it cannot be instantiated; such a 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 "@" which takes an expression e and a set of type-substitutions $[\sigma_j]_{j\in J}$ and pushes $[\sigma_j]_{j\in J}$ down to all outermost λ-abstractions occurring in e (and collects and composes with the sets of type-substitutions it meets). Precisely, $e@[\sigma_j]_{j\in J}$ is defined for λ-abstractions and applications of type-substitutions as

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

(where $\circ$ denotes the pairwise composition of all substitutions of the two sets). It erases the set of type-substitutions when e is either a variable or a constant, and it is homomorphically applied on the remaining expressions (see [3] for comprehensive definitions). The dynamic semantics is given by the following notions of reduction (where v denotes a *value*), applied by a leftmost-outermost strategy:

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

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

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

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

The first rule (8) performs relabeling, that is, it propagates the sets of type-substitutions down into the decorations of the outermost λ-abstractions. The second rule (9) states the semantics of applications: this is standard call-by-value β-reduction, with the dif-

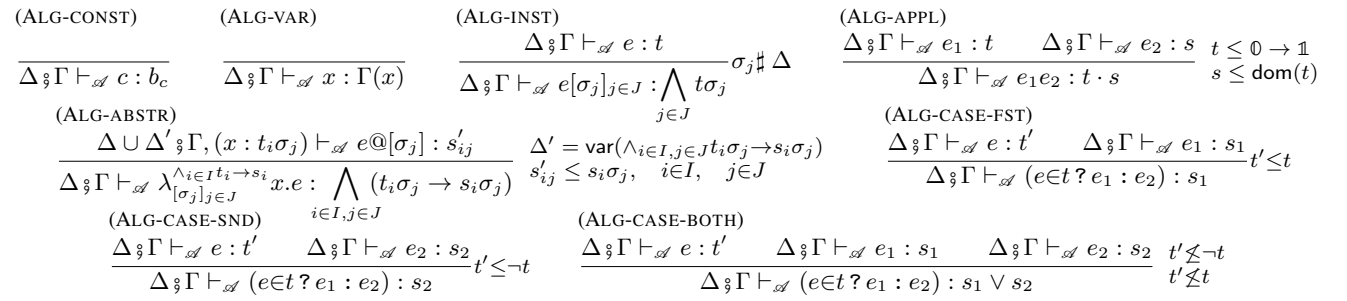

Figure 1. Typing algorithm

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

The static semantics is given by the rules in Figure 1 which form an algorithmic system (as stressed by the $\mathscr{A}$ subscript in $\vdash_{\mathscr{A}}$ 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-APP). 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 Γ. Rule (ALG-ABSTR) checks that $\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e$ has the type declared by (the combination of) its interface and its decoration, that is, $\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$. To do that it first adds all the variables occurring in this type to the set Δ, (in the function body these variables are monomorphic). Then, it checks that for every possible input type —*ie*, for every possible combination of t_i and σ_j— the function body e relabeled with the single type-substitution σ_j under consideration (*ie*, $e@[\sigma_j]$), has (a subtype of) the corresponding output type.

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

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

The type system is sound (it satisfies both subject reduction and progress), it subsumes existing intersection type systems, and type inference is decidable. Furthermore the calculus can be compiled into an intermediate language which executes relabeling only by need and, thus, can be efficiently evaluated (again, see Part 1 [3]).

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

2.1 Overview and contributions of this article (Part 2)

Recall that we want the programmer to use the implicitly-typed expressions of grammar (3), and not those of grammar (7) which would require the programmer to write explicit type-substitutions. Therefore in Section 3 we define a local type inference system that, given an implicitly-typed expression produced by the grammar (3), checks whether and where some sets of type-substitutions can be inserted in this expression so as to make it a well-typed explicitly-typed expression of grammar (7). Thus, our local type inference consists of a type-substitution reconstruction system, insofar as it has to reconstruct the sets of type-substitutions that make an expression of grammar (3) a well-typed expression of grammar (7). In order to avoid ambiguity we reserve the word "reconstruction" for the problem of reconstructing *type* annotations (in particular, function interfaces) and speak of *inference of type-substitutions* for the problem of local type inference. In particular, we show that this problem can be reduced to the problem of deciding whether for two types s and t there exist two sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that $s[\sigma_i]_{i\in I} \leq t[\sigma'_j]_{j\in J}$. We prove that when the cardinalities of I and J are given, the problem above is decidable and reduces to the problem of finding all substitutions σ such that $s'\sigma \leq t'\sigma$ for two given types s' and t' (we dub this latter problem the *tallying problem*). We show how to produce a sound and complete set of solutions for the latter problem. This is done by generating *sets* of constraint-sets that are then *normalized*, *merged*, and *solved*. The solution of the tallying problem immediately yields a semi-decision procedure (that tries all the cardinalities for I, J) for the local type inference system. Henceforth, to enhance readability, we will systematically use the metavariable "a" to denote expressions of the implicitly-typed calculus (*ie*, those of grammar (3)) and reserve the metavariable "e" for expressions of the explicitly-typed calculus (*ie*, those of grammar (7)). Finally, in Section 4 we show that the theory and algorithms developed in Section 3 can be reused to do ML-like type reconstruction, that is, to infer the interface of unannotated λ-expressions in a pure λ-calculus with type-case.

