Polymorphic Functions with Set-Theoretic Types

Giuseppe Castagna[1] Kim Nguyễn[2] Zhiwu Xu[3,1] Serguei Lenglet[1]

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

Abstract. We present a type system and local type inference for a calculus with higher-order polymorphic functions, recursive types with arrow and product type constructors and set-theoretic type connectives (union, intersection, and negation). This work provides the theoretical foundations and technical machinery needed to start the design and implementation of polymorphic functional languages for semi-structured data.

1. Introduction

The extensible markup language XML is a current standard format for exchanging structured data, which has been applied to web services, database, research on formal methods, and so on. Many recent XML processing languages 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 [4]) and higher-order functions and polymorphism have been recently proposed in the W3C draft for XQuery 3.0 [7]. Despite all this interest, spurs, and motivations a comprehensive polymorphic type system for XML was 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 [3], 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 that we develop 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 "$\wedge$" and type difference "$\setminus$"); and the pattern in the `case` expression is a type, meaning that it matches if the value returned by the matched expression has 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.

The goal of this work is to define a calculus and a type system such that it can pass three tests. The first test is that it can define the two functions above. The second, harder, test is that the type system must be able to verify that these functions have the types declared in their signatures. That `map` has the declared type will come as no surprise (in practice, we actually want the system to infer this type even in the absence of a signature given by the programmer: see Section 7). That `even` was given an intersection type means that it must have all the types of 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 integers, it returns a result of the same type. In other terms, `even` is a polymorphic (dynamically bounded) overloaded function. The third task, the really tough 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[1] to the following one:

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

since `map even` returns a function that when applied to a list of integers it returns a list of Booleans, when applied to a list that does not contain any integer then it returns a list of the same type (actually, it returns 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 lacks a satisfactory definition. Second, even if working with an explicitly typed setting may seem simpler, the system needs to solve "local type inference", 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. 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 reminder that unrestricted intersection type systems are undecidable should further strengthen it.

[1] The type that follows is redundant since the first type of the intersection is an instance (*eg*, for $\alpha =$ `Int`) of the third one. We included it for the sake of the presentation.

In the next section we outline in detail the various problems we met and how they were solved.

2. Overview

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

The essence of $\mathbb{C}$Duce is a λ-calculus with explicitly-typed recursive functions, pairs and a type-case expression. Its types, which can be recursively defined, include the arrow and product type constructors and the intersection, union, and negation type connectives.[2] In summary, they are the regular trees coinductively generated by the following productions:

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

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.

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 capture primitives for XML data). We initially focus on the functional core and disregard products, recursive functions, and constants since the results presented here can be easily extended to them (see Section 6), though we will freely use them for our examples. So we initially consider the following "Core$\mathbb{C}$Duce" terms:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e\in t\,?\,e:e \qquad (2)$$

where $e\in t\,?\,e_1 : e_2$ 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.

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 (2), namely, why a type-case expression is included and why we explicitly type whole λ-abstractions (with an intersection of arrow types) rather than just their parameters.

The reasons for a type-case are detailed in [9]: in short, intersection types are used to type overloaded functions, and without a type-case only "coherent overloading" *à la* Forsythe [12] can be defined (which, for instance, precludes the definition of a non diverging function of type $(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})$). Furthermore, 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 *strict* for all types s_1, s_2, t_1, t_2, 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.

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. As a matter of fact, if we provide just the type of the parameter x (not used in the body), then there is no type (apart from the useless $\mathbb{0}$ type) that makes the function typeable.

So, we need to define an explicitly typed language with intersection types. This is a notoriously hard problem for which no full-fledged solution exists, yet: intersection types systems with explicitly typed terms can be counted on the fingers of one hand, and none of them is completely satisfactory (see the section about related work in the full version). To give an idea of how difficult this is, 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` it returns an `Int`. For the sake of presentation imagine that we are happy to deduce for the function above the 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 what 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 make any distinction between arguments of type `Int` and those of type `Bool`.

The solution, which was introduced by $\mathbb{C}$Duce, 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 (2), $\lambda^{\wedge_{i\in I} s_i\to t_i}x.e$. 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.

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 (1) and, thus, in the types labeling λ-abstractions. It becomes thus possible to define the polymorphic identity function as $\lambda^{\alpha\to\alpha}x.x$, while classic "auto-application" term is written as $\lambda^{((\alpha\to\beta)\wedge\alpha)\to\beta}x.xx$. The intended meaning of using a type variable, such as α, is that a (well-typed) λ-abstraction not only has the type specified in its label (and by subsumption all its super-types) but also all the types obtained by instantiating the type variables occurring in the label. So $\lambda^{\alpha\to\alpha}x.x$ has by subsumption the types $\mathbb{0}\to\mathbb{1}$ (the type of all functions, which is a super-type of $\alpha\to\alpha$) and $\neg\texttt{Int}$, 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. In the monomorphic case a term can have distinct types only by subsumption and, thus, intersection types are assigned transparently to terms by the type system 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 subtype nor super-type of $\texttt{Int}\to\texttt{Int}$).

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

[2] We use the standard convention that connectives have a priority higher than constructors; the latter have priority higher than prefix connectives.

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) 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. 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 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 [13].

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:[3]

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x) \tag{3}$$

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 (3) to 3 reduces to $(\lambda^{\alpha\to\alpha}y.3)3$ which is not well-typed (it is not possible to deduce the type $\alpha\to\alpha$ for $\lambda^{\alpha\to\alpha}y.3$, which is the constant function that always returns 3) although it is the reductum of a well-typed application.

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 $(\lambda^{\texttt{Int}\to\texttt{Int}}y.3)3$ which is well typed. In general, however, the way to perform a relabeling is not so straightforward and clearly defined, since two different problems may arise: (1) it may be necessary to apply more than a single type-substitution and (2) 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 [5]). We discuss each problem in detail.

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 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 set to t (*eg*, in the example $t\{\texttt{Int}/\alpha\}\wedge t\{\texttt{Bool}/\alpha\}$). Thus the first problem has an easy solution.

