Polymorphic Functions with Set-Theoretic Types

Part 1: Syntax, Semantics, and Evaluation

Giuseppe Castagna[1] Kim Nguyễn[2] Zhiwu Xu[1,3] Hyeonseung Im[2] Sergueï Lenglet[4] Luca Padovani[5]

[1]CNRS, PPS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France [2]LRI, Université Paris-Sud, Orsay, France
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4] LORIA, Université de Lorraine, Nancy, France [5]Dipartimento di Informatica, Università di Torino, Italy

Abstract. This article is the first part of a two articles series about a calculus with higher-order polymorphic functions, recursive types with arrow and product type constructors and set-theoretic type connectives (union, intersection, and negation). In this first part we define and study the explicitly-typed version of the calculus in which type instantiation is driven by explicit instantiation annotations. In particular, we define an explicitly-typed λ-calculus with intersection types and an efficient evaluation model for it. In the second part, presented in a companion paper, we define a local type inference system that allows the programmer to omit explicit instantiation annotations, and a type reconstruction system that allows the programmer to omit explicit type annotations. The work presented in the two articles provides the theoretical foundations and technical machinery needed to design and implement higher-order polymorphic functional languages for semi-structured data.

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

Keywords Types, polymorphism, XML, intersection types

1. Introduction

The extensible markup language XML is a current standard format for exchanging structured data. Many recent XML processing languages, such as XDuce, ℂDuce, XQuery, OcamlDuce, XHaskell, XAct, are statically-typed functional languages. However, parametric polymorphism, an essential feature of such languages, is still missing, or when present it is in a limited form (no higher-order functions, no polymorphism for XML types, and so on). Polymorphism for XML has repeatedly been requested to and discussed in various working groups of standards (*eg*, RELAX NG [6]) and higher-order functions have been recently proposed in the W3C draft for XQuery 3.0 [9]. Despite all this interest, spurs, and motivations, a comprehensive polymorphic type system for XML was still missing for the simple reason that, until recently, it was deemed unfeasible. A major stumbling block to this research —*ie*, the definition of a subtyping relation for regular tree types with type variables— has been recently lifted by Castagna and Xu [5], who defined and studied a polymorphic subtyping relation for a type system with recursive, product, and arrow types and set-theoretic type connectives (union, intersection, and negation).

In this work we present the next logical step of that research, that is, the definition of a higher-order functional language that takes full advantage of the new capabilities of Castagna and Xu's system. In other words, we define and study a calculus with higher-order polymorphic functions and recursive types with union, intersection, and negation connectives. The approach is thus general and, as such, goes well beyond the simple application to XML processing languages. As a matter of facts, our motivating example developed all along this paper does not involve XML, but looks like a rather classic display of functional programming specimens:

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

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

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

The goal of this work is to define a calculus and a type system that can pass three tests. The first test is that it can define the two functions above. The second, harder, test is that the type system must be able to verify that these functions have the types declared in their signatures. That `map` has the declared type will come as no surprise (in practice, in the second part of this work we show that in the absence of a signature given by the programmer the system can reconstruct a type slightly more precise than this [4]). That `even` was given an intersection type means that it must have all the types that form the intersection. So it must be a function that when applied to an integer it returns a Boolean and that when applied to an argument of a type that does not contain any integer, it returns a result of the same type. In other terms, `even` is a polymorphic (dynamically bounded) overloaded function.

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

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

since `map even` returns a function that when applied to a list of integers it returns a list of Booleans, when applied to a list that

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

POPL '14, January 22–24 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535840

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

Technically speaking, the definition of such a calculus and its type system is difficult for two distinct reasons. First, for the reasons we explain in the next section, it demands to define an explicitly typed λ-calculus with intersection types, a task that, despite many attempts in the last 20 years, still lacked a satisfactory definition. Second, even if working with an explicitly typed setting may seem simpler, the system needs to solve "local type inference"[2], namely, the problem of checking whether the types of a function and of its argument can be made compatible and, if so, of inferring the type of their result as we did for (1). The difficulty, once more, mainly resides in the presence of the intersection types: a term can be given different types either by subsumption (the term is coerced into a super-type of its type) or by instantiation (the term is used as a particular instance of its polymorphic type) and it is typed by the intersection of all these types. Therefore, in this setting, the problem is not just to find a substitution that unifies the domain type of the function with the type of its argument but, rather, a *set* of substitutions that produce instances whose intersections are in the right subtyping relation: our `map even` example should already have given a rough idea of how difficult this is.

The presentation of our work is split in two parts, accordingly: in the first part (this paper) we show how to solve the problem of defining an explicitly-typed λ-calculus with intersection types and how to efficiently evaluate it; in the second part (the companion paper [4]) we will show how to solve the problem of "local type inference" for a calculus with intersection types. In the next section we outline the various problems we met (focusing on those that concern the part of the work presented in this paper) and how they were solved.

2. Problems and overview of the solution

The driver of this work is the definition of an XML processing functional language with high-order polymorphic functions, that is, in particular, a polymorphic version of the language $\mathbb{C}$Duce [2].

$\mathbb{C}$Duce in a nutshell. The essence of $\mathbb{C}$Duce is a λ-calculus with pairs, explicitly-typed recursive functions, and a type-case expression. Types can be recursively defined and include the arrow and product type *constructors* and the intersection, union, and negation type *connectives*. In summary, they are the regular trees coinductively generated by the following productions:

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

where b ranges over basic types (*eg*, `Int`, `Bool`) and $\mathbb{0}$ and $\mathbb{1}$ respectively denote the empty (that types no value) and top (that types all values) types. We use possibly indexed meta-variables s and t to range over types. Coinduction accounts for recursive types. We use the standard convention that infix connectives have a priority higher than constructors and lower than prefix connectives.

From a strictly practical viewpoint, recursive types, products, and type connectives are used to encode regular tree types, which subsume existing XML schema/types while, for what concerns expressions, the type-case is an abstraction of $\mathbb{C}$Duce pattern matching (this uses regular expression patterns on types to define powerful and highly optimized capture primitives for XML data). We initially focus on the functional core and disregard products and recursive functions since the results presented here can be easily extended to them (we show it in the Appendix), though we will freely use them for our examples. So we initially consider the following "Core$\mathbb{C}$Duce" terms:

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

where 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 $\wedge_{i\in I} s_i \to t_i$ that denotes the function of parameter x and body e.

In this work we show how to define the polymorphic extension of this calculus, which can then be easily extended to a full-fledged polymorphic functional language for processing XML documents. But before let us explain the two specificities of the terms in (3), namely, why a type-case expression is included and why we explicitly annotate whole λ-abstractions (with an intersection of arrow types) rather than just their parameters.

The reasons for including a type-case in the terms of the calculus are detailed in [11]: in short, intersection types are used to type overloaded functions, and without a type-case only "coherent overloading" *à la* Forsythe [16] can be defined (which, in our setting, precludes for instance the definition of a non diverging function of type $(\texttt{Int}\to\texttt{Bool}) \wedge (\texttt{Bool}\to\texttt{Int})$). Also, in our system the relation $s_1 \vee s_2 \to t_1 \wedge t_2 \;\leq\; (s_1 \to t_1) \wedge (s_2 \to t_2)$ is, in general, *strict*, and the functions that are in the difference of these two types are those that distinguish non coherent overloading from coherent one. To inhabit this difference we need "real" overloaded functions that execute different code according to the type of their input, whence the need of type-case expressions.

The need of explicitly typed functions is a direct consequence of the introduction of the type-case, because without explicit typing we can run into paradoxes such as $\mu f.\lambda x.f \in (\mathbb{1}\to\texttt{Int})\,?\,\texttt{true} : \texttt{42}$, a recursively defined (constant) function that has type $\mathbb{1}\to\texttt{Int}$ if and only if it *does not* have type $\mathbb{1}\to\texttt{Int}$. In order to decide whether the function above is well-typed or not, we must explicitly give a type to it. For instance, the function is well-typed if it is explicitly assigned the type $\mathbb{1} \to \texttt{Int}\vee\texttt{Bool}$. This shows both that functions must be explicitly typed and that specifying not only the type of parameters but also the type of the result is strictly more expressive, as more terms can be typed.

In summary, we need to define an explicitly typed language with intersection types. This is a difficult problem for which no full-fledged solution existed, yet: there exist only few intersection type systems with explicitly typed terms, and none of them is completely satisfactory (see Section 7 on related work). To give an idea of why this is difficult, imagine we adopt for functions a Church-style notation as $\lambda x^t.e$ and consider the following "switch" function $\lambda x^t.\,(x \in \texttt{Int}\,?\,\texttt{true} : \texttt{42})$ that when applied to an `Int` returns `true` and returns `42` otherwise. Intuitively, we want to assign to this function the type $(\texttt{Int}\to\texttt{Bool}) \wedge (\neg\texttt{Int}\to\texttt{Int})$, the type of a function that when applied to an `Int`, returns a `Bool`, and when applied to a value which is not an `Int`, returns an `Int`. For the sake of presentation, let us say that we are happy to deduce for the function above the less precise type $(\texttt{Int}\to\texttt{Bool}) \wedge (\texttt{Bool}\to\texttt{Int})$ (which is a super-type of the former since if a function maps anything that is not an `Int` into an `Int`—it has type $\neg\texttt{Int}\to\texttt{Int}$—, then in particular it maps Booleans to integers —*ie*, it has also type $\texttt{Bool}\to\texttt{Int}$). The problem is to determine which type we should use for t in the "switch" function above. If we use, say, $\texttt{Int}\vee\texttt{Bool}$, then under the hypothesis that $x : \texttt{Int}\vee\texttt{Bool}$ the type deduced for the body of the function is $\texttt{Int}\vee\texttt{Bool}$. So the best type we can give to the switch function is $\texttt{Int}\vee\texttt{Bool} \to \texttt{Int}\vee\texttt{Bool}$ which is far less precise than the sought intersection type, insofar as it does not

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

make any distinction between arguments of type Int and those of type Bool.

The solution, which was introduced by ℂDuce, is to explicitly type —by an intersection type— whole λ-abstractions instead of just their parameters: $\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})}x\,.\,(x{\in}\texttt{Int}\,?\,\texttt{true}:\texttt{42})$ In doing so we also explicitly define the result type of functions which, as we have just seen, increases the expressiveness of the calculus. Thus the general form of λ-abstractions is, as stated by the grammar in (3), $\lambda^{\wedge_{i\in I}\,s_i\to t_i}x.e$. Such a term is well typed if for all $i \in I$ from the hypothesis that x has type s_i it is possible to deduce that e has type t_i. Unfortunately, with polymorphic types, this simple solution introduced by ℂDuce no longer suffices.

Polymorphic extension. The novelty of this work is to allow type variables (ranged over by lower-case Greek letters: α, β, ...) to occur in the types in (2) and, thus, in the types labeling λ-abstractions in (3). It becomes thus possible to define the polymorphic identity function as $\lambda^{\alpha\to\alpha}x.x$, while classic "auto-application" term can be 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 the types obtained by instantiating the type variables occurring in the label. So $\lambda^{\alpha\to\alpha}x.x$ has not only type $\alpha\to\alpha$ but also, for example, by subsumption the types $\mathbb{0}\to\mathbb{1}$ (the type of all functions, which is a super-type of $\alpha\to\alpha$) and $\neg\texttt{Int}$ (since every well-typed λ-abstraction is not an integer, then $\neg\texttt{Int}$ contains —*ie*, is a super-type of— all function types), and by instantiation the types $\texttt{Int}\to\texttt{Int}$, $\texttt{Bool}\to\texttt{Bool}$, etc.