In summary, the results of this paper make it possible to program in the implicitly-typed calculus (3), by compiling it into well-typed explicitly-typed expressions (7) defined in [3]. Let us show

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

the details on the motivating example of the introduction. First, note that in the implicitly-typed calculus (3) `even` can be defined as

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

(where $s\setminus t$ is syntactic sugar for $s\wedge\neg t$) while —with the products and recursive function definitions given in the appendix— `map` is

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

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

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

Contributions: The overall contribution of this work (Parts I and II) 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 II 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 σ such that $t_1\sigma \leq t_2\sigma$. This is obtained by using the set-theoretic interpretation of types to reduce the problem to a unification problem on regular tree types.

– the definition of a type-substitution inference system sound and complete w.r.t. the 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 λ-calculus with intersection and recursive types and with explicitly-typed λ-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 $\lambda^{\alpha\to\alpha}x.3$ 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 $\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\}]}x.3$. Likewise, if the programmer specified the signature `map : :` $(\alpha\texttt{->}\beta)\texttt{->}\gamma$, we expect the system to answer that the definition of `map` does not conform this signature, rather than it conforms the signature by substituting $[\alpha]\texttt{->}[\beta]$ for γ (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 (*ie*, those not in Δ) that makes $s[\sigma_i]_{i\in I} \leq t$ hold. The reader may wonder why we apply the type-substitution only on the smaller type and not on both types. The reason can be understood by looking at the rules in Figure 1 and 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-*)) 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, Δ a set of type variables, and $[\sigma_i]_{i\in I}$ a set of type-substitutions. We define:*

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

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

Intuitively, it suffices to replace $\leq$ by $\sqsubseteq_\Delta$ and $\not\leq$ by $\not\sqsubseteq_\Delta$ in the algorithmic rules of Figure 1 (where Δ is the set of monomorphic variables used in the premises) to obtain the corresponding rules of type-substitution inference. This yields the system formed by the rules in Figure 2 (we subscripted the turnstile symbol by $\mathscr{I}$

$$\text{(INF-ABSTR)}\quad \frac{\Delta \cup \Delta' \wr \Gamma, x : t_i \vdash_{\mathscr{J}} a : s'_i}{\Delta \wr \Gamma \vdash_{\mathscr{J}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I}(t_i \to s_i)} \; \Delta' = \mathsf{var}(\wedge_{i\in I} t_i \to s_i),\; s'_i \sqsubseteq_{\Delta\cup\Delta'} s_i,\quad i\in I$$

$$\text{(INF-CASE-BOTH)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathscr{J}} a : t' \qquad \Delta \wr \Gamma \vdash_{\mathscr{J}} a_1 : s_1 \qquad \Delta \wr \Gamma \vdash_{\mathscr{J}} a_2 : s_2}{\Delta \wr \Gamma \vdash_{\mathscr{J}} (a \in t\,?\,a_1 : a_2) : s_1 \vee s_2} \; t' \not\sqsubseteq_\Delta \neg t,\; t' \not\sqsubseteq_\Delta t$$

$$\text{(INF-CASE-FST)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathscr{J}} a : t' \quad \Delta \wr \Gamma \vdash_{\mathscr{J}} a_1 : s_1}{\Delta \wr \Gamma \vdash_{\mathscr{J}} (a \in t\,?\,a_1 : a_2) : s_1} t' \sqsubseteq_\Delta t$$

$$\text{(INF-CASE-SND)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathscr{J}} a : t' \quad \Delta \wr \Gamma \vdash_{\mathscr{J}} a_2 : s_2}{\Delta \wr \Gamma \vdash_{\mathscr{J}} (a \in t\,?\,a_1 : a_2) : s_2} t' \sqsubseteq_\Delta \neg t$$

$$\text{(INF-APPL)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathscr{J}} a_1 : t \quad \Delta \wr \Gamma \vdash_{\mathscr{J}} a_2 : s}{\Delta \wr \Gamma \vdash_{\mathscr{J}} a_1 a_2 : u} \; u \in (t \bullet_\Delta s)$$

Figure 2. Inference system for type-substitutions

to stress that it is the $\mathscr{J}$nference 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 λ-abstractions have empty decorations, and which uses the $\Delta\cup\Delta'$ set to compare the types of the body with the result types specified in the interface ($s'_i \sqsubseteq_{\Delta\cup\Delta'} s_i$). Notice that we do not require the sets of type-substitutions that make $s'_i \sqsubseteq_{\Delta\cup\Delta'} s_i$ satisfied to be the same for all $i\in I$: this is not a problem since the case of different sets of type-substitutions corresponds to using their union as sets of type-substitutions (*ie*, to intersecting them point-wise: see Definition B.9 and Corollary B.12 —henceforth, references starting with letters refer to appendixes).

