Programming with union, intersection, and negation types*

Giuseppe Castagna
CNRS - Université de Paris

Abstract. In this essay I present the advantages and, I dare say, the beauty of programming in a language with set-theoretic types, that is, types that include union, intersection, and negation type connectives. I show by several examples how set-theoretic types are necessary to type some common programming patterns, but also how they play a key role in typing several language constructs—from branching and pattern matching to function overloading and type-cases—very precisely. I start by presenting the theory of types known as semantic subtyping and extend it to include polymorphic types. Next, I discuss the design of languages that use these types. I start by defining a theoretical framework that covers all the examples given in the first part of the presentation. Since the system of the framework cannot be effectively implemented, I then describe three effective restrictions of this system: (*i*) a polymorphic language with explicitly-typed functions, (*ii*) an implicitly typed polymorphic language *à la* Hindley-Milner, and (*iii*) a monomorphic language that, by implementing classic union-elimination, precisely reconstructs intersection types for functions and implements a very general form of occurrence typing.
I conclude the presentation with a short overview of other aspects of these languages, such as pattern matching, gradual typing, and denotational semantics.

Keywords: Type-theory · Subtyping · Union types · Intersection types · Negation types · Polymorphism · Overloading · Semantic subtyping.

1 Introduction

In this essay we present the use of set-theoretic types in programming languages and outline their theory. Set theoretic types include union types $t_1 \vee t_2$, intersection types $t_1 \wedge t_2$, and negation types $\neg t$. In strict languages it is sensible to interpret a type as the set of values that have that type (e.g., `Bool` is interpreted as the set containing the values `true` and `false`). Under this assumption, then, $t_1 \vee t_2$ is the set of values that are either of type t_1 *or* of type t_2; $t_1 \wedge t_2$ is the set of values that are both of type t_1 *and* of type t_2; $\neg t$ is the set of all values that are *not* of type t. Set-theoretic types are polymorphic when they include type variables (that we range over by Greek letters, α, β,...).

To give an idea of the kind of programming that set-theoretic types enable and that we describe in this article, consider the classic *recursive flatten* function that transforms arbitrarily nested lists in the list of their elements. In a ML-like language with pattern matching it can be defined as simply as

* Based on joint work with Pietro Abate, Véronique Benzaken, Alain Frisch, Hyeonseung Im, Victor Lanvin, Mickäel Laurent, Sergueï Lenglet, Matthew Lutze, Kim Nguyen, Luca Padovani, Tommaso Petrucciani, and Zhiwu Xu

```
let rec flatten = function
   |  [] -> []
   |  h::t -> (flatten h)@(flatten t)
   |  x -> [x]
```

The function `flatten` returns the empty list `[]` when its argument is an empty list; if its argument is a non-empty list, then it flattens the argument's head `h` and tail `t` and returns the concatenation (denoted by `@`) of the results; if its argument is not a list, then `flatten` returns the list containing just the argument.

The `flatten` function is completely polymorphic: it can be applied to any argument and, if lists are finite, always terminates. Although its semantics is easy to understand, giving a simple and general polymorphic type to this function (i.e., a type that, without complex metaprogramming constructions, allows the function to be applied to every well-typed argument) defies all existing programming languages [29] with a single exception: CDuce [17]. This is because CDuce is a language that uses a complete set of set-theoretic type connectives and we need all of them (union, intersection, and negation) to define `Tree(`α`)`, the type of nested lists whose elements are of type α:

```
type Tree(α) = (α\List(Any)) | List(Tree(α))
```

in this type definition "`|`" denotes a union, "`\`" difference (i.e., intersection with the negation: $t_1 \backslash t_2 \stackrel{\text{def}}{=} t_1 \wedge \neg t_2$), `List(`$t$`)` is the type of lists of elements of type t, and `Any` is the type of all values, so that `List(Any)` is the type of any list.[1] In words, `Tree(`α`)` is the type of nested lists whose leaves (i.e., the elements that are not lists) have type α. Thus it is either a leaf or a list of `Tree(`α`)`. Then, it just suffices to annotate `flatten` with the right type

```
let rec flatten: Tree(α)→List(α) = function ...
```

for the definition to type-check in CDuce. In other terms, in CDuce the above definition of `flatten` is of type $\forall\alpha.$`Tree(`α`)→List(`α`)`. The important point is that whatever the type of the argument of `flatten` is, the application is always well-typed: if the argument is not a list, then α is instantiated to the type of the argument; if it is a list, then it is also a nested list, and α is instantiated with the union of the types of the non-list elements of this nested list. In other terms, `flatten` can be applied to expressions of any type and the type inferred for such an application is `List(`t`)` where the type t is the union of the types of all the leaves of the argument, a non-list argument being itself a leaf. For instance, the type statically deduced for the application

```
flatten [3 "r" [4 [true 5]] ["quo" [[false] "stop"]]]
```

is `List(Int|Bool|String)`.[2]

[1] We mainly use "`&`", "`|`", and "`\`" in code snippets for intersections, unions, and differences and reserve "$\vee$" , "$\wedge$", and "$\neg$" for formal types.

[2] CDuce syntax is actually slightly different. The valid CDuce code for our example is:

```
type Tree('a) = ('a\[Any*]) | [ (Tree('a))* ]
let flatten ( (Tree('a)) -> [ 'a* ] )
     | [] -> []
     | [h;t] -> (flatten h)@(flatten t)
     | x -> [x]
```

and the type deduced by CDuce for the application is more precise than the above since it is: `[ (Bool | 3--5 | 'o'--'u')* ]` ("`--`" is for intervals and `[`t`*]` for lists of t elements).