The second problem is much harder and concerns the relabeling of the body of a function since the naive solution consisting of propagating the application of (sets of) substitutions to the bodies of functions fails in general. This can be seen by considering the relabeling via the set of substitutions $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$ of the daffy function in (3). If we apply the naive solution this yields

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x) \tag{4}$$

which is not well typed. That this term is not well typed is clear if we try apply it to, say, 3: the application of a function of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ to an `Int` should have type `Int`, but here it reduces to $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.3)3$, and there is no way to deduce the intersection type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ for the constant function $\lambda y.3$. But we can also directly verify that it is not well typed, by trying to type the function in (4). This corresponds to prove that under the hypothesis $x:\texttt{Int}$ the term $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$ has type `Int`, and that under the hypothesis $x:\texttt{Bool}$ this same term has type `Bool`. Both checks fail because, in both cases, $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is ill-typed (it neither has type $\texttt{Int}\to\texttt{Int}$ when x:`Bool`, nor has it type $\texttt{Bool}\to\texttt{Bool}$ when x:`Int`). This example shows that in order ensure that relabeling yields well-typed terms, the relabeling of the body *must change* according to the type 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`. 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 [2, 10, 14, 15] rely on the duplication of lambda terms and/or typing derivations, while other calculi such as [16] that aim at avoiding such duplication obtain it by adding new expressions and new syntax for types (see related work section in the full version).

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 (3) 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:

$$(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x). \tag{5}$$

The annotation in the outer "λ" indicates that the function must be relabeled and, therefore, that we are using the particular instance whose type is the one in the interface in 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, contrary to existing solutions, we preserve the structure of λ-terms (at the expenses of some extra annotation) which is of uttermost importance in a language-oriented study.

From a practical point of view it is important to stress that these annotations will be transparent to the programmer and that all necessary relabeling will be inferred statically and compiled away. In practice, the programmer will program in the language defined by the grammar (2), where types may contain type variables in the types of λ's. Note that the language defined by the grammar (2) passes our first test since the `even` function can be defined as

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2)=0:x \tag{6}$$

(where $s\setminus t$ is syntactic sugar for $s\wedge\neg t$) while —with the products and recursive function definitions outlined in Section 6— `map` is

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

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

[3] By convention a type variable is introduced by the outer 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 (3) 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 and of its argument are considered distinct and, thus, can be α-converted separately, as $(\lambda^{\gamma\to\gamma}x.x)(\lambda^{\delta\to\delta}x.x)$.

When fed by any expression this language, the type-checker 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`, we will see that the inference system of Section 4 infers one set of type-substitutions $[\{\texttt{Int}/\alpha, \texttt{Bool}/\beta\}, \{\alpha\vee\texttt{Int}/\alpha, (\alpha\setminus\texttt{Int})\vee\texttt{Bool}/\beta\}]$ and inserts it between the two terms. Finally, the compiler will compile the expression with the inserted type-substitutions into a monomorphic expression in which all substitutions are compiled away and only necessary relabelings are hard-coded. The rest of the presentation proceeds then in four logical phases:

1. Definition of a calculus with explicit (sets of) type-substitutions. These explicit type-substitutions are used at reduction time to perform the relabeling of the bodies of the applied function. We define a type systems and prove its soundness.

2. Inference of type-substitutions. We want to program with the terms defined by the grammar (2), and not in a calculus with explicit type-substitutions. Therefore we must define a system that infers where and whether type-substitutions can be inserted in a term to make it well typed. This can be reduced to the problem of deciding whether for two types s and t there exist two sets of substitutions $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that $s[\sigma_i]_{i\in I} \leq t[\sigma'_j]_{j\in J}$. We show that when the cardinalities of I and J are bounded the problem above reduces to finding all substitutions σ such that $t_1\sigma \leq t_2\sigma$ for two given types. We show how to produce a sound and complete set of solutions for the latter problem which immediately yields a semi-decision procedure (that tries all the cardinalities) for the inference system.

3. Compilation into a monomorphic calculus. We show how to compile this explicitly typed term into a monomorphic term of Core$\mathbb{C}$Duce, thus getting rid of explicit substitutions.

4. Extensions. Finally, we discuss additional features, such as product types and recursive functions that we omitted from the main presentation, as well as finer aspects of our system.

In the next sections we discuss in depth each of these phases, focusing on the intuition and trying to avoid as many technical details as possible. We dot the i's and cross the t's in the full version where all formal definitions and proofs of properties can be found (***n.b.*:** references in the text starting by capital letters —*eg*, Definition A.7— refer to it). But before we summarize the main contributions of our work.

Contributions: The overall contribution is the definition of a statically typed calculus with polymorphic higher-order functions for a type system with recursive types and union, intersection, and negation type connectives, as well as, its compilation into a monomorphic version. In order to obtain it we had to solve an array of problems each of which is, by itself, a stand-alone contribution of this work, namely:

- we defined an *explicitly-typed* λ-calculus with intersection (and union and negation) types, whose key idea consists in decorating terms with types *and type-substitutions* that are lazily propagated at the moment of the reduction. This contrasts with current solutions in the literature which require the addition of new operators, stores and/or pointers. In doing that we singled out that the problem of defining an explicit typed version of intersection type systems resides in the fact that the relabeling of the body of a function is dependent on the actual type of the argument, a point that, in our knowledge, was not understood before.
- we defined an algorithm that for any pair of types t_1 and t_2 produces a sound and complete set of solutions to the problem whether there exists a substitution σ such that $t_1\sigma \leq t_2\sigma$. This is obtained by using the set-theoretic interpretation of types to reduce the problem to a unification problem on regular tree types.
- we defined an algorithm for local type inference for the calculus. Practically speaking this means that the programmer has to explicitly type function definitions, but that any other type information, in particular the instantiations and expansion of type variables is inferred by the system at compile time. The algorithm yields a semi-decision procedure for the typeability of a λ-calculus with intersection types and with explicitly typed lambda expressions whose decidability is still an open issue.
- we defined a compilation of the polymorphic calculus into the monomorphic one. This is a non-trivial problem since the source polymorphic calculus includes a type-case expression. From a practical viewpoint it allows us to reuse the run-time engine developed for monomorphic $\mathbb{C}$Duce also for the polymorphic version with the sole modification of plugging the polymorphic subtyping relation in the execution of type-cases.

3. A calculus with explicit type-substitutions

The types of the calculus are those in the grammar (1) 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

$$t ::= \alpha \mid b \mid t \to t \mid t \wedge t \mid t \vee t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \tag{7}$$

and such that every infinite branch contains infinitely many occurrences of type constructors. The last condition ensures that unions, intersections, and negations are always finite, thus it rules out meaningless types such as $t = t \vee t$ or $t = \neg t$. We use $\mathcal{T}$ to denote the set of all types and $\mathsf{var}(t)$ to denote the set of all variables occurring in type t.