It still remains the most delicate rule, (INF-APPL), the one for application. It is difficult because not only it must find two distinct sets of type-substitutions (one for the function type the other for the argument type) but also because the set of type-substitutions for the function type must enforce two distinct constraints: the type resulting from applying the set of type-susbstitutions to the type of the function must be a subtype of $\mathbb{0}\to\mathbb{1}$, and its domain must be compatible with (*ie*, a supertype 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 $t \bullet_\Delta s$ as the set of types for which there exist two sets of type-substitutions (for variables not in Δ) that make s compatible with the domain of t:

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

In practice, this set takes all the pairs of sets of type-substitutions that make t a function type, and s an argument type compatible with t and collects all the possible result types. This set is closed by intersection (see Lemma 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 $t \bullet_\Delta s$. This yields the inference rule (INF-APPL) of Figure 2.

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

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

The proof of the soundness property is constructive: it builds along the derivation for the implicitly-typed expressions a an explicitly-typed expression e that satisfies the statement of the theorem; this expression is the one that is then compiled in the intermediate language we defined in Part 1 [3] and evaluated. Notice that $\sqsubseteq_\Delta$ gauges the generality of the solutions found by the inference system: the smaller the type found, the more general the solution is. As a matter of fact, adding to the system in Figure 2 a subsumption rule that uses the relation $\sqsubseteq_\Delta$ that is:

$$\text{(SUBSUMPTION)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathscr{J}} a : t_1 \quad t_1 \sqsubseteq_\Delta t_2}{\Delta \wr \Gamma \vdash_{\mathscr{J}} a : t_2}$$

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

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

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

3.2 Type tallying

We define the tallying problem as follows

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

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 $\vdash_{\mathscr{J}}$ corresponds to finding and solving a particular tallying problem. First, notice that for the remaining rules the problem we have to solve is to prove (or disprove) the relation $s \sqsubseteq_\Delta t$ for given s and t. By definition this corresponds to finding a set of n type-substitutions $[\sigma_i]_{i\leq n}$ such that $\bigwedge_{i\leq n} s\sigma_i \leq t$. We can split each type-substitution σ_i in two: a *renaming type-substitution* ρ_i that maps each variable of s not in Δ into a *fresh* type variable, and a type substitution σ'_i such that $\sigma_i = \sigma'_i \circ \rho_i$. Thus the inequation becomes $\bigwedge_{i\leq n}(s\rho_i)\sigma'_i \leq t$. The domains of σ'_i are

0. $\mathsf{norm}(t, M)$ =
1. if $t \in M$ then return $\{\varnothing\}$ else
2. if $t = \bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r$ and α_k is the smallest variable (wrt $\preccurlyeq$) for k in $P'\cup N'$then return$\{\mathsf{single}(\alpha_k, t)\}$ else
3. if $t = \bigwedge_{p\in P} b_p \wedge \bigwedge_{n\in N} \neg b_n$ then (if $\bigwedge_{p\in P} b_p \leq \bigvee_{n\in N} b_n$ then return $\{\varnothing\}$ else return $\varnothing$) else
4. if $t = \bigwedge_{p\in P}(s_p\to t_p) \wedge \bigwedge_{n\in N} \neg(s_n\to t_n)$ then return
5. $\bigsqcup_{n\in N}\Big(\mathsf{norm}(s_n \wedge \bigwedge_{p\in P} \neg s_p, M\cup\{t\}) \sqcap \Big(\prod_{P'\subset P}\big(\mathsf{norm}(s_n \wedge \bigwedge_{p\in P'} \neg s_p, M\cup\{t\}) \sqcup \mathsf{norm}(\bigwedge_{p\in P\setminus P'} t_p \wedge \neg t_n, M\cup\{t\})\big)\Big)\Big)$ else
6. if $t = \bigvee_{i\in I} t_i$ then return $\bigsqcap_{i\in I} \mathsf{norm}(t_i, M)$ else let t' be the disjunctive normal form of t in return $\mathsf{norm}(t', M)$

Figure 3. Constraint normalization

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 $\sigma = \bigcup_{i\leq n} \sigma'_i$ and apply it to t with no effect, yielding the inequation $(\bigwedge_{i\leq n} s\rho_i)\sigma \leq t\sigma$. Let $u_n = \bigwedge_{i\leq n} s\rho_i$, we have just transformed the problem of proving the relation $s \sqsubseteq_\Delta t$ into the problem of finding an n for which there exists a solution to the tallying problem for $\{u_n \leq t\}$ and Δ. The way to proceed to find n is explained in Section 3.2.3.

The (INF-APPL) rule deserves 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 Δ such that both $\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1}$ and $\bigwedge_{j\in J} s\sigma_j \leq \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i)$ hold. In this section we give an algorithm that produces a set of solutions for the (INF-APPL) rule problem that is sound (it finds only correct solutions) and complete (any other solution can be derived from those returned by the algorithm). To this end we proceed in three steps: (i) given a tallying problem, we show how to effectively produce a finite set of solutions that is sound (it contains only correct solutions) and complete (every other solution of the problem is less general—in the usual sense of unification, *ie*, it is larger wrt $\sqsubseteq$— than some solution in the set); (ii) we show that if we fix the cardinalities of I and J, then it is possible to reduce the (INF-APPL) rule problem to a tallying problem; (iii) from this we deduce a sound and complete algorithm to semi-decide the general (INF-APPL) rule problem and thus the whole inference system.