The overall type inference system is quite expressive: it types more expressions or gives more precise types (but worse error messages) than typical core-ML systems. However, such a deduction is possible only because the function `flatten` is explicitly typed: fail to specify the type annotation `Tree(α)→List(α)` and `flatten` will be rejected by all existing type-checking systems. That current type-systems cannot infer a type as sophisticated as the type of `flatten` without an explicit annotation is not surprising as it combines the full palette of set-theoretic connectives (union, intersection, *and* negation) and recursive types. However, an important limitation of current programming languages is that none of them is able to infer intersection types for functions without explicit annotations. So while any ML-like language can deduce for

```
let not_ = fun x -> if x then false else true
```

the type `Bool→Bool`, current languages with intersection types cannot deduce for the same function the more precise type `(true→false)&(false→true)` (where `true` and `false` denote the singleton types containing the respective values) without being instructed to do so by an explicit type annotation. The latter type is an intersection of types, meaning that `not_` has both type `true→false` *and* type `false→true`. The intersection type is more precise than the type `Bool→Bool`: it states that when `not_` is applied to an expression of type `true`, the result is not only a Boolean but actually `false`, and likewise for arguments of type `false`. As we show later on, this degree of "precision" between two types is formally defined since `(true→false)&(false→true)` is a strict subtype of `Bool→Bool`: every function of the former type is also of the latter type, but not viceversa. Actually, if we adopt for if-then-else a semantics similar to the one in JavaScript and consider every expression different from false to be "truthy" (i.e., equivalent to true), then an even better intersection type for `not_` would be `(¬false→false)&(false→true)` which completely specifies the behavior of the function since the function `not_` above returns `false` for every argument that is not `false` (i.e., for "truthy" values such as `42`). The more precise is a type the less functions it types, a most precise type being one that, as `(¬false→false)&(false→true)` completely defines the semantics of a function.

We will discuss recent systems by Castagna et al. [5, 16] that are able to deduce the most precise intersection type for the definition of `not_` even without any annotation. This inference is obtained by considering the conditional in the definition of `not_` akin to a type-case that tests whether `x` is of type `¬false` or not. The body of `not_` is then analyzed separately under the hypotheses that `x` has type `¬false` and `¬¬false` (i.e., `false`), yielding the corresponding intersection type. This is performed also for multiple arguments, allowing the cited systems to deduce for

```
let and_ = fun x y ->
    if x then (if y then false else true) else false
```

the following type
`(false→Any→false) & (¬false → ((¬false→true)&(false→false)))`,
which completely specifies the semantics of `and_`: if the first argument is `false`, then the result will be `false` for a second argument of any type; if the first argument is not `false` then the result will be `true` for a second argument not `false`, and `false` otherwise. It is important to notice that the analysis performed in [5, 16] is type-theoretic

rather than syntactic: the arrows forming the intersection type of a function are not determined by a syntactic recognition of type-cases, but are inferred from the types involved in the definition of the function. To illustrate the advantages of a type-based approach over a syntactic one it suffices to consider the following definition of `or_` that combines the previous `not_` and `and_` definitions according to De Morgan's laws:

```
let or_ = fun x y -> not_ (and_ (not_ x) (not_ y))
```

The type `(¬false→Any→true)&(false→((¬false→true)&(false→false)))` is deduced for this definition despite that no branching appears in it. For the same reasons we could equivalently define the previous `and_` function using a double call to `not_` so that a second argument that is not false yields true:

```
let and_ = fun x y -> if x then not_ (not_ y) else false
```

and obtain the same type as for the previous definition of `and_`.

The ultimate goal of the research we present in this article is to define a programming language whose type-inference subsumes ML-core type-inference, that can also deduce intersections of arrows types for implicitly-typed functions such as `not_`, `and_`, and `or_`, and where the programmer would be obliged to specify type annotations only in particular cases, such as for `flatten`. Unfortunately, while there exist systems that provide some of these features, it is not currently possible to have all of them simultaneously in a unique language, as we discuss in Section 4.

2 Motivations

In the previous section we gave few specific examples of use of polymorphic set-theoretic types. One of the key features of these types that makes them versatile is that they encompass all the three main forms of polymorphism, namely:

Parametric polymorphism: which describes code that can act uniformly on any type, using type variables that can be instantiated with the appropriate type (e.g., typing the identity function as $\forall\alpha.\alpha\to\alpha$). In this article we consider only the so-called *prenex* or *second-class* polymorphism (in the sense of [31]) where variable quantification cannot appear below type constructors or type connectives.

Ad-hoc polymorphism: which allows code that can act on more than one type, possibly with different behavior in each case, as in function overloading (e.g., allowing `+` to have both types $\texttt{Int}\times\texttt{Int}\to\texttt{Int}$ and $\texttt{String}\times\texttt{String}\to\texttt{String}$, corresponding to different implementations).

Subtype polymorphism: which creates a hierarchy of more or less precise types for the same code allowing it to be used wherever any of these type is expected (e.g., typing `3` as both `Int` and `Real`, with $\texttt{Int}\leq\texttt{Real}$).

In this section we reframe polymorphic set-theoretic types in a more general setting showing how these types allow us to type several features and idioms of programming languages effectively. We illustrate this with some examples.

UNION TYPES: The simplest use cases for union types include branching constructs. In a language with union types, we can type precisely conditionals that return results of

different types: for instance, `if` e `then 3 else true` has type `Int`$\vee$`Bool` (provided that e has type `Bool`). Without union types, it could have an approximated type (e.g., a top type) or be ill-typed. Similarly, we can use union types for structures like lists that mix different types: we already saw an example of this in the previous section when an application of `flatten` returned the list `[3 "r" 4 true 5 "quo" false "stop"]` of type `List(Int|Bool|String)`.