The use of instantiation in combination with intersection types has nasty consequences, for if a term has two distinct types, then it has also their intersection type (*eg*, $\lambda^{\alpha\to\alpha}x.x$ has type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})\wedge\neg\texttt{Int}$). In the monomorphic case a term can have distinct types only by subsumption and, thus, intersection types are transparently assigned to terms via subsumption. But in the polymorphic case this is no longer possible: a term can be typed by the intersection of two distinct instances of its polymorphic type which, in general, are not in any subtyping relation with the latter: for instance, $\alpha\to\alpha$ is neither a subtype of $\texttt{Int}\to\texttt{Int}$ nor vice versa, since the subtyping relation must hold for *all possible* instantiations of α and there are infinitely many instances of $\alpha\to\alpha$ that are neither a subtype nor a super-type of $\texttt{Int}\to\texttt{Int}$.

Explicit instantiation. Concretely, if we want to apply the polymorphic identity $\lambda^{\alpha\to\alpha}x.x$ to, say, 42, then the particular instance obtained by the type-substitution $\{\texttt{Int}/\alpha\}$ (denoting the replacement of every occurrence of α by Int) must be used, that is $(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)\texttt{42}$. We have thus to *relabel* the type decorations of λ-abstractions before applying them. In implicitly typed languages, such as ML, the relabeling is meaningless (no type decoration is used in terms) while in their explicitly-typed counterparts relabeling can be seen as a logically meaningful but computationally useless operation, insofar as execution takes place on type erasures (*ie*, the terms obtained by erasing all type decorations). In the presence of type-case expressions, however, relabeling is necessary since the label of a λ-abstraction determines its type: testing whether an expression has type, say, $\texttt{Int}\to\texttt{Int}$ should succeed for the application of $\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x$ to 42 and fail for its application to true. This means that, in Reynolds' terminology, our terms have an *intrinsic* meaning [17], that is to say, the semantics of a term depends on its typing. If we need to relabel some function, then it may be necessary to relabel also its body as witnessed by the following "daffy" —though well-typed— definition of the identity function:

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

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

The solution is to apply the relabeling also to the body of the function. Here what "to relabel the body" means is straightforward: apply the same type-substitution $\{\texttt{Int}/\alpha\}$ to the body. This yields a reductum $(\lambda^{\texttt{Int}\to\texttt{Int}}y.\texttt{3})\texttt{3}$ which is well typed. In general, however, the way to perform a relabeling of the body of a function is not so straightforward and clearly defined, since two different problems may arise: (i) it may be necessary to apply more than a single type-substitution and (ii) the relabeling of the body may depend on the dynamic type of the actual argument of the function (both problems are better known as —or are instances of— the problem of determining expansions for intersection type systems [7]). Next, we discuss each problem in detail.

Multiple substitutions. First of all, notice that we may need to relabel/instantiate functions not only when they are applied but also when they are used as arguments. For instance, consider a function that expects arguments of type $\texttt{Int}\to\texttt{Int}$. It is clear that we can apply it to the identity function $\lambda^{\alpha\to\alpha}x.x$, since the identity function *has* type $\texttt{Int}\to\texttt{Int}$ (feed it by an integer and it will return an integer). Before, though, we have to relabel the latter by the substitution $\{\texttt{Int}/\alpha\}$ yielding $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$. As the identity $\lambda^{\alpha\to\alpha}x.x$ has type $\texttt{Int}\to\texttt{Int}$, so it has type $\texttt{Bool}\to\texttt{Bool}$ and, therefore, the intersection of the two: $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$. So we can apply a function that expects an argument of this intersection type to our identity function. The problem is now how to relabel $\lambda^{\alpha\to\alpha}x.x$. Intuitively, we have to apply two distinct type-substitutions $\{\texttt{Int}/\alpha\}$ and $\{\texttt{Bool}/\alpha\}$ to the label of the λ-abstraction and replace it by the intersection of the two instances. This corresponds to relabel the polymorphic identity from $\lambda^{\alpha\to\alpha}x.x$ into $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$. This is the solution adopted by this work, where we manipulate *sets of type-substitutions* —delimited by square brackets. The application of such a set (*eg*, in the previous example $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$) to a type t returns the intersection of all types obtained by applying each substitution in the set to t (*eg*, in the example $t\{\texttt{Int}/\alpha\}\wedge t\{\texttt{Bool}/\alpha\}$). Thus the first problem has an easy solution.

Relabeling of function bodies. The second problem is much harder and concerns the relabeling of the body of a function. While the naive solution consisting of propagating the application of type-substitutions to the bodies of functions works for single type-substitutions, in general, it fails for *sets* of type-substitutions. This can be seen by considering the relabeling via the set of type-substitutions $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$ of the daffy function in (4). If we apply the naive solution, this yields

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

which is not well typed. That this term is not well typed is clear if we try applying it to, say, 3: the application of a function of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ to an Int should have type Int, but here it reduces to $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.\texttt{3})\texttt{3}$, and there is no way to deduce the intersection type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ for the constant function $\lambda y.\texttt{3}$. But we can also directly verify that it is not well typed, by trying typing the function in (5). This corresponds to prove that under the hypothesis x : Int the term $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$ has type Int, and that under

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

the hypothesis x : Bool this same term has type Bool. Both checks fail because, in both cases, $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is ill-typed (it neither has type Int→Int when x:Bool, nor has it type Bool→Bool when x:Int). This example shows that in order to ensure that relabeling yields well-typed terms, the relabeling of the body *must change* according to the type of the value the parameter x is bound to. More precisely, $(\lambda^{\alpha\to\alpha}y.x)$ should be relabeled as $\lambda^{\texttt{Int}\to\texttt{Int}}y.x$ when x is of type Int, and as $\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x$ when x is of type Bool. An example of this same problem less artificial than our daffy function is given by the classic apply function $\lambda f.\lambda x.fx$ which, with our polymorphic type annotations, is written as:

$$\lambda^{(\alpha\to\beta)\to\alpha\to\beta}f.\lambda^{\alpha\to\beta}x.fx \tag{6}$$

The apply function in (6) has type $(\texttt{Int}\to\texttt{Int})\to\texttt{Int}\to\texttt{Int}$, obtained by instantiating its type annotation by the substitution $\{\texttt{Int}/\alpha, \texttt{Int}/\beta\}$, as well as type $(\texttt{Bool}\to\texttt{Bool})\to\texttt{Bool}\to\texttt{Bool}$, obtained by the substitution $\{\texttt{Bool}/\alpha, \texttt{Bool}/\beta\}$. If we want to feed this function to another function that expects arguments whose type is the intersections of these two types, then we have to relabel it by using the set of type-substitutions $[\{\texttt{Int}/\alpha, \texttt{Int}/\beta\}, \{\texttt{Bool}/\alpha, \texttt{Bool}/\beta\}]$. But, once more, it is easy to verify that the naive solution that consists in propagating the application of the set of type-substitutions down to the body of the function yields an ill-typed expression.

This second problem is the showstopper for the definition of an explicitly typed λ-calculus with intersection types. Most of the solutions found in the literature [3, 13, 18, 21] rely on the duplication of lambda terms and/or typing derivations, while other calculi such as [22] that aim at avoiding such duplication obtain it by adding new expressions and new syntax for types (see related work in Section 7); but none of them is able to produce an explicitly-typed λ-calculus with intersection types, as we do, by just adding annotations to λ-abstractions.

Our solution. Here we introduce a new technique that consists in performing a "lazy" relabeling of the bodies. This is obtained by decorating λ-abstractions by (sets of) type-substitutions. For example, in order to pass our daffy identity function (4) to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$, we first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x). \tag{7}$$

The new annotation in the outer "λ" indicates that the function must be relabeled and, therefore, that we are using the particular instance whose type is the one in the interface (*ie*, $\alpha\to\alpha$) to which we apply the set of type-substitutions. The relabeling will be actually propagated to the body of the function at the moment of the reduction, only if and when the function is applied (relabeling is thus lazy). However, the new annotation is statically used by the type system to check soundness. Notice that, unlike existing solutions, we preserve the structure of λ-terms (at the expenses of some extra annotation that is propagated during the reduction) which is of the uttermost importance in a language-oriented study.

In this paper we focus on the study of the calculus with these "lazy" type-substitutions annotations. We temporarily avoid the problem of local type inference by defining a *calculus with explicit sets of type substitutions*: expressions will be explicitly annotated with appropriate sets of type-substitutions.

Polymorphic $\mathbb{C}$Duce. From a practical point of view, however, it is important to stress that, at the end, these annotations will be invisible to the programmer and, as we show in the second part presented in the companion paper [4], all the necessary type-substitutions will be inferred statically. In practice, the programmer will program in the language defined by grammar (3), but where the types that annotate λ's may contain type variables, that is, the polymorphic version of $\mathbb{C}$Duce. The problem of inferring explicit sets of type-substitutions to annotate the polymorphic version of the expressions in (3) is the topic of the second part of this work presented in the companion paper [4]. For the time being, simply notice that the language defined by (3) and extended with type variables passes our first test inasmuch as the even function can be defined as follows (where $s\backslash t$ is syntactic sugar for $s\wedge\neg t$):

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\backslash\texttt{Int}\to\alpha\backslash\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x \tag{8}$$

while —with the products and recursive functions described in the Appendix— map is defined as (see also discussion in Appendix E)

$$\mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\,f = \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell)) \tag{9}$$

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 $X = (\alpha, X)\vee\texttt{nil}$.

When fed by any expression of this language, the type inference system defined in the companion paper [4] will infer sets of type-substitutions and insert them into the expression to make it well typed (if possible, of course). For example, for the application (of the terms defining) map to even, the inference system of the companion paper [4] infers the following set of type-substitutions $[\{(\alpha\backslash\texttt{Int})/\alpha, (\alpha\backslash\texttt{Int})/\beta\}, \{\alpha\vee\texttt{Int}/\alpha, (\alpha\backslash\texttt{Int})\vee\texttt{Bool}/\beta\}]$ and textually inserts it between the two terms (so that the type-substitutions apply to the type variables of map) yielding the typing in (1). Finally, as we explain in Section 5.3 later on, the compiler will compile the expression into an expression of an intermediate language that can be evaluated as efficiently as the monomorphic calculus.

Outline. The rest of the presentation proceeds as follows. In Section 3 we define and study our calculus with explicit type-substitutions: we define its syntax, its operational semantics, and its type system; we prove that the type system is sound and subsumes classic intersection type systems. In Section 4 we define an algorithm for type inference and prove that it is sound, complete, and terminating. In Section 5 we show that the addition of type-substitutions has in practice no impact on the efficiency of the evaluation since the calculus can be compiled into an intermediate language that executes as efficiently as monomorphic $\mathbb{C}$Duce. Section 7 presents related work and in Section 8 we conclude our presentation. In the rest of the presentation we will focus on the intuition and try to avoid as many technical details as possible. We dot the i's and cross the t's in the Appendix, where all formal definitions and complete proofs of properties can be found (*n.b.*: references in the text starting by capital letters —*eg*, Definition A.7— refer to this appendix). All these as well as other details can also be found in the third author's PhD thesis manuscript [23].

Contributions. The overall contribution of our work is the definition of a statically-typed core language with (i) polymorphic higher-order functions for a type system with recursive types and union, intersection, and negation type connectives, (ii) an efficient evaluation model, (iii) local type inference for application, and (iv) a limited form of type reconstruction.