We solve each problem in one of the next subsections, but before we recall an important property of semantic subtyping systems [4, 10] which states that every type is equivalent to (and can be effectively transformed into) a type in *disjunctive normal form*, that is, a union of *uniform* intersections of *literals*. A literal is either an arrow, or a basic type, or a type variable, or a negation thereof. An intersection is uniform if it is composed of literals with the same constructor, that is, either it is an intersection of arrows, type variables, and their negations or it is an intersection of basic types, type variables, and their negations. In summary, a disjunctive normal form is a union of summands whose form is either

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

or

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

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

3.2.1 Solution of the tallying problem.

In order to solve the tallying problem for given Δ and C, we first fix some total order $\preccurlyeq$ —any will do— on the type variables occurring in C and not in Δ (from now on, when speaking of type variables we will mean type variables not in Δ), order that will be used to ensure that all inferred types satisfy the contractivity 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 solvable 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 $\mathscr{S}_1, \mathscr{S}_2 \subseteq \mathscr{P}(\mathscr{T}\times\mathscr{T})$ be two sets of constraint-sets. We define*

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

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

We also define an auxiliary function single that singles out a given top-level variable of a normal form. More precisely, given a type t which is a summand of a normal form, that is, $t = \bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r$ and $k \in P'\cup N'$, we define $\mathsf{single}(\alpha_k, t)$ as the constraint equivalent to $t \leq \mathbb{0}$ in which α_k is "singled-out", that is,[5]

$\bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in(N'\setminus\{k\})} \neg\alpha_r \leq \alpha_k$ when $k \in N'$ and

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

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

$C = \{(\alpha\to\mathtt{Bool}, \beta\to\beta)\ ,\ (\mathtt{Int}\vee\mathtt{Bool}\to\mathtt{Int}\ ,\ \alpha\to\beta)\}$

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

1. Constraint normalization. We define a function norm that takes a type t and generates a set of *normalized* constraint-sets —*ie*, constraint-sets formed by constraints whose form is either $\alpha \leq s$ or $s \leq \alpha$— whose set of solutions is sound and complete w.r.t. the constraint $t \leq \mathbb{0}$. This function is parametric in a 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 $\mathbb{0}$ (beginning of line 6). If a summand contains a top-level variable, then the smallest (wrt $\preccurlyeq$) top-level variable is singled out (line 2). If there is no top-level variable and there are only basic types, then the algorithm checks

[5] Equivalence of $t \leq \mathbb{0}$ and the two following constraints is easily derived from the De Morgan's laws and the property $t_1 \leq t_2 \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$.

the constraint by calling the subtyping algorithm and, accordingly, it returns either the unsatisfiable set of constraint-sets ($\varnothing$) or the one that is always satisfied ($\{\varnothing\}$) (line 3). Finally, if there are only intersections of arrows and their negations, then the problem is decomposed into a set of subproblems by using the decomposition rule of the subtyping algorithm for semantic subtyping (see [10] 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 norm generates a constraint (α, t) or (t, α), then every variable smaller than or equal to α may occur in t only under an arrow (equivalently, every top-level variable of t is strictly larger than α).

REMARK 3.1. *There is the special case of (α, t) or (t, α) in which t is itself a variable. In that case we give priority to the smallest variable and consider the larger variable be a bound for the lower one but not vice-versa. This point will be important for* 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* norm *works on single constraints (actually, on a type t representing the constraint $t \leq \mathbb{0}$), so let us apply it on the first constraint of the example. We want to normalize the constraint $\alpha\to\texttt{Bool} \leq \beta\to\beta$, and thus we apply* norm *to the type $(\alpha\to\texttt{Bool}) \wedge \neg(\beta\to\beta)$. Now, this constraint has two distinct solutions: either (i) β is the empty set, in which case the larger type becomes $\mathbb{0}\to\mathbb{0}$ that is the type of all functions (see Footnote 3) which contains every arrow type, in particular $\alpha\to\texttt{Bool}$, or (ii) the types satisfy the usual covariant-contravariant rule for arrows, that is, $\beta \leq \alpha$ and $\texttt{Bool} \leq \beta$. Since there are* two *distinct solutions, then* norm *generates a set of* two *constraint-sets. Precisely* $\mathsf{norm}((\alpha\to\texttt{Bool}) \wedge \neg(\beta\to\beta), \varnothing)$ *returns* $\{\ \{\ (\beta, \mathbb{0})\ \}\ ,\ \{\ (\beta, \alpha)\ ,\ (\texttt{Bool}, \beta)\ \}\ \}$. *Both constraint-sets are normalized and are computed by Line 5 in Figure 3: the first constraint-set is computed by the rightmost recursive call of* norm *(notice that $P'{=}\varnothing$ —since it ranges over the* strict subsets *of P which, in this case, is a singleton—so it requires s_n,* ie *β, to be empty), while the second constraint-set is obtained by the union of the first two recursive calls (which require $s_n \leq s_p$ and $t_p \leq t_n$).*