The subtyping relation for these types is the one defined by Castagna and Xu [3]. For this work it suffices to consider that monomorphic (*ie*, closed or, equivalently, ground) types are interpreted as sets of *values* (***n.b.*:** just values, not expressions) that have that type: in particular $s\to t$ contains all λ-abstractions that if applied to a value of type s and return a result, then it 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 their arguments). Type connectives (union, intersection, negation) are interpreted as the corresponding set-theoretic operators and subtyping is set containment. 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 $\simeq$). For what concerns polymorphic types 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: *cf.* [3]). Another 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

$$\bigwedge_{p\in P} b_p \wedge \bigwedge_{n\in N} \neg b_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \tag{8}$$

or

$$\bigwedge_{p\in P} (s_p\to t_p) \wedge \bigwedge_{n\in N} \neg(s_n\to t_n) \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \tag{9}$$

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

Terms are derived from those of Core$\mathbb{C}$Duce 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

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J} \tag{10}$$

(*subsumption*)
$$\frac{\Delta \wr \Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Delta \wr \Gamma \vdash e : t_2}$$

(*appl*)
$$\frac{\Delta \wr \Gamma \vdash e_1 : t_1 \to t_2 \quad \Delta \wr \Gamma \vdash e_2 : t_1}{\Delta \wr \Gamma \vdash e_1 e_2 : t_2}$$

(*abstr*)
$$\frac{\Delta \cup \mathsf{var}(\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j \quad i \in I \; j \in J}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}$$

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

(*inst*)
$$\frac{\Delta \wr \Gamma \vdash e : t \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e[\sigma] : t\sigma}$$

(*inter*)
$$\frac{\forall j \in J.\, \Delta \wr \Gamma \vdash e[\sigma_j] : t_j}{\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j} \; |J| > 1$$

Figure 1. Static semantics

and with the restriction that the type t occurring 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.

In order to define both static and dynamic semantics of the calculus 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 (lazily) applies $[\sigma_j]_{j\in J}$ to all outermost λ-abstractions occurring in e. Precisely, $e@[\sigma_j]_{j\in J}$ is defined for λ-abstractions as:

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

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

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

$$e[\sigma_j]_{j\in J} \quad \rightsquigarrow \quad e@[\sigma_j]_{j\in J} \tag{11}$$

$$(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e)v \quad \rightsquigarrow \quad (e@[\sigma_j]_{j\in P})\{v/x\} \tag{12}$$

$$v \in t \,?\, e_1 : e_2 \quad \rightsquigarrow \quad \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases} \tag{13}$$

where $P = \{j \in J \mid \exists i \in I, \vdash v : t_i\sigma_j\}$. The first rule performs relabeling, that is, it propagates the sets of type substitutions down into the decorations of the outermost λ-abstractions. The second rule 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). For instance, take the daffy identity function, instantiate it as in (5) 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 $(\lambda^{\texttt{Int}\to\texttt{Int}} y.\texttt{42})\texttt{42}$) having the reduction discarded the $\{\texttt{Bool}/\alpha\}$ substitution. Finally, the third rule checks whether the value returned by the expression in the type-case matches the specified type and selects the branch accordingly.

As expected in a calculus with a type-case expression, the dynamic semantics depends on the static semantics —precisely, on the typing of values— which is defined in Figure 1 (we just omitted the rule for variables, which is as customary). The judgments are of the form $\Delta \wr \Gamma \vdash e : t$, 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 interface 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 of [3]. 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 in the decoration to each input type of the interface and checks the type of the body relabeled with the substitution at issue (notice that all these checks are performed under the same updated Δ). For example, in the case of the daffy identity function instance in (5), 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 relabeling of the body in the premises is a key mechanism of the type system: if we had used $e[\sigma_j]$ instead of $e@[\sigma_j]$ in the premises of the (*abstr*) rule, then the term (5) 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 former but not to 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 $(\lambda^{\alpha\to\alpha} y.x)$ has not type $\alpha\to\alpha$. 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. Notice that the type system composes by intersection only different types of a same term obtained by instantiation. This is not restrictive since different types obtained by subsumption can be composed by intersection by applying just subsumption (see Lemma B.2). 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 since it checks the type of a branch only if there exists an instantiation of the type of e that makes the branch selectable. For instance, in $\lambda^{\alpha\to\cdots} x.\ x \in \texttt{Int} \,?\, e_1 : e_2$ we must check the type of e_1 since if the type α of x is instantiated with `Int`, then e_1 is selected.

As a final remark notice that explicit type-substitutions are only needed to type applications of polymorphic functions. Since the definitions of `map` and `even` do not apply in their bodies any polymorphic function (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 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 the most interesting case, that is, `even` (though the typing of the type-case in `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 (6) 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 [9] for a more detailed discussion on this point. This calculus satisfies the properties of subject reduction and progress.

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 $e \rightsquigarrow e'$.*

Finally, consider the sub-calculus in which type-substitutions occur only in decorations, that is, formed only by the first four productions in (10), and notice that it is closed with respect to the beta (12) and type case (13) reductions. Therefore this calculus (even without type-cases, in which case union and negation types are not needed) constitutes an explicitly-typed λ-calculus with intersection types whose expressive power subsumes those of intersection types (without an universal element, of course):

Theorem 3.3. *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 $\lceil e \rceil = m$.

4. Inference of type-substitutions

We want the programmer to write a program in the syntax defined in (2), that is, a term in which no explicit type-substitution occurs. It is then the task of the type-substitution inference system to check whether it is possible to insert some type-substitutions in some appropriate places of the term so that the resulting term type-checks with respect to the system of Figure 1. To define the type-substitution inference system we proceed in three steps:

1. In the first step we define a typing algorithm for the system in Figure 1, by giving an equivalent set of rules that is syntax directed. This essentially amounts to eliminate the subsumption rule from the the system of Figure 1.
2. The second step consists in defining a deduction system that checks whether and where it is possible to insert sets of type-substitutions into a term produced by the grammar (2) to make it well typed. There will be a single exception: we will not try to insert type-substitutions into decorations, since we suppose that all λ-abstractions initially have empty decorations. The reasons for this is that we want to infer that a term such as $\lambda^{\alpha\to\alpha}x.3$ is ill-typed and if we allowed to infer decorations, then the term could be typed by inserting a decoration as in $\lambda^{\alpha\to\alpha}_{\{\texttt{Int}/\alpha\}}x.3$.
   The set of places where the insertion of sets of type-substitutions must be tried are precisely given by the algorithm defined in the first step (they correspond to the places where a subtyping relation is checked), and this is used to define a deduction system that infers the type of the "implicitly-typed" (*ie*, without explicit type-substitutions) terms: if a type is deduced for some term, then there exists an explicitly-typed version of the term that has that same type, and vice-versa.
3. The deduction system given in the previous step is syntax directed but it does not yield an algorithm, yet, because it uses some operations that are not effective. In particular, these operations require to solve the problem of whether there exist two sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $s[\sigma_i]_{i\in I} \leq t[\sigma_j]_{j\in J}$. If the cardinalities of I and J are known, then this problem can be reduced to the *tallying problem*: given two types s and t we say that s *tallies with* t if there exists a type-substitution σ such as $s\sigma \leq t\sigma$. We show how to decide the tallying problem and devise a semi-decision procedure for the more general problem with sets of type-substitutions which essentially tries all possible cardinalities of the two sets. We conjecture decidability also for this second problem though we are not able to prove it, yet.