The main contribution of this first part of the work is the definition of an *explicitly-typed* λ-calculus (actually, a family of calculi) with intersection (and union and negation) types and of its efficient evaluation via the compilation into an intermediate language. From a syntactic viewpoint our solution is a minimal extension since it just requires to add annotations to λ-abstractions of the untyped λ-calculus (*cf.* Section 3.5). Although this problem has been studied for over 20 years, no existing solution proposes such a minimal extension, which is of paramount importance in a programming language-oriented study (see related works in Section 7).

The technical contributions are the definition of an explicitly typed calculus with intersection types; the proof that it subsumes existing intersection type systems; the soundness of its type system, the definition of a sound, complete and terminating algorithm for type inference (which as byproduct yields an intersection type proof system satisfying the Curry-Howard isomorphism); the definition

of a compilation technique into an intermediate language that can be evaluated as efficiently as the monomorphic one; its extension to the so called `let`-polymorphism and the proof of the adequacy of the compilation. Local type inference for application and type reconstruction are studied in the second part of this work presented in the companion paper [4].

3. A calculus with explicit type-substitutions

The types of the calculus are those in the grammar (2) to which we add type variables (ranged over by α) and, for the sake of presentation, stripped of product types. In summary, types are the regular trees coinductively generated by

$$t ::= \alpha \mid b \mid t \to t \mid t \wedge t \mid t \vee t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \tag{10}$$

and such that every infinite branch contains infinitely many occurrences of type constructors. We use $\mathscr{T}$ to denote the set of all types. 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 (that is, strongly normalizing). 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 (see Definition A.2). A type t is said to be *ground* or *closed* if and only if $\mathsf{var}(t)$ is empty.

The subtyping relation for the types in $\mathscr{T}$ is the one defined by Castagna and Xu [5]. For this work it suffices to consider that ground types are interpreted as sets of *values* (*n.b.*, just values, not expressions) that have that type and subtyping is set containment (*ie*, a ground type s is a subtype of a ground type t if and only if t contains all the values of type s). In particular, $s\to t$ contains all λ-abstractions that when applied to a value of type s, if they return a result, then this result is of type t (so $\mathbb{0}\to\mathbb{1}$ is the set of all functions and $\mathbb{1}\to\mathbb{0}$ is the set of functions that diverge on all arguments). Type connectives (union, intersection, negation) are interpreted as the corresponding set-theoretic operators and subtyping is set containment. For what concerns non-ground types (*ie*, types with variables occurring in them) all the reader needs to know for this work is that the subtyping relation of Castagna and Xu is preserved by type-substitutions. 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 [5]). Two types are equivalent if they denote the same set of values, that is, if they are subtype one of each other (type equivalence is denoted by $\simeq$). An important property of this system we will often use is 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 their negations. An intersection is uniform if all the literals have 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

$$\bigwedge_{p\in P} b_p \wedge \bigwedge_{n\in N} \neg b_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \tag{11}$$

or

$$\bigwedge_{p\in P} (s_p\to t_p) \wedge \bigwedge_{n\in N} \neg(s_n\to t_n) \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \tag{12}$$

When either P' or N' is not empty, we call the variables α_q's and α_r's the *top-level variables* of the normal form.

3.1 Expressions

Expressions are derived from those of Core$\mathbb{C}$Duce (with type variables in types) with the addition that sets of explicit type-substitutions (ranged over by $[\sigma_j]_{j\in J}$) may be applied to terms and decorate λ-abstractions

$$e ::= c \mid x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e\in t\,?\,e : e \mid e[\sigma_j]_{j\in J} \tag{13}$$

and with the restriction that the type tested in type-case expressions is closed. 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 the empty substitution. 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 expressions variables* of the expression e; we use $\mathsf{tv}(e)$ to denote the set of *type variables* occurring in e (see Definition A.9).

As customary, we assume bound expression variables to be pairwise distinct and distinct from any free expression variable occurring in the expressions under consideration. We equate expressions up to the α-renaming of their bound expression variables. In particular, when substituting an expression e for a variable y in an expression e' (see Definition A.11), we assume that the bound variables of e' are distinct from the bound and free variables of e, to avoid unwanted captures. For example, $(\lambda^{\alpha\to\alpha} x.x)y$ is α-equivalent to $(\lambda^{\alpha\to\alpha} z.z)y$.

The situation is a bit more complex for type variables, as we do not have an explicit binder for them. Intuitively, a type variable can be α-converted if it is a *polymorphic* one, that is, if it can be instantiated. For example, $(\lambda^{\alpha\to\alpha} x.x)y$ is α-equivalent to $(\lambda^{\beta\to\beta} x.x)y$, and $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.x)y$ is α-equivalent to $(\lambda^{\beta\to\beta}_{[\{\mathtt{Int}/\beta\}]} x.x)y$. 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 variable is called *monomorphic*. For example, the following expression

$$\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)} y.((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]y)$$

is not sound (*ie*, it cannot be typed), because α is bound at the level of the outer abstraction, not at level of the inner one. Consequently, in this expression, α is monomorphic for the inner abstraction, but polymorphic for the outer one (strictly speaking, thus, the monomorphic and polymorphic adjectives apply to occurrences of variables rather than variables themselves). Monomorphic type variables cannot be α-converted: $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)} y.(\lambda^{\alpha\to\alpha} x.x)y$ is *not* α-equivalent to $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)} y.(\lambda^{\beta\to\beta} x.x)y$ (but it is α-equivalent to $\lambda^{(\beta\to\beta)\to(\beta\to\beta)} y.(\lambda^{\beta\to\beta} x.x)y$). Note that the scope of polymorphic variables may include some type-substitutions $[\sigma_i]_{i\in I}$: for example, $((\lambda^{\alpha\to\alpha} x.x)y)[\mathtt{Int}/\alpha]$ is α-equivalent to $((\lambda^{\beta\to\beta} x.x)y)[\mathtt{Int}/\beta]$. Finally, we have to be careful when performing expression substitutions and type-substitutions to avoid clashes of polymorphic variable namespaces. For example, substituting $\lambda^{\alpha\to\alpha} z.z$ for y in $\lambda^{\alpha\to\alpha} x.x\ y$ would lead to an unwanted capture of α (assuming α is polymorphic, that is, not bound by a λ-abstraction placed above these two expressions), so we have to α-convert one of them, so that the result of the substitution is, for instance, $\lambda^{\alpha\to\alpha} x.x\ (\lambda^{\beta\to\beta} z.z)$.

To resume, we assume polymorphic variables to be pairwise distinct and distinct from any monomorphic variable in the expressions under consideration. We equate expressions up to α-renaming of their polymorphic variables. In particular, when substituting an expression e' for a variable x in an expression e, we suppose the

$$\textit{(subsum)}\ \frac{\Delta\wp\Gamma\vdash e:t_1 \quad t_1\leq t_2}{\Delta\wp\Gamma\vdash e:t_2} \qquad \textit{(appl)}\ \frac{\Delta\wp\Gamma\vdash e_1:t_1\to t_2 \quad \Delta\wp\Gamma\vdash e_2:t_1}{\Delta\wp\Gamma\vdash e_1e_2:t_2}$$

$$\textit{(abstr)}\ \frac{\Delta\cup\mathsf{var}(\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j)\wp\Gamma,(x:t_i\sigma_j)\vdash e@[\sigma_j]:s_i\sigma_j}{\Delta\wp\Gamma\vdash\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e:\bigwedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j}\ \begin{array}{l}i\in I\\ j\in J\end{array}$$

$$\textit{(var)}\ \frac{}{\Delta\wp\Gamma\vdash x:\Gamma(x)} \qquad \textit{(case)}\ \frac{\Delta\wp\Gamma\vdash e:t' \quad \begin{cases} t'\not\leq\neg t & \Rightarrow\ \Delta\wp\Gamma\vdash e_1:s\\ t'\not\leq t & \Rightarrow\ \Delta\wp\Gamma\vdash e_2:s\end{cases}}{\Delta\wp\Gamma\vdash(e\in t\,?\,e_1:e_2):s}$$

$$\textit{(inst)}\ \frac{\Delta\wp\Gamma\vdash e:t \quad \sigma\sharp\Delta}{\Delta\wp\Gamma\vdash e[\sigma]:t\sigma} \qquad \textit{(inter)}\ \frac{\forall j\in J.\,\Delta\wp\Gamma\vdash e[\sigma_j]:t_j}{\Delta\wp\Gamma\vdash e[\sigma_j]_{j\in J}:\bigwedge_{j\in J}t_j}\ |J|>1$$

Figure 1. Static semantics

polymorphic type variables of e to be distinct from the monomorphic and polymorphic type variables of e' thus avoiding unwanted captures. Detailed definitions are given in Appendix A.2.

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

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

where $\circ$ denotes the pairwise composition of all substitutions of the two sets (see Definition A.7). It erases the set of type-substitutions when e is a variable and it is homomorphically applied on the remaining expressions (see Definition A.12).

3.2 Operational semantics

The dynamic semantics is given by the following three notions of reduction (where v ranges over *values*, that is, constants and λ-abstractions), applied by a leftmost-outermost strategy:

$$e[\sigma_j]_{j\in J} \quad\rightsquigarrow\quad e@[\sigma_j]_{j\in J} \tag{14}$$

$$(\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e)v \quad\rightsquigarrow\quad (e@[\sigma_j]_{j\in P})\{v/x\} \tag{15}$$

$$v\in t\,?\,e_1:e_2 \quad\rightsquigarrow\quad \begin{cases} e_1 & \text{if } \vdash v:t\\ e_2 & \text{otherwise}\end{cases} \tag{16}$$

where in (15) we have $P\stackrel{\text{def}}{=}\{j\in J\mid\exists i\in I,\vdash v:t_i\sigma_j\}$.

The first rule (14) performs relabeling, that is, it propagates the sets of type-substitutions down into the decorations of the outermost λ-abstractions. The second rule (15) states the semantics of applications: this is standard call-by-value β-reduction, with the difference that the substitution of the argument for the parameter is performed on the relabeled body of the function. Notice that relabeling depends on the type of the argument and keeps only those 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 will ensure that P is never empty). For instance, take the daffy identity function, instantiate it as in (7) 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 (16) checks whether the value returned by the expression in the type-case matches the specified type and selects the branch accordingly.

The reader may think that defining a functional language in which each β-reduction must perform an involved relabeling operation, theoretically interesting though it may be, will result in practice too costly and therefore unrealistic. This is not so. In Section 5 we show that this reduction can be implemented as efficiently as in $\mathbb{C}$Duce. By a smart definition of closures it is possible to compute relabeling in a lazy way and materialize it only in a very specific case for the reduction of the type-case (*ie*, to perform a type-case reduction (16) where the value v is a function whose interface contains monomorphic type variables and it is the result of the partial application of a polymorphic function) while all other reductions for applications can be implemented as plain classic β-reduction. For instance, to evaluate the expressions $(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)\texttt{42}$ above, we can completely disregard all type annotations and decorations and perform a couple of standard β reductions that yield the result `42`.

3.3 Type system

As expected in a calculus with a type-case expression, the dynamic semantics depends on the static semantics —precisely, on the typing of values. The static semantics of our calculus is defined in Figure 1. The judgments are of the form $\Delta\wp\Gamma\vdash e:t$, where e is an expression, t a type, Γ a type environment (*ie*, a finite mapping from expression variables to types), and Δ a finite set of type variables. The latter is the set of all *monomorphic type variables*, that is, the variables that occur in the type of some outer λ-abstraction and, as such, cannot be instantiated; it must contain all the type variables occurring in Γ.