2. Constraint merging. Take a normalized constraint-set. Each constraint of this set isolates one particular variable. However, the same variable can be isolated by several distinct constraints in the set. We next want to transform this constraint-set into an equivalent one (*ie*, a constraint-set with exactly the same set of solutions) in which every variable is isolated in at most two constraints, one 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 $\{\ s_i{\leq}\alpha_i{\leq}t_i\ \mid\ i \in I\ \}$ where the α_i's are pairwise distinct. This is done by the function $\mathsf{merge}(C, M)$ where C is a normalized constraint-set and M a set containing the types already visited by the function.

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

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

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

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

$$\{\quad \{\ (\alpha, \mathbb{0}), (\beta, \mathbb{0})\ \}\ ,\\ \{\ (\alpha, \texttt{Int}\vee\texttt{Bool})\ ,\ (\texttt{Int}, \beta)\ ,\ (\beta, \mathbb{0})\ \}\ ,\\ \{\ (\texttt{Bool}, \beta)\ ,\ (\beta, \alpha)\ ,\ (\alpha, \mathbb{0})\ \}\ ,\\ \{\ (\texttt{Bool}, \beta)\ ,\ (\texttt{Int}, \beta)\ ,\ (\beta, \alpha)\ ,\ (\alpha, \texttt{Int}\vee\texttt{Bool})\ \}\quad \}$$

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

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

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

The function $\mathsf{solve}(C)$ takes every constraint $s \leq \alpha \leq t$ in C and replaces it by $\alpha = (s \vee \beta) \wedge t$ (with β fresh). It is clear that the constraint-set C has a solution for every possible assignment of α included between s and t if and only if the new constraint-set has a solution for every possible (unconstrained) assignment of β. At the end, the constraint-set $\{s_i \leq \alpha_i \leq t_i \mid i \in I\}$ has become a set of equations of the form $\{\alpha_i = u_i \mid i \in I\}$ where the α_i's are pairwise distinct. By construction, this set of equations has the property that every variable that is smaller than or equal to (wrt $\preccurlyeq$) α_i may occur in u_i only under an arrow (as for constraint-sets we say that the set of equations is *well ordered*). This last property ensures the contractivity of the equation defining the smallest type variable. By Courcelle [5] (and Lemma C.44) there exists a solution of this set, namely, a substitution from the type variables $\alpha_1, ..., \alpha_n$ into (possibly recursive regular) types $t_1, ..., t_n$ whose variables are contained in the fresh β_i's variables introduced by solve (all universally quantified, *ie*, no upper or lower bound) and the type variables in Δ. This solution is given by the following Unify procedure in which we use μ-notation to denote regular types and where E is a well-ordered set of equations.

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

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

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

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

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

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

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

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

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

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

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

Finally, solve introduces several fresh polymorphic variables which can be cleaned up 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 $\mathbb{0}$ (resp. $\mathbb{1}$). This is what we implicitly did in our example to solve β and eliminate γ from the constraint of α.

3.2.2 Solution for application with fixed cardinalities

It remains to solve the problem for the (INF-APPL) rule. We recall that given two types s and t, a solution for this problem is *a pair* of sets of type-substitutions $[\sigma_i]_{i\in I}$, $[\sigma_j]_{j\in J}$ for variables not in Δ that make both of these two inequations

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

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

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

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

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

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

3.2.3 Solution of the application problem

The algorithm to solve the general problem for the (INF-APPL) rule explores all the possible combinations of the cardinalities of I

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

and J by, say, a dove-tail order. More precisely, we start with both I and J at cardinality 1 and:

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

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

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

This constitutes a sound and complete semi-decision procedure for the application problem and, thus, for the type-substitution inference system (Theorem 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 $t = (\mathtt{Int} \to \mathtt{Bool}) \wedge (\alpha \setminus \mathtt{Int} \to \alpha \setminus \mathtt{Int})$ 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 [18] for more details on this example): one is unsatisfiable since it contains the constraint $t \leq \mathbb{0}$ (an intersection of arrows is never empty since it always contains $\mathbb{1} \to \mathbb{0}$ the type of the diverging functions); four of these are less general than some other (their solutions are included in the solutions of the other) and the remaining four are obtained by adding the constraint $\gamma \geq [\alpha_1] \to [\beta_1]$ respectively to $\{\alpha_1 \leq \mathbb{0}\}$, $\{\alpha_1 \leq \mathtt{Int}, \mathtt{Bool} \leq \beta_1\}$, $\{\alpha_1 \leq \alpha \setminus \mathtt{Int}, \alpha \setminus \mathtt{Int} \leq \beta_1\}$, $\{\alpha_1 \leq \alpha \vee \mathtt{Int}, (\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool} \leq \beta_1\}$, yielding the following four solutions for γ: $\{\gamma = [] \to []\}$, or $\{\gamma = [\mathtt{Int}] \to [\mathtt{Bool}]\}$, or $\{\gamma = [\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]\}$, or $\{\gamma = [\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}]\}$. 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 $\{((\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1]) \wedge ((\alpha_2 \to \beta_2) \to [\alpha_2] \to [\beta_2]) \leq t \to \gamma\}$ for which the minimal solution is, as expected:
$\{\gamma = ([\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]) \wedge ([\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}])\}$