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

4.1 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 (plus the standard rule for variables). This system is algorithmic (as stressed by $\vdash_{\mathcal{A}}$): 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. As expected, the core of the typing algorithm is the rule for

[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 exhaustivity of pattern matching. We omitted this check since in the formal development of a calculus it is not necessary.

$$\text{(ALG-INST)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathcal{A}} e : t}{\Delta \wr \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\,\sigma_j \sharp\, \Delta$$

$$\text{(ALG-APPL)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Delta \wr \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Delta \wr \Gamma \vdash_{\mathcal{A}} e_1 e_2 : t \cdot s}\ \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathsf{dom}(t) \end{array}$$

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

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

$$\text{(ALG-CASE-SND)}\quad \frac{\Delta \wr \Gamma \vdash_{\mathcal{A}} e : t' \qquad \Delta \wr \Gamma \vdash_{\mathcal{A}} e_2 : s_2}{\Delta \wr \Gamma \vdash_{\mathcal{A}} (e \in t\,?\, e_1 : e_2) : s_2}\, t' \leq \neg t$$

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

Figure 2. Typing algorithm

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. The algorithmic rule (ALG-APPL) checks that the type the algorithm returns for the function is a functional type (*ie*, $t \leq \mathbb{0}\to\mathbb{1}$). It also checks that the type of the argument is a subtype of the domain (denoted by $\mathsf{dom}(t)$) of the function type. Since in general the latter does not have the form of an arrow type, the definition of 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 $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ is $\texttt{Int} \vee \texttt{Bool}$, since it can be indifferently applied to integer or Boolean arguments, while the domain of `even` as defined in (6) is $\texttt{Int}\vee(\alpha\setminus\texttt{Int})$, that is $\texttt{Int}\vee\alpha$. 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 is defined as the smallest result type that can be obtained by subsuming t to an arrow function compatible with s. Formally, $t\cdot s \stackrel{\text{def}}{=} \min\{u \mid t \leq s\to u\}$. 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:

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

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

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

Subject reduction and progress are direct consequences of these theorems and of the corresponding properties stated in Section 3.

4.2 Type substitution assignment

The next goal is to define a type-substitution assignment system for the calculus without explicit type-substitutions, that is, the one defined by the grammar in (2) that for convenience and clarity we reproduce here with a different the meta-variable.

$$a ::= x \mid aa \mid \lambda^{\wedge_{i\in I} s_i\to t_i} x.a \mid a\in t\,?\,a : a \qquad (14)$$

We call this calculus and its terms the *implicitly-typed* calculus and terms, respectively, in order to differentiate it from the one of Section 3 which has explicit type-substitutions and, therefore, will be called the *explicitly-typed* calculus. In this section we define an inference system for the implicitly-typed calculus that is sound and complete with respect to explicitly-typed one modulo the single restriction we already mentioned, namely we will consider only terms in the explicitly-typed calculus in which all decorations are the empty set of substitutions.

We have to define a system that guesses where sets of type-substitution must be inserted to make an implicitly-typed term well-typed in the system of Figure 2 (equivalently, of Figure 1). The general role of type-substitutions is to make the type of some term satisfy some subtyping constraints. Examples of this are the type of the body of a function which must match the result type declared in the interface, or the type of the argument of a function which must be a subtype of the domain of the function. Actually *all* the cases in which subtyping constraints must be satisfied are enumerated in Figure 2: they coincide with the subtyping relation checks occurring in the rules. Figure 2 is our Ariadne's thread through the definition of the type-substitution inference system: the rule (ALG-INT) must be removed and wherever the type algorithm in Figure 2 checks whether for some types s and t the relation $s \leq t$ holds, then the type inference system must check whether there exists a set of substitutions $[\sigma_i]_{i\in I}$ for the polymorphic variables (those not in Δ) that makes $s[\sigma_i]_{i\in I} \leq t$ hold.

The essence of the type inference system is all there. Of course things get a little more complicated in the rule for application since the algorithm must find two sets of substitutions (one for the function and another for the argument) and the minimum of a set. In order to ease the presentation it is quite handy to introduce a family of preorders $\sqsubseteq_\Delta$ that merge subtyping and instantiation:

Definition 4.4. *Let s and t be two types, Δ a set of type variables, and $[\sigma_i]_{i\in I}$ a set of type-substitutions. We define:*

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

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

Intuitively, it suffices to replace $\leq$ by $\sqsubseteq_\Delta$ (with the Δ used in the premises) in the algorithmic rules of Figure 2 to obtain the corresponding rules of type-substitution inference. This yields the three rules for the type-case (omitted: just replace $\leq$ and $\not\leq$ by $\sqsubseteq_\Delta$ and $\not\sqsubseteq_\Delta$ in the corresponding rules of Figure 2) as well as the rule (INF-ABST) of Figure 3 which has become simpler since we are working under the hypothesis that λ-abstractions have empty decorations, and which uses the $\Delta\cup\Delta'$ set to compare the types of the body with the result types specified in the interface ($s'_i \sqsubseteq_{\Delta\cup\Delta'} s_i$). Notice that we do not require the sets of type-substitutions that make $s'_i \sqsubseteq_{\Delta\cup\Delta'} s_i$ satisfied to be the same for all s'_i: this is not a problem since the case of different sets of type-substitutions

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

(INF-APPL)
$$\frac{\Delta \wr \Gamma \vdash_{\mathcal{I}} a_1 : t \qquad \Delta \wr \Gamma \vdash_{\mathcal{I}} a_2 : s}{\Delta \wr \Gamma \vdash_{\mathcal{I}} a_1 a_2 : u} \quad u \in (t \bullet_\Delta s)$$

Figure 3. Inference system for type-substitutions

corresponds to using their union as sets of type-substitutions (*ie*, to intersecting them point-wise: see Definition D.9).