This makes union types invaluable to design type systems for existing untyped languages: witness for example their inclusion in Typed Racket [59] which allows the incremental addition of statically-checked type annotations on a dialect of Scheme and in TypeScript [43] and Flow [21] which extend JavaScript with static type checking.

FUNCTION OVERLOADING: We can use intersection types to assign more than one type to an expression. This is particularly relevant for functions. We have already seen it in the previous section for the functions `not_`, `and_`, and `or_`. But even the simple identity function can be typed as $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$: this means it has both types `Int`$\to$`Int` and `Bool`$\to$`Bool`, because it maps integers to integers and Booleans to Booleans. This type describes a uniform behavior over two different argument types (the function uniformly maps an argument into itself independently from the argument's type), which can also be described using parametric polymorphism. However, intersection types let us express *ad-hoc* polymorphism (i.e., function overloading) if coupled with some mechanism that allows functions to test the type of their arguments. For example, let $(e\in t)\,?\,e_1:e_2$ be the type-case expression that first evaluates e to a value v and continues as e_1 if v is of type t, and as e_2 otherwise. The function $\lambda x.(x\in\texttt{Int})\,?\,(x+1):\neg x$ checks whether its argument x is an `Int` and returns the successor of x in that case, its negation otherwise. The function can be applied to integers, returning their successor, and to Booleans, returning their negation. This behavior can be described by the same type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ used for the identity function, but does not correspond to parametric behavior.

A function of type $(t_1\to t_1')\wedge(t_2\to t_2')$ can be safely applied to any argument of type $t_1\vee t_2$, since it is defined on both t_1 and t_2. We know that the result will always have type $t_1'\vee t_2'$. However, if we know the type of the argument more precisely, we can predict the type of the result more precisely: for example, if the argument is of type t_1, then the result will be of type t_1'. So the intersection type of the function $\lambda x.(x\in\texttt{Int})\,?\,(x+1):\neg x$ allows us to deduce that its application to an integer will return an integer.

We said that the type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ can be assigned to the identity function and expresses parametric behavior. In this respect, we can see intersection types as a finitary form of parametric polymorphism; however, they are not constrained to represent uniform behavior, as our other example illustrates. Conversely, we could see a polymorphic type (or type scheme) $\forall\alpha.\alpha\to\alpha$ as an infinite intersection (intuitively, $\bigwedge_{t\in\mathsf{Types}} t\to t$, where Types is the set of all types), but infinite intersections do not actually exist in our types.

OCCURRENCE TYPING: *Occurrence typing* or *flow typing* [60, 47, 18] is a typing technique pioneered by Typed Racket that uses the information provided by a type test to specialize the type of some variables in the branches of a conditional. For example, if x is of type `Int`$\vee$`Bool`, then to type the expression $(x\in\texttt{Int})\,?\,e_1:e_2$ we can assume that the occurrences of x in e_1 have type `Int` and those in e_2 have type `Bool`, because

the first branch will only be reached if x is an $\texttt{Int}$ and the second if it is not an $\texttt{Int}$ (and is therefore a $\texttt{Bool}$). Intersection and negation types are useful to describe this type discipline. If we test x for the type $\texttt{Int}$ as in our example, then we can assign to x the type $\texttt{Int}$ if the test succeeds and $\neg\texttt{Int}$ if it fails. Using intersections, we can add this information to what we already knew, so the type of x is $(\texttt{Int}\vee\texttt{Bool})\wedge\texttt{Int}$ (which is equivalent to $\texttt{Int}$) in the first branch and $(\texttt{Int}\vee\texttt{Bool})\wedge\neg\texttt{Int}$ (which is equivalent to $\texttt{Bool}$) in the second branch. We already implicitly used this technique when, earlier in this section, we said that $\lambda x.(x{\in}\texttt{Int})\,?\,(x+1)\,:\,\neg x$ is of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ since we must assume that x is of type $\texttt{Int}$ to type $x+1$ and that it is of type $\texttt{Bool}$ to type $\neg x$: we took into account the result of the type-test.

This method of refining types according to conditionals is important in type systems for dynamic languages and in those that enforce null safety: some examples include Ceylon [39], Dart [28], Flow, Kotlin [38], Typed Racket, TypeScript, and Whiley [48]. In particular, Ceylon relies on intersection types [39, 44] and Whiley on both intersection and negation types [47].

This same method is at the basis of the systems by Castagna et al. [5, 16] we cited in the introduction as the sole capable of inferring intersection of arrow types for functions without needing explicit type annotations. These systems use the characteristics of set-theoretic types, as outlined above, to implement and generalize occurrence typing and decide how to split the type analysis to deduce intersection types for function expressions, as we detail in Section 4.4.

ENCODING DISJOINT UNION TYPES: Disjoint union types (also known as variant or sum types) are an important feature of functional programming languages. They can be encoded using union types and product (or record, or object) types. It is also useful to have *singleton types*, that is, types that correspond to a single value as we already saw with the two types $\texttt{true}$ and $\texttt{false}$ for the respective constants, both subtypes of the Boolean type (which we can then see as equivalent to the union $\texttt{true}\vee\texttt{false}$).

For instance, consider this example in Flow.[3]

```
type Success = { success: true, value: boolean }
type Failed = { success: false, error: string }
type Response = Success | Failed

function handleResponse(response: Response) {
  if (response.success) { var value: boolean = response.value }
  else { var error: string = response.error }
}
```

The type Response is the union (denoted by |) of two object types: both have a Boolean field success, but the types state that success must be true for objects of type Success and false for objects of type Failure. An analogous type could be declared in OCaml as `type response = Success of bool | Failed of string`. Occurrence typing is used to distinguish the two cases, like pattern matching could do in ML: if response.success is true, then response must be of type Success; if it is false, response must be of type Failure.