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, continuing with a further expansion may yield a more general solution. We have just seen that in the case of `map even` the good solution is the second one, although this solution could already have been deduced by intersecting the first minimal solutions we found. A simple example that shows that carrying on after a first solution may yield a better solution is the application of a function of type $(\alpha \times \beta) \to (\beta \times \alpha)$ to an argument of type $(\mathtt{Int} \times \mathtt{Bool}) \vee (\mathtt{Bool} \times \mathtt{Int})$. For this applications our algorithm (extended with product types) returns after one iteration the type $(\mathtt{Int} \vee \mathtt{Bool}) \times (\mathtt{Int} \vee \mathtt{Bool})$ (since it unifies α with β) while one further iteration would deduce the more precise type $(\mathtt{Int} \times \mathtt{Bool}) \vee (\mathtt{Bool} \times \mathtt{Int})$. Of course this raises the problem of the existence of principal types (notice that the type t' in the statement of Theorem 3.3 is not unique): 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.

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

In summary, programming in the implicitly-typed calculus corresponds to programming in the explicitly-typed one with the difference that we delegate to the system the task to write type-substitutions for us and with the caveat that by doing that we make the dynamic test of the type of a polymorphic function to be implementation dependent. Of course, forbidding the dynamic test of the type of polymorphic functions (or of functions *tout court*) make this problem disappear (and yields much a simpler implementation).

3.3 Examples and experiments

We developed two implementations (see the last page of the appendix for download instructions), a complete but non-optimized prototype with products and arrow type constructors, and an extension of the full ℂDuce language which is highly optimized (types are stored in hash-consed binary decision trees to avoid the cost of normalization, pattern-matching has optimal implementation, static

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.[7] In this extension `map` has type `(α→β)→[α*]→[β*]` 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

```
let pretty (x: Int): String = string_of x
```

or the whole interface when they are typed by an intersection type:

```
let even (Int->Bool ; (α\Int)->(α\Int))
  | x & Int -> (x mod 2) = 0
  | x -> x
```

The type returned for the partial application `map even` is then

```
([(α\Int)*]→[(α\Int)*])&([(α|Int)*]→[(α\Int|Bool)*])
```

(unions and intersections are denoted by `|` and `&`, respectively) while the one for `map pretty` is `([]→[])&([Int*]->[String*])`. 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 `[]→[]`, stating that the function maps the empty list into itself. Interestingly, the solver does not need to know the *body* of `map` to deduce it; this is because ℂDuce encodes lists by recursive union types (`[α*]` stands for $\mu X.\texttt{nil}\vee(\alpha, X)$ where `nil` denotes the empty list) and our system tries to infer a result 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):

```
# let g ((Int -> Int) -> Int -> Int ;
         (Bool -> Bool) -> Bool -> Bool) x -> x;;
val g : (Bool -> Bool) -> Bool -> Bool
      & (Int -> Int) -> Int -> Int = <fun>

# let id ('a -> 'a) x -> x;;
val id : 'a -> 'a = <fun>

# g id;;
- : (Bool -> Bool) & (Int -> Int) = <fun>

# id (id g) ;;
- : (Bool -> Bool) -> Bool -> Bool
  & (Int -> Int) -> Int -> Int = <fun>
```

Our system includes singleton types (types that contain a single value) and thus the type returned, for instance, for

```
let churchtrue (x: α)(y: β): α = x in churchtrue 42;;
```

is $\beta \to 42$ (*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 `max` (whose definition uses the ℂDuce's polymorphic comparison operator `>>` "greater than"):

```
# let max (x: α)(y: α): α = if (x >> y) then x else y;;
val max : α -> α -> α = <fun>
```