It still remains the most delicate rule, the one for application. This is difficult because not only it must find two distinct sets of type-substitutions (one for the function type the other for the argument type) but also because the set of type-substitutions for the function type must enforce two distinct constraints: that the type is smaller than $\mathbb{0}\to\mathbb{1}$, and that its domain is compatible with the type of the argument. In order to solve all these constraints we collapse them into a unique definition which is the algorithmic counterpart of the set of types used in the Section 4.1 to define the operation $t\cdot s$ of rule (ALG-APPL). Precisely we define $t\bullet_\Delta s$ as the set of types for which there exist two sets of type-substitutions (for variables not in Δ) that make s compatible with the domain of t:

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

This set is upward closed and closed by intersection. For the application of a function of type t to an argument of type s the inference system deduces every type in $t \bullet_\Delta s$. This yields the inference rule (INF-APPL) of Figure 3.

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

Theorem 4.5 (Soundness of inference)**.** *If $\Delta \wr \Gamma \vdash_{\mathcal{I}} a : t$, then there exists a term e such that erase$(e) = a$ and $\Delta \wr \Gamma \vdash_{\mathcal{A}} e : t$.*

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

Theorem 4.6 (Completeness of inference)**.** *Let e be an (explicitly typed) term in which all decorations are empty. If $\Delta \wr \Gamma \vdash_{\mathcal{A}} e : t$, then there exists a type t' such that $\Delta \wr \Gamma \vdash_{\mathcal{I}}$ erase$(e) : t'$ and $t' \sqsubseteq_\Delta t$.*

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

4.3 Type tallying

Definition 4.7 (Tallying problem)**.** *Let C be a* constraint-set*, that is, a finite set of pairs of types (these pairs are called* constraints*), and Δ a finite set of type variables. A type-substitution σ is a solution for the* tallying problem of C and Δ *(noted $\sigma \Vdash_\Delta C$) if $\sigma\sharp\Delta$ and for all $(s,t) \in C$, $s\sigma \le t\sigma$ holds.*

It is clear that the "implementation" of most of the rules of the type-substitution inference system in Figure 3 corresponds to solving a particular tallying problem. The notable exception is the (INF-APPLY) rule since it requires to solve a more difficult problem. A "solution" for the (INF-APPLY) rule problem is *a pair* of sets of type-substitutions $[\sigma_i]_{i\in I}$, $[\sigma_j]_{j\in J}$ for variables not in Δ such that both $\bigwedge_{i\in I} t\sigma_i \le \mathbb{0}\to\mathbb{1}$ and $\bigwedge_{j\in J} s\sigma_j \le \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i)$ hold. In this section we give an algorithm that produces a set of solutions for the (INF-APPLY) rule problem that is sound (it finds only correct solutions) and complete (any other solution can be derived from those returned by the algorithm). To this end we proceed in three steps: (1) given a tallying problem, we show how to effectively produce a finite set of solutions that is sound (it contains only correct solutions) and complete (every other solution of the problem is less general—in the usual sense of unification, *ie*, it is larger wrt $\sqsubseteq$— than some solution in the set); (2) we show that if we fix the cardinalities of I and J, then it is possible to reduce the (INF-APPLY) rule problem to a tallying problem; (3) from this we deduce a sound and complete algorithm to semi-decide the general (INF-APPLY) rule problem and thus the whole inference system.

4.3.1 *Solution of the tallying problem.*

In order to solve the tallying problem for given Δ and C, we first fix some total order $\preccurlyeq$ —any will do— on the type variables occurring in C that are not in Δ (from now on, when we speak of type variables we will intend only type variables that are not in Δ). Next, we produce a set of constraint-sets that are in a particular form, first by *normalizing* the constraint-sets (so that at least one of the two types of every constraint is a type variable) and then by *merging* constraints that are on the same variables. Finally, we solve all these constraint-sets thus obtaining a sound and complete set of solutions of the tallying problem. To this end it is useful to define two operations on sets of constraint-sets:

Definition 4.8. *Let $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathscr{P}(\mathcal{T}\times\mathcal{T})$ be two sets of constraint-sets. We define $\mathcal{S}_1 \sqcap \mathcal{S}_2 \stackrel{\text{def}}{=} \{C_1 \cup C_2 \mid C_1 \in \mathcal{S}_1, C_2 \in \mathcal{S}_2\}$ and $\mathcal{S}_1 \sqcup \mathcal{S}_2 \stackrel{\text{def}}{=} \mathcal{S}_1 \cup \mathcal{S}_2$.*

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

Given a type t which is a summand of a normal form, that is, $t = \bigwedge_{p\in P} t_p \wedge \bigwedge_{n\in N} \neg t_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r$ and $k \in P' \cup N'$ we define $\mathsf{single}(\alpha_k, t)$ the constraint equivalent to $t \le \mathbb{0}$ in which α_k is "singled-out", that is,

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

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

We define a function norm that takes a type t and generates a set of *normalized* constraint-sets —*ie*, constraint-sets formed by constraints whose form is either $\alpha\le s$ or $s\le\alpha$— whose set of solutions is sound and complete w.r.t. the constraint $t \le \mathbb{0}$. This function is parametric in a *memoization* set M and the algorithm to compute it is given in Figure 4. If the input type t is not in normal form, then the algorithm is applied to the disjunctive normal form t' of t (end of line 6). Since a union is empty if and only if every summand that composes it is empty, then the algorithm generates a new constraint-set for the problem that equates all the summands to $\mathbb{0}$ (beginning of line 6). If a summand contains a top-level variable, then the smallest top-level variable is singled out (line 2). If there is no top-level variable and there are only basic types, then the algorithm checks the constraint by calling the subtyping algorithm and, accordingly, it returns either the unsatisfiable set of constraint-sets ($\varnothing$) or the one that is always satisfied ($\{\varnothing\}$) (line 3). Finally if there are only intersections of arrows and their negations, then the problem is decomposed into a set of subproblems by using

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

Figure 4. Constraint normalization

the decomposition rule of the subtyping algorithm for semantic subtyping (see [9] for details). The regularity of types ensures that the algorithm always terminates. The algorithm to solve the tallying problem for C and variables not in Δ, then, proceeds in three steps:

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

Step 2. Let $\mathcal{M} = \bigsqcup_{C\in\mathcal{S}} \mathsf{merge}(C, \varnothing)$. If $\mathcal{M} = \varnothing$ then **fail** else proceed to the next step. Where $\mathsf{merge}(C, M)$ is the function that performs the following two sub-steps:

M1. Rewrite C by applying as long as possible the following rules according to the order on the variables (lowest first) :
– if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
– if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1\vee s_2, \alpha)$;

M2. if there exist two constraints (s, α) and (α, t) in C such that $s\wedge\neg t \notin M$, then let $\mathcal{S}' = \{C\} \sqcap \mathsf{norm}(s\wedge\neg t, \varnothing)$ in return $\bigsqcup_{C'\in\mathcal{S}'} \mathsf{merge}(C', M\cup\{s\wedge\neg t\})$ else return $\{C\}$

Step 3. Solve all constraint-sets in $\mathcal{M}$.