[3] This example is copied verbatim from the documentation of Flow, available at https://flow.org/en/docs/types/unions.

ENCODING BOUNDED POLYMORPHISM: Using union and intersection types, we can encode bounded polymorphism without adding specific syntax for the bounds in quantifications. For example, a type scheme with bounded polymorphism is $\forall(\alpha \leq t).\alpha \to \alpha$: it describes functions that can be applied to arguments of any subtype of t and that return a result of the same type as the argument. Using intersection types, we can write this type scheme as $\forall\alpha.(\alpha \wedge t) \to (\alpha \wedge t)$, writing the bound on the occurrences of the type variable and not on the quantifier: as the previous type scheme it accepts only arguments of a type smaller than t and returns results of the same type.[4] Analogously, we can use union types to represent lower bounds: in general, a bound $t' \leq \alpha \leq t$ on a type variable can be eliminated by replacing every occurrence of α in the type with $t' \vee (\alpha \wedge t)$, yielding bounded quantifications of the form $\forall(t' \leq \alpha \leq t).t''$. Notice however that the form of bounded polymorphism we obtain by this encoding is limited, insofar as two bounded types may be in subtyping relation only if they have the same bounds,[5] yielding a second-class polymorphism more akin to Fun [3] (where $\forall(\alpha \leq s_1).t_1 \leq \forall(\alpha \leq s_2).t_2$ is possible only for $s_1 = s_2$) than to $F_{<:}$ [4] (which allows $\forall(\alpha \leq s_1).t_1 \leq \forall(\alpha \leq s_2).t_2$ even for $s_1 \neq s_2$, typically $s_2 \leq s_1$).

As a concrete example, consider again the `flatten` function of the introduction. We can give this function a type slightly more precise than the one in the introduction by using the annotation `Tree(α)→List((α\List(Any)))` which states that the elements of the resulting list cannot be themselves lists: the list is flat. With such an annotation the current version of polymorphic CDuce deduces for `flatten` the (equivalent) type `∀α.Tree(α\List(Any)))→List((α\List(Any)))`. Since $t_1 \setminus t_2 = t_1 \wedge \neg t_2$, this corresponds to the bounded quantification `∀(α ≤ ¬List(Any)).Tree(α)→List(α)` stating that α can be instantiated with any type that is not a list (though the domain `Tree(α)` can still match any type).

TYPING PATTERN MATCHING: Pattern matching is widely used in functional programming. However, using pattern matching in ML-like languages, we can write functions that cannot be given an exact domain in the type system. For instance, the OCaml code

```
let f = function 0 -> true | 1 -> false
```

defines a function that can only be applied to the integers 0 and 1, but OCaml infers the unsafe type int → bool (albeit with a warning that pattern matching is not exhaustive). The precise domain cannot be expressed in OCaml. Using set-theoretic types and singleton types, we can express it precisely as $0 \vee 1$. Furthermore, we can use the inference of intersection of arrows we outlined in the introduction, which for the function `f` gives the type `(0→true)&(1→false)` which completely defines the semantics of `f`.

More generally, set-theoretic types are a key ingredient to achieve a precise typing of pattern matching. For instance, in a language as CDuce the set of values that

[4] Of course, the syntax $\forall(\alpha \leq t).\alpha \to \alpha$ is likely to be clearer to a programmer and should be privileged. The point is that set-theoretic types provide all is needed to account for bounded polymorphism without the need to add new machinery or rules for this sort of typing.

[5] This is necessary only for bounded variables that occur in the type both in covariant and in contravariant positions. Notice however that variables that do not satisfy this property can be easily eliminated by replacing `Any` for all covariantly-only occurring variables, and `¬Any` for all contravariantly-only occurring ones.

match a given pattern form a type (see Section 5.1). This can be used to precisely type a single branch of pattern matching since the set of values processed by a given branch are all the values in the type of the matched expression *minus* (set-theoretic difference) the *union* (set-theoretic union) of all the values matched by the preceding branches, *intersected* (set-theoretic intersection) with the values matched by the pattern of the branch at issue. We will give all the details about it in Section 5.1 but in this essay we already met several examples of application of this technique. For instance, in the definition of `flatten` in the introduction, the first pattern `[]` captures the empty list, that is the singleton type `[]`; the second pattern `h::t` captures all the non-empty lists, that is the type `List(Any)\[]`; the third pattern `x` captures all values, that is the type `Any`. From that CDuce deduces that the variable `x` in the third branch will capture any value that is not captured by the two previous patterns, that is `Any` minus `[]`$\vee$`(List(Any)\[])`=`List(Any)` (i.e., the type captured by the first pattern union the type captured by the second pattern) and deduces that the list returned by the branch cannot contain other lists: this is the key mechanism that allows CDuce to type `flatten` also when it is annotated with the more precise type `Tree(`α`)`$\rightarrow$`List((`α`\List(Any)))`.