An ML programmer would probably expect the partial application `max 42` to be typed as `42→42` or `Int→Int` (at least, *we* were naively expecting that). Instead, for this application the system returns the type `(β|42)→(β|42)`, and rightly so. The point is that our system includes union types and, therefore, an application such as `max 42 "3"` is well typed: it suffices to instantiate the variable α in the type of `max` by the union type `Int|String`. To give the final instantiation for α the type system must know the type of *both* arguments of `max`, therefore in the case of the partial application, it instantiates α with `β|42` stating that it knows that α contains *at least* the value 42 and waits for the second argument to instantiate the missing part, represented by β; and the type returned for `max 42 "3"` is `42|"3"`. In this example we specified the type `α->α->α` for `max`, since this is what an ML programmer would have written. However, in a system with polymorphic union types a more meaningful type for `max` is `α->β->α|β`: if we specify such a type for `max`, then the type deduced for the partial application `max 42` is, more intuitively, `β->(β|42)`. The fact that `α->β->α|β` and `α->α->α` cannot, from any practical point of view, be distinguished seems a nice feature of our system. Nevertheless notice that the same type deduction as for `max` would have happened if in the definition of `churchtrue` we had used α instead of β; in that case `churchtrue 42 "3"` would have been typed by the (less precise) union type `42|"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` follows a pattern that is common in programming functional data structures. This can be seen by examining Okasaki's implementation of red-black trees [13]. 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 expressive enough to verify that his code, without any modification, satisfies color invariants. Our types, instead, can do it as follows:

```
type RBtree = Btree | Rtree
type Btree  = <blk elem=α>[ RBtree RBtree ] | []
type Rtree  = <red elem=α>[ Btree  Btree ]
type Unbal  = <blk elem=α>( [ Wrong RBtree ]
                          | [ RBtree Wrong ])
type Wrong  = <red elem=α>( [ Rtree Btree ]
                          | [ Btree Rtree ])

let balance ( Unbal -> Rtree ; (β\Unbal) -> (β\Unbal) )
  | <blk (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ]
  | <blk (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ]
  | <blk (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ]
  | <blk (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ]
      -> <red (y)>[  <blk (x)>[ a b ]   <blk (z)>[ c d ]  ]
  | x -> x
```

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 `elem` for their content, and that delimit 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→Btree→Btree`, 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 represented as a hash-consed 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 types 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-

[7] 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 $\alpha \to \beta$ rather than `'a -> 'b`.

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 $\mathbb{C}$Duce 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 $\alpha\to\beta$ and deduce the corresponding constraints. Formally, we consider expressions produced by the following grammar:

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

together with the judgment $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$ that states that under the typing environment Γ, m has type t under the constraints in $\mathscr{S}$, provided that $\mathscr{S}$ is satisfiable (the turnstile subscript $\mathscr{R}$ indicates that this is a type $\mathscr{R}$econstruction 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 $\mathscr{S}$) and thus it is an intersection type obtained by applying this set of type-substitutions to the type.

The soundness of this system is a consequence of the results on the type-substitution inference of the previous sections. As a matter of facts, this system is precisely the same system as the one in the previous sections with the only difference that all interfaces are of the form $\alpha\to\beta$ and, to compensate that, we infer type-substitutions

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

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

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

$$\text{(R-CASE)}\quad \begin{array}{rl} \mathscr{S} = & (\mathscr{S}_0 \sqcap \{\{(t_0 \le \mathbb{0})\}\}) \\ \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0 \le t), (t_1 \le \alpha)\}\}) \\ \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0 \le \neg t), (t_2 \le \alpha)\}\}) \\ \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1 \vee t_2 \le \alpha)\}\}) \end{array}$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m_0 : t_0 \rightsquigarrow \mathscr{S}_0 \qquad \Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} (m_0 {\in} t\,?\,m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}$$

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

Figure 4. Type reconstruction rules

in decorations (we also used a different and more standard presentation to stress constraint generation). Of course, completeness does not hold: far from that. For instance, it is impossible, in general, to deduce for a function without type annotations the type $\mathbb{1}\to\mathbb{0}$ — the type of all diverging functions— since this would correspond to decide the halting problem (though our algorithm returns for $\mu f\, x{=}f(x)$ the same type as in ML, that is, $\alpha\to\beta$). 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.* [15]) 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 [11], 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 $\lambda x.xx$ for which it returns the recursive type $t = \mu X.(\alpha\wedge(X\to\beta)) \to \beta$. This type is a subtype of —thus, it is more precise than— the classic (non-recursive) typing of auto-application in intersection type systems $t \le (\alpha\wedge(\alpha\to\beta)) \to \beta$ (though it is not as precise as its subtype $\mu X.(\alpha\vee(X\to\beta)) \to \beta$ which can also type auto-application). As a final example, if we apply our type reconstruction algorithm (extended with products and recursive functions) to the type erasure of the map function defined in equation (12), then we obtain the type (in CDuce's syntax) $((\alpha\to\beta) \to \texttt{[}\alpha\texttt{*]}\to\texttt{[}\beta\texttt{*]}) \wedge ((\mathbb{0}\to\mathbb{1}) \to \texttt{[]}\to\texttt{[]})$ (see the complete unfolding of the algorithm in Appendix D). Thanks to the precise typing of the type-case, our type is slightly more accurate than the ML type, since it states that the application of map to any function and the empty list returns the empty list.