Let us detail each step and in particular what step 3 consists of. In step 1 we have transformed the initial constraint-set into an equivalent set $\mathcal{S}$ of constraint-sets in which all constraints are on at least one type variable. In the second step we transformed every constraint-set in $\mathcal{S}$ in which each variable had several upper and lower bounds into a constraint-set in which each variable has at most one lower bound and at most one upper bound (obtained by unioning all its lower bounds and intersecting all its upper bounds). Thus each constraint-set in $\mathcal{M}$ represents a set of constraints of the form $C = \{s_i\leq\alpha_i\leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct variables and s_i and t_i are respectively $\mathbb{0}$ or $\mathbb{1}$ whenever the corresponding constraint is absent. To solve $\mathcal{M}$ —*ie*, implement step 3— we apply to every $C \in \mathcal{M}$ the following transformation:
– select the constraint $s \leq \alpha \leq t$ for the lowest variable α and replace it by the equality constraint $\alpha = (s \vee \beta) \wedge t$ where β is a fresh variable.
– in all other constraints in C replace every occurrence of α by $(s \vee \beta) \wedge t$.
It is clear that the constraint-set C has a solution for every possible assignment of α included between s and t if and only if the new constraint-set has a solution for every possible (unconstrained) assignment of β. We reiterate the process with the next type variable in the order and at the end every constraint-set has become a set of equations of the form $\{\alpha_i = u_i \mid i \in [1..n]\}$. By construction, this set of equations has the property that every variable that is smaller than or equal to (wrt $\preccurlyeq$) α_i may occur in u_i only under an arrow. This last property ensures the contractivity of the system of equations and by Courcelle [6] there exists a solution for this set, namely, a substitution from the type variables $\alpha_1, ..., \alpha_n$ into (possibly recursive regular) types $t_1, ..., t_n$ whose variables are contained in the fresh β_i's variables introduced in step 3 and the type variables in Δ, and are all universally quantified (no upper or lower bound).

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

Theorem 4.9 (Soundness and completeness)**.**

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

where $\approx$ means that the two substitutions map the same variable into equivalent types. Regularity of types ensures the termination of the algorithm and, hence, the decidability of the tallying problem.

4.3.2 Solution for application with fixed cardinalities

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

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

hold. Two complicacies are to be dealt with: (1) we must find *sets* of type substitutions, rather than a single substitution as in the tallying problem and (2) we have to get rid of the $\mathsf{dom}()$ function. If I and J have fixed cardinalities, then both difficulties can be easily surmounted and the whole problem be reduced to a tallying problem. To see how, consider the two inequations in (15). Since the two sets of substitutions are independent, then without loss of generality we can split each substitution σ_k (for $k\in I\cup J$) in two substitutions: a renaming substitution ρ_k that maps each variable in the domain of σ_k into a different fresh variable, and a second substitution σ'_k defined such that $\sigma_k = \rho_k \circ \sigma'_k$. The two inequations thus become $\bigwedge_{i\in I}(t\rho_i)\sigma'_i \leq \mathbb{0}\to\mathbb{1}$ and $\bigwedge_{j\in J}(s\rho_j)\sigma'_j \leq \mathsf{dom}(\bigwedge_{i\in I}(t\rho_i)\sigma'_i)$ Since the various σ'_k have disjoint domains, then we can union them into a single substitution $\sigma = \bigcup_{k\in I\cup J} \sigma'_k$, and the two inequations become $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \mathsf{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$. Now if we fix the cardinalities of I and J since the ρ_k are generic renamings, we have just transformed the problem in (15) into the problem of finding for two given types t_1 and t_2.[5] all substitutions σ such that

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

hold. Finally, we can prove (see Lemmas E.47 and E.48) that a type-substitution σ solves (16) if and only if it solves

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

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

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

4.3.3 Solution of the application problem

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

Step A: Generate the constraint-set $\{(t_1, t_2 \to \gamma)\}$ as explained in Subsection 4.3.2 (the constraint $t_1 \leq \mathbb{0} \to \mathbb{1}$ is implied by this one since $\mathbb{0} \to \mathbb{1}$ contains every arrow type) and apply the algorithm described in Subsection 4.3.1, yielding either a solution (a substitution for variables not in Δ) or a failure.

Step B: If all the constraint-sets failed at *Step 1* of the algorithm of Subsection 4.3.1, then fail (the term is not typeable). If they all failed but at least one did not fail in *Step 1*, then increment the cardinalities I and J to their successor in the dove-tail order and start from *Step A* again. Otherwise all substitutions found by the algorithm are solutions of the application problem.

Notice that the algorithm returns a failure only if all the constraint-sets fail at *Step 1* of the algorithm for the tallying problem. The reason is that that up to *Step 1* all the constraints at issue are on distinct occurrences of type variables: if they fail there is no possible expansion that can make the constraint-set satisfiable. In *Step 2* instead constraints of different occurrences of a same variable are merged. Thus even if the constraints fail, it may be the case that they will be satisfied by expanding different occurrences of a same variable into different variables. Therefore an expansion is tried.