The rules for application and subsumption are standard. In the latter, the subtyping relation is the one defined in [5]. We just omitted the rule for constants (which states that c has type b_c).

The rule for abstractions applies each substitution specified in the decoration to each arrow type in the interface, adds all the variables occurring in these types to the set of monomorphic type variables Δ, and checks whether the function has all the resulting types. Namely, it checks that for every possible input type, the (relabeled) body has the corresponding output type. To that end, it applies each substitution σ_j in the decoration to each input type t_i of the interface and checks that, under the hypothesis that x has type $t_i\sigma_j$, the function body relabeled with the substitution σ_j at issue has type $s_i\sigma_j$ (notice that all these checks are performed under the same updated set of monomorphic type variables, that is, $\Delta\cup\mathsf{var}(\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j)$). If the test succeeds, then the rule infers for the function the type obtained by applying the set of substitutions of the decoration to the type of the interface. For example, in the case of the instance of the daffy identity function given in (7), the Δ is always empty and the rule checks whether under the hypothesis $x:\alpha\{\texttt{Int}/\alpha\}$ (*ie*, $x:\texttt{Int}$), it is possible to deduce that $(\lambda^{\alpha\to\alpha}y.x)x@[\{\texttt{Int}/\alpha\}]$ has type $\alpha\{\texttt{Int}/\alpha\}$ (*ie*, that $(\lambda^{\texttt{Int}\to\texttt{Int}}y.x)x:\texttt{Int}$), and similarly for the substitution $\{\texttt{Bool}/\alpha\}$. The type deduced for the function is then $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$. The relabeling of the body in the premises of the rule (*abstr*) is a key mechanism of the type system: had we used $e[\sigma_j]$ instead of $e@[\sigma_j]$ in the premises of the (*abstr*) rule, the expression (7) could not be typed. The reason is that $e[\sigma_j]$ is more demanding on typing than $e@[\sigma_j]$, since the well typing of e is necessary to the well-typing of the former but not to that of the

latter. Indeed while under the hypothesis $x : \texttt{Int}$ we just showed that $((\lambda^{\alpha\to\alpha} y.x)x)@[\{\texttt{Int}/\alpha\}]$ —*ie*, $((\lambda^{\texttt{Int}\to\texttt{Int}} y.x)x)$— is well-typed, the term $((\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/\alpha\}]$ is not, for $(\lambda^{\alpha\to\alpha} y.x)$ does not have type $\alpha\to\alpha$. The rule for abstractions also justifies the need for an explicit set Δ for monomorphic type variables while, for instance, in ML it suffices to consider monomorphic type variables that occur in the image of Γ [15]: when checking an arrow of the interface of a function, the variables occurring in the other arrows must be considered monomorphic, too.

To type the applications of a set of type-substitutions to an expression, two different rules are used according to whether the set contains one or more than one substitution. When a single substitution is specified, the rule (*inst*) instantiates the type according to the specified substitution, provided that σ does not substitute variables in Δ (*ie*, $\mathsf{dom}(\sigma) \cap \Delta = \varnothing$, noted $\sigma\sharp\Delta$). If more than one substitution is specified, then the rule (*inter*) composes them by an intersection.

Finally, the (*case*) rule first infers the type t' of the expression whose type is tested. Then the type of each branch e_i is checked only if there is a chance that the branch can be selected. Here the use of "$\not\leq$" is subtle but crucial (it allows us to existentially quantify over type-substitutions). The branch, say, e_1 can be selected (and therefore its well-typedness must be checked) only if e can return a value that is in t. But in order to cover all possible cases we must also consider the case in which the type of e is instantiated as a consequence of an outer application. A typical usage pattern (followed also by our `even` function) is $\lambda^{\alpha\to\cdots} x.\ x{\in}\texttt{Int}\,?\,e_1 : e_2$: the branch e_1 is selected only if the function is applied to a value of type `Int`, that is, if the type α of x is instantiated to `Int` (notice that when typing the body of the function Δ contains only α). More generally, the branch e_1 in $e{\in}t\,?\,e_1 : e_2$ can be selected only if e can return a value in t, that is to say, if there exists a substitution σ for any type variables *even those in* Δ such as the intersection of $t'\sigma$ and t is not empty (t is a closed, so $t\sigma = t$). Therefore, in order to achieve maximum precision the rule (*case*) must check $\Delta \fatsemi \Gamma \vdash e_1 : s$ only if there exists σ such that $t'\sigma \wedge t \not\simeq \mathbb{0}$. Since $t' \leq \neg t$ (strictly) implies that for all substitutions σ, $t'\sigma \leq \neg t$ (recall that t is a closed type), then by the contrapositive the existence of a substitution σ such that $t'\sigma \not\leq \neg t$ implies $t' \not\leq \neg t$. The latter is equivalent to $t' \wedge t \not\simeq \mathbb{0}$: the intersection of t and t' is not empty. So we slightly over-approximate the test of selection and check the type of e_1 under the weaker hypothesis $t' \wedge t \not\simeq \mathbb{0}$ which ensures that the typing will hold also under the stronger (and sought) hypothesis that there exists σ such that $t'\sigma \wedge t \not\simeq \mathbb{0}$ (the difference only matters with some specific cases involving *indivisible* types: see [5]).

Notice that explicit type-substitutions are only needed to type applications of polymorphic functions. Since no such application occurs in the bodies of `map` and `even` as defined in Section 2 (the m and f inside the body of `map` are abstracted variables and, thus, have monomorphic types), then they can be typed by this system as they are (as long as they are not applied one to the other there is no need to infer any set of type-substitutions). So we can already see that our language passes the second test, namely, that `map` and `even` have the types declared in their signatures. Let us detail just the most interesting case, that is, the typing of the term `even` defined in equation (8) (even though the typing of the type-case in (9), the term defining `map`, is interesting, as well). According to the rule (*abstr*) we have to check that under the hypothesis $x{:}\texttt{Int}$ the expression $x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x$ has type `Bool`, and that under the hypothesis $x : \alpha\backslash\texttt{Int}$ the same expression has type $\alpha\backslash\texttt{Int}$. So we have two distinct applications of the (*case*) rule. In one x is of type `Int`, thus the check $\texttt{Int} \not\leq \texttt{Int}$ fails, and therefore only the first branch, $(x \bmod 2) = 0$, is type checked (the second is skipped). Since under the hypothesis $x : \texttt{Int}$ the expression $(x \bmod 2) = 0$ has type `Bool`, then so has the whole type-case expression. In the other application of (*case*), x is of type $\alpha\backslash\texttt{Int}$, so the test $\alpha\backslash\texttt{Int} \not\leq \neg\texttt{Int}$ clearly fails, and only the second branch is checked (the first is skipped). Since this second branch is x, then the whole type-case expression has type $\alpha\backslash\texttt{Int}$, as expected. This example shows two important aspects of our typing rules. First, it shows the importance of Δ to record monomorphic variables, since it may contain some variables that do not occur in Γ. For instance, when typing the first branch of `even`, the type environment contains only $x : \texttt{Int}$ but Δ is $\{\alpha\}$ and this forbids to consider α as polymorphic (if we allowed to instantiate any variable that does not occur in Γ, then the term obtained from the `even` function (8) by replacing the first branch by $(\lambda^{\alpha\to\alpha} y.y)[\{\texttt{Bool}/\alpha\}]\texttt{true}$ would be well-typed, which is wrong since α is monomorphic in the body of `even`). Second, this example shows why if in some application of the (*case*) rule a branch is not checked, then the type checking of the whole type-case expression must not necessarily fail: the well-typing of this branch may be checked under different hypothesis (typically when occurring in the body of an overloaded function).[4] The reader can refer to Section 3.3 of [11] for a more detailed discussion on this point.

Finally, notice that the rule (*subsum*) makes the type system dependent on the subtyping relation $\leq$ defined in [5]. It is important not to confuse the subtyping relation $\leq$ of our system, which denotes semantic subtyping (*ie*, set-theoretic inclusion of denotations), with the one typically used in the type reconstruction systems for ML, which stands for type variable instantiation. For example, in ML we have $\alpha \to \alpha \leq \texttt{Int} \to \texttt{Int}$ (because $\texttt{Int} \to \texttt{Int}$ is an instance of $\alpha \to \alpha$). But this is not true in our system, as the relation must hold for *every possible instantiation of* α, thus in particular for α equal to `Bool`. In the companion paper [4] we define the preorder $\sqsubseteq_\Delta$ which includes the type variable instantiation of the preorder typically used for ML, so any direct comparison with constraint systems for ML types should focus on $\sqsubseteq_\Delta$ rather than $\leq$ and it can be found in the companion paper [4].

3.4 Type soundness

Subject reduction and progress properties hold for this system.

Theorem 3.1 (Subject Reduction)**.** *For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.*

Theorem 3.2 (Progress)**.** *Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.*

The proofs of both theorems, though unsurprising, are rather long and technical and can be found in Appendix B.2. They allow us to conclude that the type system is sound.

Corollary 3.3 (Type soundness)**.** *Let e be a well-typed closed expression, that is, $\vdash e : t$ for some t. Then either e diverges or it returns a value of type t.*

3.5 Expressing intersection type systems

We can now state the first stand-alone theoretical contribution of our work. Consider the sub-calculus of our calculus in which type-substitutions occur only in decorations and without constants and type-case expressions, that is,

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \tag{17}$$

and whose types are *inductively* produced by the grammar

$$t ::= \alpha \mid t \to t \mid t \wedge t$$

[4] From a programming language point of view it is important to check that during type checking every branch of a given type-case expression is checked —*ie*, it can be selected— at least once. This corresponds to checking the absence of redundant cases in pattern matching. We omitted this check since it is not necessary for formal development.

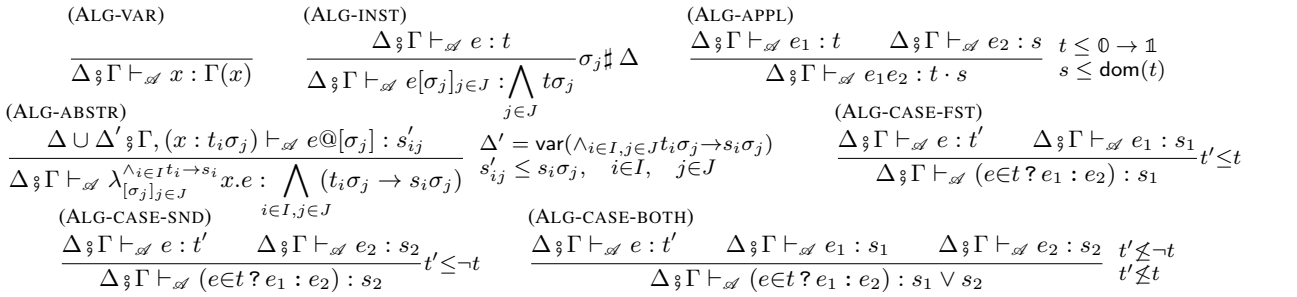

Figure 2. Typing algorithm

This calculus is closed with respect to β-reduction as defined by the reduction rule (15) (without type-cases, union and negation types are not needed). It constitutes an explicitly-typed λ-calculus with intersection types whose expressive power subsumes that of classic intersection type systems (without an universal element ω, of course), as expressed by the following theorem.