NEGATION TYPES: We have already seen several applications of negation types. In the examples we gave we mostly used type differences, since they better fit a usage for programming, but this is completely equivalent since negation and differences can encode each other (i.e., $t_1 \backslash t_2 = t_1 \wedge \neg t_2$ and $\neg t = \texttt{Any} \backslash t$). As a matter of fact, type difference is pervasive in all programming languages that use union types. However the vast majority of these languages hide type difference to the programmer and use it only as a meta-operation on types, implemented in the type-checker which uses it to produce precise types or analyze the flow of values in pattern matching. For instance, to type the type-case expression `(`x`&in;Int)?`e_1`:`e_2 where x has type `Int`$\vee$`Bool`, a type checker such as the one for Flow would assume that the occurrences of x in e_2 have type `Bool`, since this type is the result of `(Int`$\vee$`Bool)\Int`. But to compute this result it would use an internal type-difference operator without exposing it to the programmer: the programmer can write its types by using unions, intersections, but not differences.

Nevertheless, first-class difference (or negation) types are useful to type several programming patterns and idioms. We already seen this with the `flatten` function, whose type critically relies on the use of difference types to define the type of nested lists. But much simpler examples exist: consider for instance a function $\lambda x.$`(`x`&in;Int)?(`$x+1$`):`x. It can act on arguments of any type, computing the successor of integers and returning any other argument unchanged. Using intersection and difference types, plus parametric polymorphism, we can type it as $\forall\alpha.(\texttt{Int} \rightarrow \texttt{Int}) \wedge (\alpha\backslash\texttt{Int} \rightarrow \alpha\backslash\texttt{Int})$, which expresses the function's behavior fairly precisely and that corresponds to the bounded quantification $\forall(\alpha \leq \neg\texttt{Int}).(\texttt{Int} \rightarrow \texttt{Int}) \wedge (\alpha \rightarrow \alpha)$: the function returns integer results for integer arguments and returns α results for α arguments that are not integers.

Although the example $\lambda x.$`(`x`&in;Int)?(`$x+1$`):`x is not very enthralling, it yields a type that is extremely useful in practice since it precisely types functions defined by pattern matching with a last default case that returns the argument. In [11, Appendix A] the reader can find the detailed presentation of a couple of compelling examples of standard functions (on binary trees and SOAP envelopes) whose typing is only possible

or can be improved thanks to set-theoretic types that use differences as in the example above. In particular, [11] shows how to type the function to insert a new node in a red-black tree (one of the most popular implementation of self-balancing binary search tree, due to Guibas and Sedgewick [30]). The types used in the definition given in [11] enforce three out of the four invariants of red-black trees,[6] requiring only the addition of type annotations to the code and no other change to a standard implementation due to Okasaki [45, 46] to which the reader can refer for more details. The core of Okasaki's definition is the `balance` function which is defined (in our ML-like syntax) as follows:

```
type RBTree(α) = Leaf | ( (Red|Black), α, RBTree(α), RBTree(α))

let balance = function
  | (Black, z, (Red, y, (Red, x, a, b), c), d)
  | (Black, z, (Red, x, a, (Red, y, b, c)), d)
  | (Black, x, a, (Red, z, (Red, y, b, c), d))
  | (Black, x, a, (Red, y, b, (Red, z, c, d))) ->
        (Red, y, (Black, x, a, b), (Black, z, c, d))
  | x -> x
```

which is of type `RBTree(α)→RBTree(α)`. In the definition of `RBTree(α)` nothing distinguishes a red-black tree from a vanilla binary tree with some red or black tags. If we want to enforce some of the invariants of red-black trees (cf. Footnote 6) we must modify the type definition as follows

```
type RBTree(α) = BTree(α) | RTree(α)
type  BTree(α) = ( Black, α, RBTree(α), RBTree(α)) | Leaf
type  RTree(α) = ( Red, α, BTree(α), BTree(α))
```

However, with these definitions the `insert` function of binary trees no longer type-checks. But it is just the matter of giving a precise type to `balance`, since it suffices to add the following type annotation:

```
(Unbalanced(α) → RTree(α)) & (β\Unbalanced(α) → β\Unbalanced(α))
```

where

```
type  WrongTree(α) = (Red, α, RTree(α), BTree(α))
                   | (Red, α, BTree(α), RTree(α))

type Unbalanced(α) = (Black, α, WrongTree(α), RBTree(α))
                   | (Black, α, RBTree(α), WrongTree(α))
```

The two type definitions state that a wrong tree is a red tree with a black child and that an unbalanced tree is a black tree with a wrong child. The annotation describes the semantics of `balance`: it transforms an unbalanced tree into a red tree and leaves any other argument unchanged. We recognize in this type the pattern of our simpler example (the same pattern appears also in some other parts of the red-black tree implementation). It is then possible to deduce for the insertion function for red-black trees the type `BTree(α)` → α →`BTree(α)`. For the valid CDuce code with an explanation of subtler typing details, the reader can refer to Appendix A of [11].

[6] Specifically, that the root of the tree is black, that the leaves of the tree are black, and that no red node has a red child; the missing invariant is that every path from the root to a leaf should contain the same number of black nodes.

Recap. In this section we have seen several examples of use of set-theoretic types. In the next sections we outline their theory. In particular, in Section 3 we formally define set-theoretic types, their subtyping relation, as well as their extension with type variables for parametric polymorphism. Section 4 presents various languages that use these types: we start with a theoretical language that covers all the examples we used in this presentation so far and then describe three practical systems that partially implement this theoretical language, each implementation being the result of a certain number of choices and trade-offs that we will discuss.

3 Types

We have seen in the previous section that set-theoretic types play a key role in typing several language constructs—from branching and pattern matching to function overloading—very precisely. However, we have glossed over exactly how a type checker should treat them. It is essential to define a suitable notion of *subtyping* on these types. The informal description we have given suggests that certain properties should hold. In particular, we expect union and intersection types to satisfy commutative and distributive properties of Boolean algebras. Moreover, we expect, for example,

$$(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}) \leq (\texttt{Int} \vee \texttt{Bool}) \to (\texttt{Int} \vee \texttt{Bool})$$

to hold, so that the typing of functions with type-cases works as we sketched. To model occurrence typing, we want $(\texttt{Int} \vee \texttt{Bool}) \wedge \texttt{Int}$ to be equivalent to $\texttt{Int}$ and $(\texttt{Int} \vee \texttt{Bool}) \wedge \neg\texttt{Int}$ to be equivalent to $\texttt{Bool}$.