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

If we apply the algorithm to `map even`, then we start with the constraint set $\{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq t \to \gamma\}$ where $t = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of `even` (we renamed only the variables of the type of `map`). At step 2 the algorithm generates a set of nine constraint-sets: one is unsatisfiable since it contains the constraint $t \leq \mathbb{0}$ (an intersection of arrows is never empty since it always contains $\mathbb{1} \to \mathbb{0}$ the type of the diverging functions); four of these are less general than some other (their solutions are included in the solutions of the other) and the remaining four are obtained by adding the constraint $\gamma \leq [\alpha_1] \to [\beta_1]$ respectively to $\{\alpha_1 \leq \mathbb{0}\}$, $\{\alpha_1 \leq \texttt{Int}, \texttt{Bool} \leq \beta_1\}$, $\{\alpha_1 \leq \alpha \setminus \texttt{Int}, \alpha \setminus \texttt{Int} \leq \beta_1\}$, $\{\alpha_1 \leq \alpha \vee \texttt{Int}, (\alpha \setminus \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$, yielding the following four solutions for γ: $\{\gamma = [] \to []\}$, or $\{\gamma = [\texttt{Int}] \to [\texttt{Bool}]\}$, or $\{\gamma = [\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]\}$, or $\{\gamma = [\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}]\}$. Of these solutions only the last two are minimal and since both are valid we can take their intersection, yielding the type expected in the introduction. Or we can dully follow the algorithm, perform an iteration, expand the type of the function, yielding $\{((\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1]) \wedge ((\alpha_2 \to \beta_2) \to [\alpha_2] \to [\beta_2]) \leq t \to \gamma\}$ for which the minimal solution will be, as expected:
$\{\gamma = ([\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]) \wedge ([\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}])\}$

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

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

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

5. Compilation into Core$\mathbb{C}$Duce

We want to compile the calculus of (10) into a variant of the (monomorphic) Core$\mathbb{C}$Duce, whose syntax is defined by the grammar (2) (Section 2) and whose types defined by the grammar in (7) (Section 3) but *where type variables are considered ground basic types* (*ie*, they cannot be instantiated). The reduction and typing rules are those defined for $\mathbb{C}$Duce by Frisch, Castagna, and Benzaken [9] in which the subtyping relation is the one of Castagna and Xu [3]. To compile a term e that contains explicit type-substitutions, we first apply relabeling in order to push all the explicit substitutions into decorations of the underlying λ-abstractions. Then we translate the body of each λ-abstraction into a set of nested type-case expressions that test the dynamic type of the actual parameter of the function and accordingly branch into the version of the body with the corresponding relabeling. The set

of type-cases is generated so as to cover all possible cases that can be obtained by combining the input types of the interface and the substitutions of the decorations of the λ-abstraction.

The translation we wish to define is *type-driven*, and therefore occurs on an annotated, well-typed expression of the polymorphic calculus. The translation relies on an extension of the (monomorphic) type-case expression that features binding, which we write $(x{=}e){\in}t\,?\,e_1 : e_2$, and that can be encoded by the term $(\lambda^{((s\wedge t)\to t_1)\wedge((s\wedge\neg t)\to t_2)}x.x{\in}t\,?\,e_1 : e_2)e$, where s is the type of e, t_1 the type of e_1 and t_2 the type of e_2. We add another twist to this construct and define a particular (purely) syntactic sugar: $x\,\boldsymbol{\in}\,t\,?\,e_1 : e_2$ (notice the boldface "belongs to" symbol) which stands for $(x{=}x){\in}t\,?\,e_1 : e_2$. The reader may wonder what is the interest of binding a variable to itself. Actually, the two occurrences of x in $(x{=}x){\in}t$ denote two distinct variables: the one on the right is recorded in the environment with some type s; this variable does not occur either in e_1 or e_2 because it is hidden by the x on the left; this binds the occurrences of x in e_1 and e_2 but with different types, $s\wedge t$ in e_1 and $s\wedge\neg t$ in e_2. This allows the system to use different type assumptions for x in each branch. We already silently used this construction in the body of `map` in Section 2 (which should have been): $\lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell\,\boldsymbol{\in}\,\mathtt{nil}\,?\,\mathtt{nil} : (f(\pi_1\ell),mf(\pi_2\ell))$, since typing the "$\pi_i$" in the second branch requires that $\ell : [\alpha]\setminus\mathtt{nil}$.

We can now use this "binding" type-case to express our translation to Core$\mathbb{C}$Duce, which we just illustrate on an example (the definition of the translation and the proof of its soundness are given in Section F). Consider the Church Boolean `TRUE` defined as $\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x$ in a context where its interface must be instantiated with two given types s and t, that is $(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)[\{s/\alpha\},\{t/\alpha\}]$. To produce its monomorphic counter part, we first produce a (still polymorphic) term where the set of type-substitutions decorates the outer lambda: $\lambda^{\alpha\to\alpha\to\alpha}_{[\{s/\alpha\},\{t/\alpha\}]}x.\lambda^{\alpha\to\alpha}y.x$, which is obtained from the relabeling $\mathtt{TRUE}@[\{s/\alpha\},\{t/\alpha\}]$. We now must translate this function, so that: (*i*) its interface is monomorphic (*ie*, there must not remain any unapplied substitution) and (*ii*) its body simulates the relabeling @ during evaluation. To this end, we use the binding type-case:

$$\begin{aligned}\lambda^{(s\to s\to s)\wedge(t\to t\to t)}x\,.\,x\,\boldsymbol{\in}\,s\wedge t\,?\,[\![(\lambda^{\alpha\to\alpha}y.x)@[\{s/\alpha\},\{t/\alpha\}]]\!] :\\ x\,\boldsymbol{\in}\,s\,?\,[\![(\lambda^{\alpha\to\alpha}y.x)@[\{s/\alpha\}]]\!] :\\ x\,\boldsymbol{\in}\,t\,?\,[\![(\lambda^{\alpha\to\alpha}y.x)@[\{t/\alpha\}]]\!] : \mathit{dummy}\end{aligned}$$

where $[\![e]\!]$ denotes the translation of e and *dummy* is any expression (*eg*, a constant). The latter will be never returned since the branch is selected only for values in $\neg(s\vee t)$, that is, outside the domain of the function. Finally, computing the various translations yields:

$$\begin{aligned}\lambda^{(s\to s\to s)\wedge(t\to t\to t)}x\,.\,x\,\boldsymbol{\in}\,s\wedge t\,?\,\lambda^{(s\to s)\wedge(t\to t)}y.x :\\ x\,\boldsymbol{\in}\,s\,?\,\lambda^{s\to s}y.x :\\ x\,\boldsymbol{\in}\,t\,?\,\lambda^{t\to t}y.x : \mathit{dummy}\end{aligned}$$

as expected. Notice that the smaller types are tested first in order to comply with the matching order. Also notice that the "binding" type-case is essential since the only way to type the various branches is under the hypothesis that x has the type tested in the case at issue.

The translation is sound:

Theorem 5.1. *If* $\vdash e : t$ *and* $[\![e]\!]\rightsquigarrow^*_C v'$*, then*
1. $\exists v\ s.t.\ e\rightsquigarrow^* v$ *and* $[\![v]\!]=v'$.
2. $\vdash_C v' : t$.

where "$\vdash_C$:" and "$\rightsquigarrow_C$" denote the type system and reduction relation of Core$\mathbb{C}$Duce. In other words, a translated term $[\![e]\!]$ really is a static representation of all the possible instanciations that may happen at runtime, for a term e with explicit substitutions.