Theorem 3.4. *Let $\vdash_{BCD}$ denote provability in the Barendregt, Coppo, and Dezani system [1], and $\lceil e \rceil$ be the pure λ-calculus term obtained from e by erasing all types occurring in it.*

If $\vdash_{BCD} m : t$, then $\exists e$ such that $\vdash e : t$ and $\lceil e \rceil = m$.

Therefore, this sub-calculus solves a longstanding open problem, that is the definition of explicit type annotations for λ-terms in intersection type systems, without any further syntactic modification. See Section 7 on related work for an extensive comparison.

The proof of Theorem 3.4 is constructive (*cf.*, Appendix B.3). Therefore we can transpose decidability results of intersection type systems to our system. In particular, type reconstruction for the sub-calculus (17) is undecidable and this implies the undecidability of type reconstruction for the whole calculus *without recursive types* (with recursive types type reconstruction is trivially decidable since every λ-term can be typed by the recursive type $\mu X.(X \to X) \vee *$). In Section 4 we prove that type inference for our system is decidable. The problem of reconstructing type-substitutions (*ie*, given a term of grammar (3), deciding whether it is possible to add sets of type-substitutions in it so that it becomes a well-typed term of our calculus) is dealt with in the companion paper [4].

To compare with existing intersection type systems, the calculus in (17) includes neither type-cases nor expressions of the form $e[\sigma_j]_{j\in J}$. While it is clear that type-cases increase the expressive power of the calculus, one may wonder whether the same is true for $e[\sigma_j]_{j\in J}$. This is not so as expressions $e[\sigma_j]_{j\in J}$ are redundant. Consider the subcalculus whose terms are

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e \in t\,?\,e : e \qquad (18)$$

that is, the calculus in which sets of type-substitutions appear only in decorations. Consider the embedding $[\![.]\!]$ of our calculus (13) into this subcalculus, defined as $[\![e[\sigma_j]_{j\in J}]\!] = e@[\sigma_j]_{j\in J}$, as the identity for variables, and as its homomorphic propagation for all the other expressions. Then it is easy to prove the following theorem

Theorem 3.5. *For every well-typed expression e:*
1. $e \rightsquigarrow^* v \Rightarrow [\![e]\!] \rightsquigarrow^* [\![v]\!]$,
2. $[\![e]\!] \rightsquigarrow^* v \Rightarrow e \rightsquigarrow^* v'$ and $v = [\![v']\!]$

meaning that the subcalculus defined above is equivalent to the full calculus. Although expressions of the form $e[\sigma_j]_{j\in J}$ do not bring any further expressive power, they play a crucial role in local type inference, which is why we included them in our calculus. As we explain in details in the companion paper, for local type inference we need to reconstruct sets of type-substitutions that are applied to expressions but we *must not* reconstruct sets of type-substitutions that are decorations of λ-expressions. The reason is pragmatic and can be shown by considering the following two terms: $(\lambda^{\alpha\to\alpha} x.x)3$ and $(\lambda^{\alpha\to\alpha} x.4)3$. Every functional programmer will agree that the first expression must be considered well-typed while the second must not, for the simple reason that the constant function $(\lambda^{\alpha\to\alpha} x.4)$ does not have type $\alpha \to \alpha$. Indeed in the first case it is possible to apply a set of type-substitutions that makes the term well typed, namely $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]3$, while no such application exists for the second term. However, if we allowed reconstruction also for decorations, then the second term could be made well typed by adding the following decoration $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.4)3$.

4. Typing algorithm

The rules in Figure 1 do not describe a typing algorithm since they are not syntax directed. As customary the problem is the subsumption rule, and the way to go is to eliminate this rule by embedding appropriate checks of the subtyping relation into the rules that need it. This results in the system formed by the rules of Figure 2. This system is algorithmic (as stressed by $\vdash_{\mathscr{A}}$): in every case at most one rule applies, either because of the syntax of the term or because of mutually exclusive side conditions. Subsumption is no longer present and, instead, subtype checking has been pushed in all the remaining rules.

The rule for type-cases has been split in three rules (plus a fourth uninteresting rule we omitted that states that when $e : \mathbb{0}$ —*ie*, it is the provably diverging expression— then the whole type-case expression has type $\mathbb{0}$) according to whether one or both branches can be selected. Here the only modification is in the case where both branches can be selected: in the rule (*case*) in Figure1 the types of the two branches were subsumed to a common type s, while (ALG-CASE-BOTH) returns the least upper bound (*ie*, the union) of the two types.

The rule for abstractions underwent a minor modification with respect to the types returned for the body, which before were subsumed to the type declared in the interface while now the subtyping relation $s'_{ij} \leq s_i\sigma_j$ is explicitly checked.

The elimination of the subsumption yields a simplification in typing the application of type-substitutions, since in the system of Figure 1 without subsumption every premise of an (*inter*) rule is the consequence of an (*inst*) rule. The two rules can thus be merged into a single one, yielding the (ALG-INST) rule (see Appendix C.1 and in particular Theorem C.1).

As expected, the core of the typing algorithm is the rule for application. In the system of Figure 1, in order to apply the (*appl*) rule, the type of the function had to be subsumed to an arrow type, and the type of the argument had to be subsumed to the domain of that arrow type; then the co-domain of the arrow is taken to type the application. In the algorithmic rule (ALG-APPL), this is done by the type meta-operator "$\cdot$" which is formally defined as follows: $t \cdot s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\}$. In words, if t is the type of the function and s the type of the argument, this operator looks for the smallest arrow type larger than t and with domain s, and it

returns its co-domain. More precisely, when typing e_1e_2, the rule (ALG-APPL) checks that the type t of e_1 is a functional one (*ie*, $t \leq \mathbb{0}\to\mathbb{1}$). It also checks that the type s of e_2 is a subtype of the domain of t (denoted by $\mathsf{dom}(t)$). Because t is not necessarily an arrow type (in general, it is equivalent to a disjunctive normal form like the one of equation (12) in Section 3), the definition of the domain is not immediate. The domain of a function whose type is an intersection of arrows and negation of arrows is the union of the domains of all positive literals. For instance the domain of a function of type $(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$ is $\mathtt{Int} \vee \mathtt{Bool}$, since it can be equally applied to integer or Boolean arguments, while the domain of `even` as defined in (8) is $\mathtt{Int} \vee (\alpha\setminus\mathtt{Int})$, that is $\mathtt{Int} \vee \alpha$. The domain of a union of functional types is the intersection of each domain. For instance an expression of type $(s_1\to s_2) \vee (t_1\to t_2)$ will return either a function of type $s_1\to s_2$ or a function of type $t_1\to t_2$, so this expression can be applied only to arguments that fit both cases, that is, to arguments in $s_1 \wedge t_1$. Formally, if $t \leq \mathbb{0}\to\mathbb{1}$, then $t \simeq \bigvee_{i\in I}(\bigwedge_{p\in P_i}(s_p\to t_p) \wedge \bigwedge_{n\in N_i} \neg(s_n\to t_n) \wedge \bigwedge_{q\in Q_i}\alpha_q \wedge \bigwedge_{r\in R_i}\neg\beta_r)$ (with all the P_i's not empty), and therefore $\mathsf{dom}(t) \stackrel{\text{def}}{=} \bigwedge_{i\in I}\bigvee_{p\in P_i} s_p$ (here type variables do not count since they are intersected and universally quantified so the definition of the domain must hold also when their intersection is $\mathbb{1}$). Finally, the type returned in (ALG-APP) is $t \cdot s$, which we recall is the smallest result type that can be obtained by subsuming t to an arrow type compatible with s. We can prove that 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 (see Lemma C.12).

The algorithmic system is sound and complete with respect to the type system of Figure 1 and satisfies the minimum typing property (see Appendix C for the proofs).

Theorem 4.1 (Soundness). *If $\Delta\,;\Gamma \vdash_{\mathscr{A}} e : t$, then $\Delta\,;\Gamma \vdash e : t$.*

Theorem 4.2 (Completeness). *If $\Delta\,;\Gamma \vdash e : t$, then there exists a type s such that $\Delta\,;\Gamma \vdash_{\mathscr{A}} e : s$ and $s \leq t$.*

Corollary 4.3 (Minimum typing). *If $\Delta\,;\Gamma \vdash_{\mathscr{A}} e : t$, then $t = \min\{s \mid \Delta\,;\Gamma \vdash e : s\}$.*

Finally, it is quite easy to prove that type inference is decidable. It suffices to observe that the algorithmic rules strictly reduce the size of the expressions.

Theorem 4.4 (Termination). *Let e be an expression. Then the type inference algorithm for e terminates.*

This system constitutes a further theoretical contribution of our work since with this type system the language defined by grammar (13), the one by grammar (18), and *a fortiori* the one by grammar (17) are intersection type systems that all satisfy the Curry-Howard isomorphism since there is a one-to-one correspondence between terms and proofs of the algorithmic system.

5. Evaluation

In this section we define an efficient execution model for the polymorphic calculus as a conservative extension of the execution model of the monomorphic calculus: by "efficient" we mean that monomorphic expressions will be evaluated as efficiently as in the original ℂDuce runtime. In fact, even polymorphic expressions will be evaluated as efficiently as well (as if type variables were basic monomorphic types) despite the fact that the formal reduction semantics of polymorphic expressions includes a run-time relabeling operation. The key observation that allows us to define an efficient execution model for the polymorphic calculus is that relabeling can be implemented *lazily* so that the only case in which relabeling is computed at run-time will correspond to testing the type of a partial application of a polymorphic function. In practice, this case is so rare —at least in the XML setting— that there is no difference between monomorphic and polymorphic evaluation.

5.1 Monomorphic Language

Let us start by recalling the execution model of monomorphic ℂDuce, which is a classic closure-based evaluation. Expressions and values are defined as

$$e ::= c \mid x \mid \lambda^t x.e \mid ee \mid e \in s\,?\,e : e \qquad v ::= c \mid \langle \lambda^t x.e, \mathscr{E}\rangle$$

where t denotes an intersection of arrow types, s denotes a closed type, and $\mathscr{E}$ denotes an *environment*, that is, a substitution mapping expression variables into values. The big step semantics is:

$$\text{(ME-CONST)}\ \ \mathscr{E} \vdash_{\mathsf{m}} c \Downarrow c \qquad \text{(ME-VAR)}\ \ \mathscr{E} \vdash_{\mathsf{m}} x \Downarrow \mathscr{E}(x) \qquad \text{(ME-CLOSURE)}\ \ \mathscr{E} \vdash_{\mathsf{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathscr{E}\rangle$$

$$\text{(ME-APPLY)}\ \ \frac{\mathscr{E} \vdash_{\mathsf{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathscr{E}'\rangle \qquad \mathscr{E} \vdash_{\mathsf{m}} e_2 \Downarrow v_0 \qquad \mathscr{E}', x \mapsto v_0 \vdash_{\mathsf{m}} e \Downarrow v}{\mathscr{E} \vdash_{\mathsf{m}} e_1e_2 \Downarrow v}$$

$$\text{(ME-TYPE CASE T)}\ \ \frac{\mathscr{E} \vdash_{\mathsf{m}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathsf{m}} t \qquad \mathscr{E} \vdash_{\mathsf{m}} e_2 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{m}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\text{(ME-TYPE CASE F)}\ \ \frac{\mathscr{E} \vdash_{\mathsf{m}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathsf{m}} t \qquad \mathscr{E} \vdash_{\mathsf{m}} e_3 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{m}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

To complete the definition we define the relation $v \in_{\mathsf{m}} t$, that is, membership of a (monomorphic) value to a (monomorphic) type:

$$c \in_{\mathsf{m}} t \quad \stackrel{\text{def}}{\iff} \quad b_c \leq t$$
$$\langle \lambda^s x.e, \mathscr{E}\rangle \in_{\mathsf{m}} t \quad \stackrel{\text{def}}{\iff} \quad s \leq t$$

where $\leq$ is the subtyping relation of ℂDuce [11].