Arguably, it is intuitive to view types and subtyping in terms of sets and set inclusion, especially to describe set-theoretic types.[7] We can see a type as the set of the values of that type in the language we consider. Then, we expect t_1 to be a subtype of t_2 if every value of type t_1 is also of type t_2, that is, if the set of values denoted by t_1 is included in the set denoted by t_2. In this view, union and intersection types correspond naturally to union and intersections of sets; negation corresponds to complementation with respect to the set of all values.

However, most systems reason on subtyping using rules that are sound but not complete with respect to this model: that is, they do not allow $t_1 \leq t_2$ in some cases in which every value of type t_1 is in fact a value of type t_2. Incompleteness is not necessarily a problem, but it can result in unintuitive behavior. We show two examples below.

LACK OF DISTRIBUTIVITY: Consider this code in Flow.[8]

```
type A = { a: number }
type B = { kind: "b", b: number }
type C = { kind: "c", c: number }

type T = (A & B) | (A & C)
function f(x: T) { return (x.kind === "b") ? x.b : x.c }
```

[7] For instance, this model is used to explain subtyping in the online documentation of Flow at https://flow.org/en/docs/lang/subtypes.

[8] Adapted from the StackOverflow question at https://stackoverflow.com/questions/44635326.

The first three lines declare three object types; in B and C, "b" and "c" are the singleton types of the corresponding strings. The type T is defined as the union of two intersection types, namely, A&B (the type of objects with a fields a and b of type number and a field kind of type "b") and A&C (the type of objects with a fields a and c of type number and a field kind of type "c").

The function f is well typed: as in handleResponse before, occurrence typing recognizes that x is of type A & B in the branch x.b and of type A & C in the branch x.c. However, if we replace the definition of T to be type T = A & (B | C), the code is rejected by the type checker of Flow. Occurrence typing does not work because T is no longer explicitly a union type. Flow considers (A & B) | (A & C) to be a subtype of A & (B | C): indeed, this can be proven just by assuming that unions and intersections are respectively joins and meets for subtyping. But subtyping does not hold in the other direction, because Flow does not consider distributivity.

UNION AND PRODUCT TYPES: Apart from distributivity laws, we could also expect interaction between union and intersection types and various type constructors. Consider product types; we might expect the two types $(t_1 \times t) \vee (t_2 \times t)$ and $(t_1 \vee t_2) \times t$ to be equivalent (i.e., each one subtype of the other one): intuitively, both of them describe the pairs whose first component is either in t_1 or in t_2 and whose second component is in t. But this reasoning is not always reflected in the behavior of type checkers.

For example, consider this code in Typed Racket (similar examples can be written in Flow or TypeScript).

```
(define-type U-of-Pair (U (Pair Integer Boolean) (Pair String Boolean)))
(define-type Pair-of-U (Pair (U Integer String) Boolean))

(define f (lambda ([x : U-of-Pair]) x))
(define x (ann (cons 3 #f) Pair-of-U))
(f x)
```

We define two type abbreviations. In Typed Racket, U denotes a union type and Pair a product type, so U-of-Pair is $(\mathsf{Integer} \times \mathsf{Boolean}) \vee (\mathsf{String} \times \mathsf{Boolean})$, and Pair-of-U is $(\mathsf{Integer} \vee \mathsf{String}) \times \mathsf{Boolean}$. The two types are not considered equivalent. To show it, we define a function f whose domain is U-of-Pair (for simplicity, we take the identity function) and try to apply it to an argument x of type Pair-of-U; to define x, we use an explicit type annotation (ann) to mark the pair (cons 3 #f) as having type Pair-of-U. The application is rejected. If we exchange the two type annotations, instead, it is accepted: the type checker considers U-of-Pair a subtype of Pair-of-U, but not the reverse.

3.1 Semantic subtyping

In a nutshell we have to define the subtyping relation so that the types satisfy all the commutative and distributive laws we expect from their set-theoretic interpretation. But a "syntactic" definition of subtyping—i.e., a definition given by a set of deduction rules—is hard to devise since, as shown by the previous examples, it may yield a definition that is sound but not complete. To obviate this problem we follow the *semantic subtyping* approach [23, 24]. In this approach subtyping is defined by giving an

interpretation $[\![\cdot]\!]$ of types as sets and defining $t_1 \leq t_2$ as the inclusion of the interpretations, that is, $t_1 \leq t_2$ is defined as $[\![t_1]\!] \subseteq [\![t_2]\!]$. Intuitively, we can see $[\![t]\!]$ as the set of values that inhabit t in the language. By interpreting union, intersection, and negation as the corresponding operations on sets and by giving appropriate interpretations to the other constructors, we ensure that subtyping will satisfy all expected commutative and distributive laws.

Formally, we proceed as follows. We first fix two countable sets: a set $\mathscr{C}$ of *language constants* (ranged over by c) and a set $\mathscr{B}$ of *basic types* (ranged over by b). For example, we can take constants to be Booleans and integers: $\mathscr{C} = \{\mathsf{true}, \mathsf{false}, 0, 1, \text{-}1, \dots\}$. $\mathscr{B}$ might then contain `Bool` and `Int`; however, we also assume that, for every constant c, there is a "singleton" basic type which corresponds to that constant alone (for example, a type for `true`, which will be a subtype of `Bool`). We assume that a function $\mathbb{B} : \mathscr{B} \to \mathscr{P}(\mathscr{C})$ assigns to each basic type the set of constants of that type and that a function $b_{(\cdot)} : \mathscr{C} \to \mathscr{B}$ assigns to each constant c a basic type b_c such that $\mathbb{B}(b_c) = \{c\}$.