Finally, the "type" returned by the type reconstruction algorithm is not always very readable and often needs to be simplified. For instance, the type we showed for map was obtained after apply-

ing some simplifications —one of which was done by hand—, and defining an algorithm that does the right simplifications is not obvious (*eg*, how to detect that the type $(\alpha\wedge(\alpha\to\beta))\to\beta$ is much more readable than the type $\mu X.(\alpha\wedge(X\to\beta))\to\beta$ reconstructed for auto-application by our algorithm?). The simplification of types (or of type constraints) is a stand alone research topic that deserves further investigation. Nevertheless our reconstruction algorithm can already be used as is, to make type declaration of local functions optional. Indeed for local functions the system is not required to return a "readable" type to the programmer, but just to check whether there exists a typing for local functions that is compatible with their usage; and, for that, our system is enough, even though we will probably have to couple it with bidirectional typing techniques [7] 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 $\vdash_{\mathscr{I}}$ is the same as in the explicitly-typed calculus (this corresponds to disregarding sets of type-substitutions applied inside a pair, as they can equivalently be inferred outside the pair: $t_i \not\simeq \mathbb{0}$ and $(t_1\times t_2) \sqsubseteq_\Delta (s_1\times s_2) \Leftrightarrow t_i \sqsubseteq_\Delta s_i$). Instead, as expected, the rule for projection $\pi_i e$ needs some special care since if the type inferred for e is, say, t, then we need to find a set of substitutions $[\sigma_i]_{i\in I}$ such that $\bigwedge_{i\in I} t\sigma_i \leq \mathbb{1}\times\mathbb{1}$. This problem can be solved by using the very same technique we introduced for $\bullet_\Delta$, namely by solving a sequence of tallying problems generated by increasing at each step the cardinality of I (see the appendix).

In the first part of this work [3] we studied the extension of the explicitly-typed calculus with `let`-polymorphism, in particular, its typing and efficient execution (see Section 5.4 of [3]). There we distinguished `let`-bound variables by underlining them. Reconstruction is mostly useful when combined with `let`-polymorphism. To extend our reconstruction to `let` we use a separate type environment Φ for these variables (while we reserve Γ for λ-abstracted variables). As in Damas-Milner $\mathscr{W}$ algorithm [6] we need to define $\overline{\Gamma}(t)$, the generalization (*closure* in [6]) of a type t wrt the type environment Γ, that is, $\overline{\Gamma}(t) \stackrel{\text{def}}{=} t[\{\gamma_i/\alpha_i \mid \alpha_i \in \mathsf{var}(t)\backslash\mathsf{var}(\Gamma)\}]$ where γ_i are fresh. Then the rules for type reconstruction are

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

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

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 1 of this work [3].

Type inference in ML has essentially been considered as a constraint solving problem [12, 15]. 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 $\sqsubseteq$ relation which is much more thorough than the corresponding relation used in ML inference insofar as it encompasses instantiation, expansion, and subtyping. The important novelty of our work comes from the use of set-theoretic connectives, which allows us to turn sets of constraints of the form $s\leq\alpha\leq t$ into sets of equations of the form $\alpha = (\beta \vee s) \wedge t$, 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 [5]. 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 [12, 15] 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 [12, 15] 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 —*ie*, polymorphic $\mathbb{C}$Duce— is Typed Racket [16, 17] since it has recursive types, union types, top and singleton types, subtyping, dynamic type-cases (called *occurrence typing* in [16, 17]), 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 (*eg*, 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 internally defined in terms of propositional logic where atoms are the elements of a type environment (*eg*, $x : \tau$). The use of logical formulas coincides in $\mathbb{C}$Duce to computing the types of capture variable of patterns (*cf.* the operator $t /\!\!/ p$ in Appendix E or in [9]): the use of propositional logic corresponds to the use of Boolean connectives in $\mathbb{C}$Duce's patterns and unsatisfiability of a formula to type emptiness. This is why all the examples in the "Challenges" section of [17] can be easily defined and precisely typed in our system (straightforwardly with the syntactic sugar defined in the Appendix E of Part 1 [3]). Actually, these examples can already be defined and typed in monomorphic $\mathbb{C}$Duce [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 substitutitions) both for the function and for its argument. It is not clear whether using generic SMT solvers for typing (as suggested by [17]) also in $\mathbb{C}$Duce 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 [14]. 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 [14]: 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 [14] does just the former (Typed Racket does the same). As a consequence while the application of the polymorphic identity $\lambda^{\alpha\to\alpha}x.x$ to a function f of type $(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}$ can be typed in their systems (by inferring the instantiation $\{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}/_\alpha\}$) 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 $\{\mathtt{Int}/_\alpha\}$). Second, our system accounts for more expressive types, expressions, and subtyping relations. For instance, [14] 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, [14] 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 are thoroughly discussed in the extended version of this paper available on line.

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 ℂDuce [2] possible and pave the way to the definition of a *real* type system for XQuery 3.0 [8] (not the current one in which all higher-order functions have type "`function()`"). Thanks to type reconstruction, these languages can have a syntax and semantics close to those of 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 demonstrates, 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 ℂDuce we never met a single typecase for a function type. Nevertheless, it may be interesting to study how to remove such a latitude either by defining a canonical choice for the instances deduced by the inference system (a problem related to the existence of principal types), or by imposing reasonable restrictions, or by checking the flow of polymorphic functions by a static analysis.

On the practical side, by implementing the polymorphic extension of ℂDuce 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 OCaml and ℂDuce 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

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

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

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

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

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

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

[7] R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005.

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

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

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

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

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

[13] C. Okasaki. *Purely Functional Data Structures*. CUP, 1998.

[14] B.C. Pierce and D.N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

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

[16] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL '08*, 2008.

[17] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP '10*, 2010.

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