5.2 Polymorphic Language

In the naive extension of this semantics to the explicitly-typed polymorphic calculus of Section 3, we deal with type-substitutions as we do for environments, that is, by storing them in closures. This is reflected by the following definition where, for brevity, we write σ_I to denote the set of type-substitutions $[\sigma_i]_{i\in I}$:

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda^t_{\sigma_I} x.e \mid ee \mid e \in t\,?\,e : e \mid e\sigma_I \\ v & ::= & c \mid \langle \lambda^t_{\sigma_I} x.e, \mathscr{E}, \sigma_I\rangle \end{array}$$

The big-step semantics is then defined as follows, where each expression is evaluated with respect to an environment $\mathscr{E}$ determining the current value substitutions and a set of type-substitutions σ_I:

$$\text{(PE-CONST)}\ \ \sigma_I;\mathscr{E} \vdash_{\mathsf{p}} c \Downarrow c \qquad \text{(PE-VAR)}\ \ \sigma_I;\mathscr{E} \vdash_{\mathsf{p}} x \Downarrow \mathscr{E}(x)$$

$$\text{(PE-CLOSURE)}\ \ \sigma_I;\mathscr{E} \vdash_{\mathsf{p}} \lambda^t_{\sigma_J} x.e \Downarrow \langle \lambda^t_{\sigma_J} x.e, \mathscr{E}, \sigma_I\rangle \qquad \text{(PE-INSTANCE)}\ \ \frac{\sigma_I \circ \sigma_J;\mathscr{E} \vdash_{\mathsf{p}} e \Downarrow v}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e\sigma_J \Downarrow v}$$

$$\text{(PE-APPLY)}\ \ \frac{\begin{array}{c}\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow \langle \lambda^{\wedge_{l\in L} s_l\to t_l}_{\sigma_K} x.e, \mathscr{E}', \sigma_H\rangle \qquad \sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_2 \Downarrow v_0 \\ \sigma_J = \sigma_H \circ \sigma_K \quad P = \{j \in J \mid \exists l \in L : v_0 \in_{\mathsf{p}} s_l\sigma_j\} \\ \sigma_P;\mathscr{E}', x \mapsto v_0 \vdash_{\mathsf{p}} e \Downarrow v\end{array}}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1e_2 \Downarrow v}$$

$$\text{(PE-TYPE CASE T)}\ \ \frac{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathsf{p}} t \qquad \sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_2 \Downarrow v}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\text{(PE-TYPE CASE F)}\ \ \frac{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathsf{p}} t \qquad \sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_3 \Downarrow v}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

The membership relation $v \in_{\mathsf{p}} t$ for polymorphic values is inductively defined as:

$$c \in_{\mathsf{p}} t \quad \stackrel{\text{def}}{\iff} \quad b_c \leq t$$
$$\langle \lambda^s_{\sigma_J} x.e, \mathscr{E}, \sigma_I\rangle \in_{\mathsf{p}} t \quad \stackrel{\text{def}}{\iff} \quad s(\sigma_I \circ \sigma_J) \leq t$$

where $\leq$ is the subtyping relation of Castagna and Xu [5]. It is not difficult to show that this big-step semantics is equivalent to the small-step one of Section 3. Let $(\!|.|\!)$ be the transformation that maps values of the polymorphic language into corresponding values of the calculus, that is

$$(\!|c|\!) = c \qquad \text{and} \qquad (\!|\langle \lambda^s_{\sigma_J} x.e, \mathscr{E}, \sigma_I \rangle|\!) = \lambda^s_{\sigma_I \circ \sigma_J} x.(e(\!|\mathscr{E}|\!)) \quad (19)$$

where $(\!|\mathscr{E}|\!)$ applies $(\!|.|\!)$ to all the values in the range of $\mathscr{E}$. Let $\mathbb{i}$ denote the singleton set containing the empty type-substitution $[\{\,\}]$, which is the neutral element of the composition of sets of type-substitutions. Then we have:

Theorem 5.1. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then:* $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v \iff e \rightsquigarrow^* (\!|v|\!)$.

This implementation has a significant computational burden compared to that of the monomorphic language: first of all, each application of (PE-APPLY) computes the set P, which requires to implement several type-substitutions and membership tests; second, each application of (PE-INSTANCE) computes the composition of two sets of type-substitutions. In the next section we describe a different solution consisting in the compilation of the explicitly typed calculus into an intermediate language so that these computations are postponed as much as possible and are performed only if and when they are really necessary.

5.3 Intermediate Language

The intermediate language into which we compile the explicitly-typed polymorphic language is very similar to the monomorphic version. The only difference is that λ-abstractions (both in expressions and closures) may contain type variables in their interface and have an extra decoration Σ which is a term *denoting* a set of type-substitutions.

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda^t_\Sigma x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^t_\Sigma x.e, \mathscr{E} \rangle \\ \Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma) \end{array}$$

Intuitively, a $\mathtt{comp}(\Sigma, \Sigma')$ term corresponds to an application of the $\circ$ composition operator to the sets of type substitutions denoted by Σ and Σ', while a $\mathtt{sel}(x, t, \Sigma)$ term selects the subset of type substitutions σ denoted by Σ that are compatible with the fact that (the value instantiating) x belongs to the domain of $t\sigma$.

The big step semantics for this intermediate language is:

(OE-CONST) $\mathscr{E} \vdash_{\mathsf{o}} c \Downarrow c$ (OE-VAR) $\mathscr{E} \vdash_{\mathsf{o}} x \Downarrow \mathscr{E}(x)$ (OE-CLOSURE) $\mathscr{E} \vdash_{\mathsf{o}} \lambda^t_\Sigma x.e \Downarrow \langle \lambda^t_\Sigma x.e, \mathscr{E} \rangle$

(OE-APPLY)
$$\frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow \langle \lambda^t_\Sigma x.e, \mathscr{E}' \rangle \qquad \mathscr{E} \vdash_{\mathsf{o}} e_2 \Downarrow v_0 \qquad \mathscr{E}', x \mapsto v_0 \vdash_{\mathsf{o}} e \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} e_1 e_2 \Downarrow v}$$

(OE-TYPE CASE T)
$$\frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathsf{o}} t \qquad \mathscr{E} \vdash_{\mathsf{o}} e_2 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

(OE-TYPE CASE F)
$$\frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathsf{o}} t \qquad \mathscr{E} \vdash_{\mathsf{o}} e_3 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

Notice that this semantics is structurally the same as that of the monomorphic language. There are only two minor differences: (i) λ-abstractions have an extra decoration Σ (which has no impact on efficiency since it corresponds in the implementation to manipulate descriptors with an extra field) and (ii) the corresponding (_E-TYPE CASE) rules use a slightly different relation: $\in_{\mathsf{o}}$ instead of $\in_{\mathsf{m}}$. It is thus easy to see that in terms of steps of reduction the two semantics have the same complexity. What changes is the test of the membership relation ($\in_{\mathsf{o}}$ rather than $\in_{\mathsf{m}}$) since, when the value to be tested is a closure, we need to materialize relabelings. In other words, we have to evaluate the Σ expression decorating the function and apply the resulting set of substitutions to the interface of the function. Formally:

$$\begin{array}{rcl} c \in_{\mathsf{o}} t & \stackrel{\text{def}}{\iff} & b_c \leq t \\ \langle \lambda^s_\Sigma x.e, \mathscr{E} \rangle \in_{\mathsf{o}} t & \stackrel{\text{def}}{\iff} & s(\mathsf{eval}(\mathscr{E}, \Sigma)) \leq t \end{array}$$

where the evaluation of the symbolic set of type-substitutions is inductively defined as

$$\begin{array}{c} \mathsf{eval}(\mathscr{E}, \sigma_I) = \sigma_I \\ \mathsf{eval}(\mathscr{E}, \mathtt{comp}(\Sigma, \Sigma')) = \mathsf{eval}(\mathscr{E}, \Sigma) \circ \mathsf{eval}(\mathscr{E}, \Sigma') \\ \mathsf{eval}(\mathscr{E}, \mathtt{sel}(x, \bigwedge_{i \in I} t_i \to s_i, \Sigma)) = \\ [\ \sigma_j \in \mathsf{eval}(\mathscr{E}, \Sigma) \mid \exists i \in I : \mathscr{E}(x) \in_{\mathsf{o}} t_i \sigma_j\ ] \end{array}$$

Notice in the last rule the crucial role played by x and $\mathscr{E}$: by using an expression variable x in the symbolic representation of type-substitutions and relying on its interpretation through $\mathscr{E}$, we have transposed to type-substitutions the same benefits that closures bring to value substitutions: just as closures allow *value*-substitutions to be materialized only when a formal parameter is used rather than at the moment of the reduction, so our technique allows *type*-substitutions to be materialized only when a type variable is effectively tested, rather than at the moment of the reduction.

It is easy to see that the only case in which the computation of $\in_{\mathsf{o}}$ is more expensive than that of $\in_{\mathsf{m}}$ is when the value whose type is tested is a closure $\langle \lambda^t_\Sigma x.e, \mathscr{E} \rangle$ in which t is not closed and Σ is not $\mathbb{i}$.[5] The Σ decoration is different from $\mathbb{i}$ only if the closure is the result of a partial application of a curried function. The type t is not closed only if such partial application yielded a polymorphic function. In conclusion, the evaluation of an expression in the polymorphic language is more expensive than the evaluation of a similar[6] expression of the monomorphic language only if it tests the type of a polymorphic function resulting from the partial application of a polymorphic curried function. The additional overhead is limited only to this particular test and in all the other cases the evaluation is as efficient as in the monomorphic case. Finally, it is important to stress that this holds true also if we add product types: the test of a pair of values in the polymorphic case is as expensive as in the monomorphic case and so is the rest of the evaluation. Since in the XML setting the vast majority of the computation time is spent in testing products (since they encode sequences, trees, and XML elements), then the overhead brought by adding polymorphism —*ie*, the overhead due to testing the type of a polymorphic partial application of a polymorphic curried function— is negligible in practice.

All that remains to do is to define the compilation of the explicitly-typed language into the intermediate language:

$$\begin{array}{rcl} [\![x]\!]_\Sigma & = & x \\ [\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\mathtt{comp}(\Sigma, \sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x, t, \mathtt{comp}(\Sigma, \sigma_I))} \\ [\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\ [\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma, \sigma_I)} \\ [\![e_1 \in t\,?\,e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t\,?\,[\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma \end{array}$$

Given a closed program e we compile it in the intermediate language as $[\![e]\!]_{\mathbb{i}}$. In practice, the compilation will be even simpler since we apply it only to expressions generated by local type inference algorithm described in the companion paper where all λ's are decorated by $\mathbb{i}$ (*cf.* discussion at the end of Section 3.5). So the second case of the definition simplifies to: $[\![\lambda^t_{\mathbb{i}} x.e]\!]_\Sigma = \lambda^t_\Sigma x.[\![e]\!]_{\mathtt{sel}(x, t, \Sigma)}$.