Definition 1 (Types). *The set $\mathscr{T}$ of* types *is the set of terms t coinductively produced by the following grammar*

$$t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

and which satisfy two additional constraints: (1) regularity*: the term must have a finite number of different sub-terms;* (2) contractivity*: every infinite branch must contain an infinite number of occurrences of the product or arrow type constructors.*

We use the abbreviations $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \backslash t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2)$, and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$ (in particular, $\mathbb{1}$ corresponds to the type `Any` we used in the examples of Section 2). We refer to b, $\times$, and $\to$ as *type constructors*, and to $\vee$, $\neg$, $\wedge$, and $\backslash$ as *type connectives*. As customary, connectives have priority over constructors and negation has the highest priority—e.g., $\neg s \vee t \to u \wedge v$ denotes $((\neg s) \vee t) \to (u \wedge v)$.

Coinduction accounts for recursive types and it is coupled with a contractivity condition which excludes infinite terms that do not have a meaningful interpretation as types or sets of values: for instance, the trees satisfying the equations $t = t \vee t$ (which gives no information on which values are in it) or $t = \neg t$ (which cannot represent any set of values). Contractivity also gives an induction principle on $\mathscr{T}$ that allows us to apply the induction hypothesis below type connectives (union and negation), but not below type constructors (product and arrow). As a consequence of contractivity, types cannot contain infinite unions or intersections. The regularity condition is necessary only to ensure the decidability of the subtyping relation.

In the semantic subtyping approach we give an interpretation of types as sets; this interpretation is used to define the subtyping relation in terms of set containment. We want to see a type as the set of the values that have that type in a given language. However, this set of values cannot be used directly to define the interpretation, because of a problem of circularity. Indeed, in a higher-order language, values include well-typed λ-abstractions; hence to know which values inhabit a type we need to have already defined the type system (to type λ-abstractions), which depends on the subtyping relation, which in turn depends on the interpretation of types. To break this circularity, types are actually interpreted as subsets of a set $\mathscr{D}$, an *interpretation domain*, which is not the set of values, though it corresponds to it intuitively (in [24], a correspondence is also shown formally: we return to this point in Section 4.2.1). We use the following domain.

Definition 2 (Interpretation domain). *The* interpretation domain $\mathscr{D}$ *is the set of finite terms* d *produced inductively by the following grammar*

$$d ::= c \mid (d,d) \mid \{(d,\partial),\ldots,(d,\partial)\} \qquad\qquad \partial ::= d \mid \Omega$$

where c *ranges over the set* $\mathscr{C}$ *of constants and where* Ω *is such that* $\Omega \notin \mathscr{D}$.

The elements of $\mathscr{D}$ correspond, intuitively, to the results of the evaluation of expressions. These can be constants or pairs of results, so we include both in $\mathscr{D}$. Also, in a higher-order language, the result of a computation can be a function which are represented in this model by finite relations of the form $\{(d_1,\partial_1),\ldots,(d_n,\partial_n)\}$, where Ω (which is not in $\mathscr{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input.

The restriction to *finite* relations is standard in semantic subtyping and it is one of its subtler aspects (see [7] for a detailed explanation of this aspect). In principle, given some mathematical domain $\mathscr{D}$, we would like to interpret $t_1 \to t_2$ as the set of functions from $[\![t_1]\!]$ to $[\![t_2]\!]$. For instance if we consider functions as binary relations, then $[\![t_1 \to t_2]\!]$ could be the set $\{\ f \subseteq \mathscr{D}^2 \mid \text{for all } (d_1,d_2)\in f,\ \text{if } d_1\in[\![t_1]\!] \text{ then } d_2\in[\![t_2]\!]\ \}$ or, compactly, $\mathscr{P}(\overline{[\![t_1]\!]\times\overline{[\![t_2]\!]}})$, where the $\overline{S}$ denotes the complement of the set S within the appropriate universe (in words, these are the sets of pairs in which it is *not* true that the first projection belongs to $[\![t_1]\!]$ and the second does not belong to $[\![t_2]\!]$). But here the problem is not circularity but cardinality, since this would require $\mathscr{D}$ to contain $\mathscr{P}(\mathscr{D}^2)$, which is impossible. The solution given by [24] relies on the observation that in order to use types in a programming language we do not need to know what types are, but just how they are related (by subtyping). In other terms, we do not require the interpretation of an arrow type to be *exactly* the set of all functions of that type. We just require that this interpretation induces the same subtyping relation as interpreting an arrow type with this set would yield. That is, the interpretation must satisfy the (weaker) property

$$[\![s_1\to s_2]\!] \subseteq [\![t_1\to t_2]\!] \iff \mathscr{P}(\overline{[\![s_1]\!]\times\overline{[\![s_2]\!]}}) \subseteq \mathscr{P}(\overline{[\![t_1]\!]\times\overline{[\![t_2]\!]}}).$$

If we interpret $t_1\to t_2$ as the set $\mathscr{P}_{\text{fin}}(\overline{[\![t_1]\!]\times\overline{[\![t_2]\!]}})$ (where $\mathscr{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets), then this property holds.