While this translation allows us to encode the relabeling operation with monomorphic constructs, one may wonder how it would impact a real programming language. In particular, the size of the introduced type-case seems large (impacts code size), and the generation of the type case depends on the decorations present on the lambda, which in turns depend on the *argument* of the function (impacts modularity). Even if for a function $\lambda^{\wedge_{i\in I}s_i\to t_i}_{[\sigma_j]_{j\in J}}x.e$ the number of case is exponential in $|I|\times|J|$, we can note that type cases are only needed when the resulting value is a lambda with type variables, that is, in the case of a polymorphic partial application. Another drawback is that since the resulting type depends on the particular instantiation of the type variables, one may need to generate several translations of a function, one for each call site with distinct argument types. This clearly breaks modularity and there seem that there is little hope to recover it with this approach.

6. Design choices and extensions

For the sake of presentation we omitted three key features in the calculus: constants, recursive functions, and pairs. The first two can be straightforwardly added with minor modifications. In particular, for recursive functions, whose syntax is $\mu f^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e$ (without decoration in the implicitly typed calculus), it suffices to add in the type environment the recursion variable f associated with the type obtained by applying the decoration to the interface, that is $\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j$: the reader can refer to Section 7.5 in [9] 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. The reduction rules and typing rules for pairs and projections are standard. The rule for pairs in the algorithmic system $\vdash_{\mathcal{A}}$ 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 [9]). Finally, for what concerns the inference system $\vdash_{\mathcal{I}}$, the rule for pairs remains once more the same. Instead as expected, the rule for projection $\pi_i e$ needs some special care since if the type inferred for e is, say, t, then we need to find a set of substitutions $[\sigma_i]_{i\in I}$ such that $\bigwedge_{i\in I}t\sigma_i \leq \mathbb{1}\times\mathbb{1}$. This problem can be solved by using the very same technique we introduced for $\bullet_\Delta$, namely by solving a sequence of tallying problems generated by increasing at each step the cardinality of I. All the details can be found in the full version.

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 tallying problems at run-time. 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 .

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

7. Conclusion

The work presented here provides all the theoretical basis and machinery needed to start the design and implementation of polymorphic functional languages for semi-structured data. Although many problems are still to be solved most of them are of theoretical nature with negligible impact on practical aspects. Foremost, the problem of the decidability of inference of type-substitutions. The property that our calculus can express intersection type systems *à la* Barendregt, Coppo, and Dezani is not of much help: if we know that a term is well-typed, then it is not difficult to extract from its type the interfaces of the functions occurring in it; undecidability for intersection type systems tells us that it is not possible to decide whether these interfaces exist; but here we are in a simpler setting where the interfaces are given and we "just" want to decide whether they are correct. The problem we cannot solve is when should we stop trying to expand the types of a function and of its argument, in particular when they are union types. The non determinism of the implicitly typed calculus has also a negligible practical impact, insofar as it is theoretical (in practice, the semantics is deterministic but implementation dependent) and it concerns only the case when the type of (an instance of) a polymorphic function is tested: in our programming experience with $\mathbb{C}$Duce we never met a test for a function type. Nevertheless, it may be interesting to study how to remove such a latitude either by a defining a canonical choice for the instances deduced by the inference system (a problem related to the existence of principal types), or by imposing reasonable restrictions, or by checking the flow of polymorphic functions by a static analysis.

On the practical side the most interesting direction of research is to study efficient compilation of the polymorphic language. A naive compilation technique that would implement the beta reduction of the explicitly-typed calculus as defined by (12) is out of question since it would be too inefficient. The compilation technique described in Section 5 coupled with some optimization techniques that would limit the expansion in type-cases to few necessary cases can provide a rapidly available solution for a prototype implementation that reuses the efficient run-time engine of $\mathbb{C}$Duce (interestingly, with such a technique the compilation does not need to keep any information about the source: the compiled term has all the information needed). However it could not scale up to a full-fledged language since it is not modular (a function is compiled differently according to the argument it is applied to) and the use nested type-cases, even in a limited number of cases, causes an exponential blow-up that hinders any advantage of avoiding dynamic relabeling. We think that the solution to follow is a smart compilation in typed virtual machines using JIT compilation techniques. Since the types in the interfaces of λ-abstractions are used to determine the semantics of the type-cases, then relabeling cannot be avoided. However relabeling is necessary only in one very particular situation, that is in case of partial application of curried polymorphic functions. Therefore we plan to explore compilation techniques such as those introduced in the ZINC virtual machine to single out partial applications and apply possible relabeling just to them. This can be further coupled with static analysis techniques that would limit relabeling only to partial applications that may end up in a type cases or escape in the result, yielding an efficient runtime on par with the monomorphic version of $\mathbb{C}$Duce.

Finally, all the machinery developed here can be reused to perform type reconstruction, that is, to assign a type to functions whose interface is not specified (quite a desirable feature, especially for local functions). The idea is to type the body of the function under the hypothesis that the function has the generic type $\alpha\to\beta$ and deduce the corresponding constraints. The minimum requirement is to obtain at least the types deduced by the ML inference system, but preliminary experiments show that in general it is possible to deduce far more precise types. For instance, by running our experiments by hand we deduced for `map` the following type:

$$(\alpha\to\beta) \to (\,(\,[\,]\to[\,]\,)\wedge(\,[\alpha]\setminus[\,]\to[\beta]\setminus[\,]\,)\,)\wedge$$
$$(\mathbb{1}\to\mathbb{0}) \to (\,(\,[\,]\to[\,]\,)\wedge(\,[\alpha]\setminus[\,]\to\mathbb{0}\,)\,)$$

which is extremely (even too much) precise since, for instance, it states that the application of `map` to any function and the empty list returns the empty list, and that if `map` is applied to a diverging function, then the resulting function does not diverge only if this is applied to the empty list. Of course, general type reconstruction is undecidable for this system and we will not be able to avoid to stumble into the paradoxes outlined in Section 2, but we reckon it should not be hard to find reasonable restrictions on the complexity of the inferred types to ensure decidability without hindering type inference in practical cases.

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

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

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

[5] M. Coppo, M Dezani, and B. Venneri. Principal type schemes and lambda-calculus semantics. In R. Hindley and J. Seldin, editors, *To H.B. Curry. Essays on Combinatory Logic, Lambda-calculus and Formalism*, pages 480–490. Academic Press, 1980.

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

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

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

[9] 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.

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

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

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

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

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

[15] 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.

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