The compilation is adequate:

[5] To be more precise, when there exists a substitution $\sigma \in \mathsf{eval}(\mathscr{E}, \Sigma)$ such that $\mathsf{var}(t) \cap \mathsf{dom}(\sigma) \neq \varnothing$. Notice that the tests of the subtyping relation for monomorphic and polymorphic types have the same complexity [5].

[6] By similar we intend with the same syntax tree but only closed types.

Theorem 5.2. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then* $\mathbb{i};\varnothing \vdash_{\mathsf{p}} e \Downarrow v \iff \vdash_{\mathsf{o}} [\![e]\!]_{\mathbb{i}} \Downarrow v'$ *, with* $(\!|v|\!) = (\!|v'|\!)$.

where $(\!|\langle\lambda^t_\Sigma x.e, \mathscr{E}\rangle|\!)$ evaluates all the symbolic expressions and type-substitutions in the term (see (19)). By combining Theorems 5.1 and 5.2 we obtain the adequacy of the compilation:

Corollary 5.3. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then* $\vdash_{\mathsf{o}} [\![e]\!]_{\mathbb{i}} \Downarrow v \iff e \rightsquigarrow^* (\!|v|\!)$.

5.4 Let-polymorphism

A function is polymorphic if it can be safely applied to arguments of different types. The calculus presented supports a varied palette of different forms of polymorphism: it uses subtype polymorphism (a function can be applied to arguments whose types are subtypes of its domain type), the combination of intersection types and typecase expressions yields "ad hoc" polymorphism (aka overloading), and finally the use of type variables in function interfaces provides parametric polymorphism. Polymorphism is interesting when used with bindings: instead of repeating the definition of a function every time we need to apply it, it is more convenient to define the function once, bind it to an expression variable, and use the variable every time we need to apply the function. In the current system it is possible to combine binding only with the first two kinds of polymorphism: different occurrences of a variable bound to a function can be given different types —thus, be applied to arguments of different types—, either by subsumption (*ie*, by assigning to the variable a super-type of the type of the function it denotes) or by intersection elimination (*ie*, by assigning to a variable one of the types that form the intersection type of the function it denotes). However, as it is well known in the languages of the ML-family, in the current setting it is not possible to combine binding and parametric polymorphism. Distinct occurrences of a variable cannot be given different types by instantiation (*ie*, by assigning to the variable a type which is an instance of the type of the function it denotes). In other terms, all λ-abstracted variables have monomorphic types (with respect to parametric polymorphism), which is why in ML auto-application $\lambda x.xx$ is not typeable.

The solution is well known and consists in introducing `let` bindings. This amounts to defining a new class of expression variables so that variables introduced by a `let` have polymorphic types, that is, types that have been generalized at the moment of the definition and can be instantiated in the body of the let. To sum up, λ-abstracted variables have monomorphic types, while `let`-bound variables (may) have polymorphic types and, thus, be given different types obtained by instantiation. For short we call the former λ-abstracted variables "monomorphic (expression) variables" and the latter `let`-bound variables "polymorphic (expression) variables".

In the explicitly-typed calculi of the previous sections we had just λ-abstracted variables. That these variables have monomorphic types is clearly witnessed by the fact that, operationally, $x\sigma_I$ is equivalent to (*ie*, reduces to) x. Clearly this property must not hold for polymorphic type variables since

$$\texttt{let}\ x = (\lambda^{\alpha\to\alpha} y.y)\ \texttt{in}\ (x[\{\alpha\to\alpha/\alpha\}])x \tag{20}$$

is, intuitively, well typed, while the same term obtained by replacing $x[\{\alpha \to \alpha/\alpha\}]$ with x is not (see the extension of the definition of relabeling for polymorphic variables later on).

To enable the definition of polymorphic functions to our calculus we add a `let` expression. To ease the presentation and to stress that the addition of `let`-bindings is a conservative extension of the previous system, we syntactically distinguish the current monomorphic variables (*ie*, those abstracted by a λ) from polymorphic variables by underlining the latter ones.

$$e ::= \cdots \mid \underline{x} \mid \texttt{let}\ \underline{x} = e\ \texttt{in}\ e$$

Reduction is as usual:

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

Relabeling is extended by the following definitions

$$\begin{array}{rcl} \underline{x}@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \underline{x}[\sigma_j]_{j\in J} \\ (\texttt{let}\ \underline{x}=e'\ \texttt{in}e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \texttt{let}\ \underline{x}=(e'@[\sigma_j]_{j\in J})\,\texttt{in}(e@[\sigma_j]_{j\in J}) \end{array}$$

and the (algorithmic) typing rule is as expected:

$$(\textit{let})\ \frac{\Delta\,;\Gamma \vdash_{(\mathscr{A})} e_1 : t_1 \qquad \Delta\,;\Gamma, (\underline{x} : t_1) \vdash_{(\mathscr{A})} e_2 : t_2}{\Delta\,;\Gamma \vdash_{(\mathscr{A})} \texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2 : t_2}$$

Type environments Γ now map also polymorphic expression variables into types. Notice that for a polymorphic expression variable $\underline{x}$ it is no longer true that $\mathsf{var}(\Gamma(\underline{x})) \subseteq \Delta$ (not adding $\mathsf{var}(\Gamma(\underline{x}))$ to Δ corresponds to generalizing the type of $\underline{x}$ before typing e_2 as in the GEN rule of the Damas-Milner algorithm: *cf.* [15]). As before we assume that polymorphic type variables of a `let`-expression (in particular those generalized for let-polymorphism) are distinct from monomorphic and polymorphic type variables of the context that the `let`-expression occurs in.

Likewise, environments now map both monomorphic and polymorphic expression variables into values so that the rules for evaluation in Section 5.2 are extended with the following ones:

$$(\text{PE-PVAR}_c)\ \frac{\mathscr{E}(\underline{x}) = c}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} \underline{x} \Downarrow c} \qquad (\text{PE-PVAR}_f)\ \frac{\mathscr{E}(\underline{x}) = \langle\lambda^t y.e, \mathscr{E}', \sigma_J\rangle}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} \underline{x} \Downarrow \langle\lambda^t y.e, \mathscr{E}', \sigma_I \circ \sigma_J\rangle}$$

$$(\text{PE-LET})\ \frac{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_0 \qquad \sigma_I;\mathscr{E}, \underline{x} \mapsto v_0 \vdash_{\mathsf{p}} e_2 \Downarrow v}{\sigma_I;\mathscr{E} \vdash_{\mathsf{p}} \texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2 \Downarrow v}$$

To compile `let`-expressions we have to extend the intermediate language likewise: we will distinguish polymorphic expression variables by decorating them with sets of type-substitution formulæ Σ that apply to *that particular occurrence* of the variable. So we add to the productions of Section 5.3 the following ones:

$$e ::= \cdots \mid x_\Sigma \mid \texttt{let}\ \underline{x} = e\ \texttt{in}\ e$$

while the big-step semantics of the new added expressions is

$$(\text{OE-PVAR}_c)\ \frac{\mathscr{E}(\underline{x}) = c}{\mathscr{E} \vdash_{\mathsf{o}} x_\Sigma \Downarrow c} \qquad (\text{OE-PVAR}_f)\ \frac{\mathscr{E}(\underline{x}) = \langle\lambda^t_{\Sigma'} y.e, \mathscr{E}'\rangle}{\mathscr{E} \vdash_{\mathsf{o}} x_\Sigma \Downarrow \langle\lambda^t_{\mathtt{comp}(\Sigma,\Sigma')} y.e, \mathscr{E}'\rangle}$$

$$(\text{OE-LET})\ \frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow v_0 \qquad \mathscr{E}, \underline{x} \mapsto v_0 \vdash_{\mathsf{o}} e_2 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} \texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2 \Downarrow v}$$

Notice how rule (OE-PVAR$_f$) uses the Σ decoration on the variable to construct the closure. The final step is the extension of the compiler for the newly added terms:

$$\begin{array}{rcl} [\![\underline{x}]\!]_\Sigma & = & x_\Sigma \\ [\![\texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2]\!]_\Sigma & = & \texttt{let}\ \underline{x} = [\![e_1]\!]_\Sigma\ \texttt{in}\ [\![e_2]\!]_\Sigma \end{array}$$

As an example, the `let`-expression (20) is compiled into

$$\texttt{let}\ \underline{x} = (\lambda^{\alpha\to\alpha} y.y)\ \texttt{in}\ x_{[\{\alpha\to\alpha/\alpha\}]\mathbb{i}}$$

where the substitution $[\{\alpha\to\alpha/\alpha\}]$ that is applied to the leftmost occurrence of x is recorded in the variable and will be used to instantiate the closure associated with x by the environment; the rightmost occurrence of x is decorated by $\mathbb{i}$ and therefore the value bound to it will not be instantiated. Theorems 5.1, 5.2, and Corollary 5.3 hold also for these extensions (see Appendix D for the proofs).

In an actual programming language there will not be any syntactic distinction between the two kinds of expression variables and compilation can be optimized by transforming variables that are `let`-bound to monomorphic values into monomorphic variables. So, whenever e_1 has a monomorphic type t_1, the `let`-expression should be compiled as

$$[\![\texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2]\!]_\Sigma \quad = \quad [\![(\lambda^{t_1\to t_2} x.e_2\{x/\underline{x}\})e_1]\!]_\Sigma$$

where t_2 is the type deduced for e_2 under the hypothesis that $\underline{x}$ has type t_1. Notice that this optimization is compositional.

6. Design choices and extensions

For the sake of concision we omitted two key features in the presentation: recursive functions and pairs. Recursive functions can be straightforwardly added with minor modifications. In particular, for recursive functions, whose syntax is $\mu f^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e$, it suffices to add in the type environment Γ the recursion variable f associated with the type obtained by applying the decoration to the interface, that is, $f : \wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$: the reader can refer to Section 7.5 in [11] for a discussion on how and why recursion is restricted to functions.

The extension with product types, instead, is less straightforward but can be mostly done by using existing techniques. Syntactically, we add pairs (e, e) and projections $\pi_i e$ (for $i{=}1, 2$) to terms and the product type constructor $t{\times}t$ to types. Reduction semantics is standard: two notions of reduction $\pi_i(v_1, v_2)\rightsquigarrow v_i$ (for $i{=}1, 2$) plus the usual context reduction rules. Typing rules are standard, as well: a pair is typed by the product of the types of its components and if e is of type $t_1{\times}t_2$, then its i-th projection $\pi_i e$ has type t_i. The rule for pairs in the algorithmic system $\vdash_{\mathscr{A}}$ is the same as in the static semantics, while the rules for projections $\pi_i e$ become more difficult because the type inferred for e may not be of the form $t_1{\times}t_2$ but, in general, is (equivalent to) a union of intersections of types. We already met the latter problem for application (where the function type may be different from an arrow) and there we checked that the type deduced for the function in an application is a functional type (*ie*, a subtype of $\mathbb{0}{\to}\mathbb{1}$). Similarly, for products we must check that the type of e is a product type (*ie*, a subtype of $\mathbb{1}{\times}\mathbb{1}$). If the constraint is satisfied, then it is possible to define the type of the projection (in a way akin to the definition of the domain $\mathsf{dom}()$ for function types) using standard techniques of semantic subtyping (see Section 6.11 in [11]). This is explained in details in Appendix C.

For the semantics of the calculus we made few choices that restrict its generality. One of these, the use of a call-by-value reduction, is directly inherited from $\mathbb{C}$Duce and it is required to ensure subject reduction. If e is an expression of type $\mathtt{Int}{\vee}\mathtt{Bool}$, then the application $(\lambda^{(\mathtt{Int}\to\mathtt{Int}\times\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}\times\mathtt{Bool})}x.(x, x))e$ has type $(\mathtt{Int}{\times}\mathtt{Int}){\wedge}(\mathtt{Bool}{\times}\mathtt{Bool})$. If we use call-by-name, then this redex reduces to (e, e) whose type $(\mathtt{Int}{\vee}\mathtt{Bool}{\times}\mathtt{Int}{\vee}\mathtt{Bool})$ is larger than the type of the redex. Although the use of call-by-name would not hinder the soundness of the type system (expressed in terms of progress) we preferred to ensure subject reduction since it greatly simplifies the theoretical development.

A second choice, to restrict type-cases to closed types, was made by practical considerations: using open types in a type-case would have been computationally prohibitive insofar as it demands to solve at run-time the problem whether for two given types s and t there exists a type-substitution σ such that $s\sigma \leq t\sigma$ (we study this problem, that we call the tallying problems, in the companion paper [4]). Our choice, instead, is compatible with the highly optimized (and provably optimal) pattern matching compilation technique of $\mathbb{C}$Duce. We leave for future work the study of type-cases on types with monomorphic variables (*ie*, those in Δ). This does not require dynamic type tallying resolution and would allow the programmer to test capabilities of arguments bound to polymorphic type variables.

7. Related work

We focus on work related to this specific part of our work, namely, existing explicitly-typed calculi with intersection types, and functional languages to process XML data. Comparison with work on local type inference and type reconstruction is done in the second part of this work presented in the companion paper [4].

To compare the differences between the existing explicitly-typed calculi for intersection type systems, we discuss how the term of our daffy identity $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$ is rendered.

In [18, 21], typing derivations are written as terms: different typed representatives of the same untyped term are joined together with an intersection $\wedge$. In such systems, the function in (4) relabeled with $[\{\mathtt{Int}/\alpha\}, \{\mathtt{Bool}/\alpha\}]$ is written $(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x) \wedge (\lambda^{\mathtt{Bool}\to\mathtt{Bool}}x.(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)x)$. Type checking verifies that both $\lambda^{\mathtt{Int}\to\mathtt{Int}}x.(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x$ and $\lambda^{\mathtt{Bool}\to\mathtt{Bool}}x.(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)x$ are well typed separately, which generates two very similar typing derivations. The proposal of [13] follows the same idea, except that a separation is kept between the computational and the logical contents of terms. A term consists in the association of a *marked term* and a *proof term*. The marked term is just an untyped term where term variables are marked with integers. The proof term encodes the structure of the typing derivation and relates marks to types. The aforementioned example is written in this system as $(\lambda x : 0.(\lambda y : 1.x)x)@((\lambda 0^{\mathtt{Int}}.(\lambda 1^{\mathtt{Int}}.0)0) \wedge (\lambda 0^{\mathtt{Bool}}.(\lambda 1^{\mathtt{Bool}}.0)0))$. In general, different occurrences of a same mark can be paired with different types in the proof term. In [3], terms are duplicated (as in [18, 21]), but the type checking of terms does not generate copies of almost identical proofs. The type checking derivation for the term $((\lambda^{\mathtt{Int}\to\mathtt{Int}}x.(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x)\|\lambda^{\mathtt{Bool}\to\mathtt{Bool}}x.(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)x)$ verifies in parallel that the two copies are well typed.

The duplication of terms and proofs makes the definition of beta reduction (and other transformations on terms) more difficult in the calculi presented so far, because it has to be performed in parallel on all the typed instances that correspond to the same untyped term. *Branching types* have been proposed in [22] to circumvent this issue. The idea is to represent different typing derivations for a same term into a compact piece of syntax. To this end, the branching type which corresponds to a given intersection type t records the *branching shape* (*ie*, the uses of the intersection introduction typing rule) of the typing derivation corresponding to t. For example, the type $(\mathtt{Int}{\to}\mathtt{Int}) \wedge (\mathtt{Bool}{\to}\mathtt{Bool})$ has only two branches, which is represented in [22] by the branching shape $\mathsf{join}\{i{=}*, j{=}*\}$. Our running example is then written as $\Lambda\mathsf{join}\{i{=}*, j{=}*\}.\lambda x^{\{i=\mathtt{Int}, j=\mathtt{Bool}\}}.(\lambda y^{\{i=\mathtt{Int}, j=\mathtt{Bool}\}}.x)x$. Note that the lambda term itself is not copied, and no duplication of proofs happens during type checking either: the branches labeled i and j are verified in parallel.

In [8], the authors propose an expressive refinement type system with intersection, union, but also (a form of) dependent types, making possible to define, *eg*, the type of integer lists of length n, written $[\mathtt{Int}]^n$. The variable n can be quantified over either universally or existentially (using respectively Π and Σ). Thanks to this it is possible to consider different instantiations of a dependent type and, thus to type our daffy function by different instances of $[\mathtt{Int}]^n$ (rather than with any type, as for $\mathtt{Int}$ and $\mathtt{Bool}$ in our example). Type checking requires type annotations to be decidable: to check that $\lambda x.(\lambda y.x)x$ has type $\Pi n.(([\mathtt{Int}]^{2n}{\to}[\mathtt{Int}]^{2n}){\wedge}([\mathtt{Int}]^{2n+1}{\to}[\mathtt{Int}]^{2n+1}))$, the subterm $\lambda y.x$ has to be annotated. This problem is similar to finding appropriate annotations for the daffy function (4) in our language. In [8], terms are annotated with a list of typings: for example, $\lambda y.x$ can be annotated with $A = (x : [\mathtt{Int}]^{2n} \vdash \mathbb{1}{\to}[\mathtt{Int}]^{2n}, x : [\mathtt{Int}]^{2n+1} \vdash \mathbb{1}{\to}[\mathtt{Int}]^{2n+1})$, which says that if $x : [\mathtt{Int}]^{2n}$, then $\lambda y.x$ has type $\mathbb{1}{\to}[\mathtt{Int}]^{2n}$ (and similarly if $x : [\mathtt{Int}]^{2n+1}$). The above annotation A is not sound because when checking that $\lambda x.(\lambda y.x : A)x$ has result type $\Pi n.(([\mathtt{Int}]^{2n}{\to}[\mathtt{Int}]^{2n}) \wedge ([\mathtt{Int}]^{2n+1}{\to}[\mathtt{Int}]^{2n+1}))$, one can see that the occurrences of n in A escape their scope: they should be bound by the quantifier Π in the result type. To fix this, typing environments in annotations are extended with universally quanti-

fied variables, that can be instantiated at type checking. For example, $\lambda y.x : (m : \texttt{Nat}, x : [\texttt{Int}]^m \vdash \mathbb{1} \to [\texttt{Int}]^m)$ means that $\lambda y.x$ has type $\mathbb{1} \to [\texttt{Int}]^m$, assuming $x : [\texttt{Int}]^m$, where m can be instantiated with any natural number. With this annotation, the daffy function can be checked against $\Pi n.(([\texttt{Int}]^{2n} \to [\texttt{Int}]^{2n}) \wedge ([\texttt{Int}]^{2n+1} \to [\texttt{Int}]^{2n+1}))$, by instantiating m with respectively $2n$ and $2n+1$. It is possible to find a similarity between the annotations of [8] (the lists of typings) and our annotations (*ie*, the combination of interface and decoration) although instantiation in the former is much more harnessed. There is however a fundamental difference between the two systems and it is that [8] does not include a type case. Because of that annotations need not to be propagated at run-time: in [8] they are just used statically to check soundness and then erased at run-time. Without type-cases we could do the same, but it is precisely the presence of type-cases that justifies our formalism.

For what concerns XML programming, let us cite polymorphic XDuce [12] and the work by Vouillon [20]. In both, pattern matching is designed so as not to break polymorphism, but both have to give up something: higher-order functions for [12] and intersection, negation, and local type inference in [20] (the type of function arguments must be explicitly given). Furthermore, Vouillon's work suffers from the original sin of starting from a subtyping relation that is given axiomatically by a deduction system. This makes the intuition underlying subtyping very difficult to grasp (at least, for us). Another route taken is the one of OCamlDuce [10], which juxtaposes OCaml and $\mathbb{C}$Duce's type systems in the same language, keeping them separated. This practical approach yields little theoretical problems but forces a value to be of one kind of type or another, preventing the programmer from writing polymorphic XML transformations. Lastly, XHaskell by Sulzmann *et al.* [19] mixes Haskell type classes with XDuce regular expression types but has two main drawbacks. First, every polymorphic variable must be annotated wherever it is instantiated with an XML type. Second, even without inference of explicit annotations (which they do not address), their system requires several restrictions to be decidable (while our system with explicit type-substitutions is decidable).

8. Conclusion

The work presented in this and in its companion paper [4] provides the theoretical basis and all the algorithmic tools needed to design and implement polymorphic functional languages for semi-structured data and, more generally, for functional languages with recursive types and set-theoretic unions, intersections, and negations. In particular, our results pave the way to the polymorphic extension of $\mathbb{C}$Duce [2] and to the definition of a real type system for XQuery 3.0 [9] (not just one in which all higher-order functions have type "`function()`"). Thanks to local type inference and type reconstruction defined in the second part of this work, these languages can have a syntax and semantics very close to those of OCaml or Haskell, but will include primitives (in particular complex patterns) to exploit the great expressive power of full-fledged set-theoretic types.

Some problems are still open, notably the decidability of type-substitution inference defined in the second part of this work, but these are of theoretical nature and should not have any impact in practice (as a matter of facts people program in Java and Scala even though the decidability of their type systems is still an open question). On the practical side, the most interesting directions of research is to couple the efficient compilation of the polymorphic calculus with techniques of static analysis that would perform partial evaluation of relabeling so as to improve the efficiency of type-case of functional values even in the rare cases in which it is more expensive than in the monomorphic version of $\mathbb{C}$Duce.

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

References

[1] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

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

[3] V. Bono, B. Venneri, and L. Bettini. A typed lambda calculus with intersection types. *Theor. Comput. Sci.*, 398(1-3):95–113, 2008.

[4] G. Castagna, K. Nguyễn, and Z. Xu. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. Unpublished manuscript, available at `http://hal.archives-ouvertes.fr/hal-00880744`, November 2013.

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

[6] J. Clark and M. Murata. Relax-NG, 2001. `www.relaxng.org`.

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

[8] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *POPL '04*. ACM Press, 2004.

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

[10] A. Frisch. OCaml + XDuce. In *ICFP '06*, 2006.

[11] 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.

[12] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. *ACM TOPLAS*, 32(1):1–56, 2009.

[13] L. Liquori and S. Ronchi Della Rocca. Intersection-types à la Church. *Inf. Comput.*, 205(9):1371–1386, 2007.

[14] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[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] J.C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[17] J.C. Reynolds. What do types mean?: from intrinsic to extrinsic semantics. In *Programming methodology*. Springer, 2003.

[18] S. Ronchi Della Rocca. Intersection typed lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 70(1):163–181, 2002.

[19] M. Sulzmann, K. Zhuo, and M. Lu. XHaskell - Adding Regular Expression Types to Haskell. In *IFL*, LNCS n. 5083. Springer, 2007.

[20] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL '06*, pages 103–114, 2006.

[21] J.B. Wells, A. Dimock, R. Muller, and F.A. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.*, 12(3):183–227, 2002.

[22] J.B. Wells and C. Haack. Branching types. In *ESOP '02*, volume 2305 of *LNCS*, pages 115–132. Springer, 2002.

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