The above explains why we use a domain $\mathscr{D}$ with finite relations and define the interpretation $[\![t]\!]$ of a type t so that it satisfies the following equalities, where $\mathscr{D}_\Omega = \mathscr{D}\cup\{\Omega\}$:

$$\begin{aligned}
[\![t_1 \vee t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![\neg t]\!] = \mathscr{D}\backslash[\![t]\!] \qquad [\![\mathbb{0}]\!] = \emptyset \\
[\![b]\!] &= \mathbb{B}(b) \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \\
[\![t_1 \to t_2]\!] &= \{R \in \mathscr{P}_{\text{fin}}(\mathscr{D}\times\mathscr{D}_\Omega) \mid \forall (d,\partial)\in R.\, d\in[\![t_1]\!] \implies \partial\in[\![t_2]\!]\}
\end{aligned}$$

This interpretation is reminiscent of a common practice in denotational semantics that consists in interpreting functions as the set of their finite approximations: we will discuss this relation more in Section 5.3. A consequence of this interpretation is that the type $\mathbb{0}\to\mathbb{1}$ contains all the (well-typed) functions: it will play an important role in Section 4. The interpretation also explains the need of the constant Ω: this constant is used to ensure that $\mathbb{1}\to\mathbb{1}$ is not a supertype of all function types: in a domain without Ω (i.e., where the last of the equalities above would use d instead of ∂) every well-typed

function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently from the types of the function and of its argument; thanks to Ω instead $\mathbb{1} \to \mathbb{1}$ contains only the functions whose domain is exactly $\mathbb{1}$, since a function with domain, say, `Int`, could map non-integer elements to Ω, thus excluding it from $\mathbb{1} \to \mathbb{1}$ (since $\Omega \not\leq \mathbb{1}$): see Section 4.2 of [24] for details.

We cannot take the equations above directly as an inductive definition of $[\![\cdot]\!]$ because types are not defined inductively but coinductively. Therefore we give the following definition, which validates these equalities and which uses the aforementioned induction principle on types and structural induction on $\mathscr{D}$.

Definition 3 (Set-theoretic interpretation of types). *We define a binary predicate $(\partial : t)$ ("the element ∂ belongs to the type t"), where $\partial \in \mathscr{D} \cup \{\Omega\}$ and $t \in \mathscr{T}$, by induction on the pair (∂, t) ordered lexicographically. The predicate is defined as:*

$$\begin{aligned}
(c : b) &= c \in \mathbb{B}(b) \\
((d_1, d_2) : t_1 \times t_2) &= (d_1 : t_1) \text{ and } (d_2 : t_2) \\
(\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\} : t_1 \to t_2) &= \forall i \in \{1, \ldots, n\}. \text{ if } (d_i : t_1) \text{ then } (\partial_i : t_2) \\
(d : t_1 \vee t_2) &= (d : t_1) \text{ or } (d : t_2) \\
(d : \neg t) &= \text{not}\,(d : t) \\
(\partial : t) &= \text{false} \qquad \textit{otherwise}
\end{aligned}$$

We define the set-theoretic interpretation $[\![\cdot]\!] : \mathscr{T} \to \mathscr{P}(\mathscr{D})$ *as* $[\![t]\!] = \{d \in \mathscr{D} \mid (d : t)\}$.

Notice that $\Omega \notin [\![t]\!]$, for any type t. Finally, we define the subtyping preorder and its associated equivalence relation as:

Definition 4 (Subtyping). *We define the* subtyping *relation* $\leq$ *and the* subtyping equivalence *relation* $\simeq$ *as* $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ *and* $t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leq t_2)$ and $(t_2 \leq t_1)$.

The subtyping relation is decidable. Deciding whether t_1 is a subtype of t_2 is equivalent to deciding whether $t_1 \wedge \neg t_2$ is the empty type, insofar as $t_1 \leq t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \cap (\mathscr{D} \setminus [\![t_2]\!]) \subseteq \varnothing \iff [\![t_1 \wedge \neg t_2]\!] \subseteq \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$. A detailed description of the subtyping algorithm and of the data structures used to implement it efficiently can be found in [6].

3.2 Polymorphic Extension

The examples we gave at the beginning of this article used polymorphic types. Syntactically, this means extending the grammar of types with type variables drawn from a countable set $\mathscr{V}$ ranged over by α:

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

However, extending the semantic subtyping approach to define a subtyping relation on these types is not straightforward and has been a longstanding open problem. The reason is explained by Hosoya et al. [37] who point out that the naive solution of defining subtyping of two polymorphic types as equivalent to the subtyping of all their ground instances yields a subtyping relation that is both untreatable and counterintuitive. They

demonstrate this by defining the following problematic example:

$$t \times \alpha \leq (t \times \neg t) \vee (\alpha \times t)$$

One could expect this judgment not to hold, because the type variable α appears in unrelated positions in the two types (in the second component on the left of a product, in the first one on the right). According to the naive definition, instead, the judgment holds if and only if t is a singleton type.

The solution to this problem was found by Castagna and Xu [9] who argue that one should consider only interpretations of types where judgments such as the above do not hold. This should ensure that subtyping on type variables behaves closer to the expectations for parametric polymorphism, so that a type variable can occur on the right-hand side of a subtyping judgment only if it occurs in a corresponding position on the left-hand side. To that end Castagna and Xu [9] propose *convexity* as a general property of interpretations that avoid pathological behavior such as the example above. We leave the interested reader to refer to [9] for details on the convexity property and its interpretation. Here we present a very simple way to extend the interpretation of Definition 3 into a convex interpretation for polymorphic types. The idea, due to Gesbert et al. [26, 27], is to consider the domain $\mathscr{D}$ of Definition 2 in which all elements are labeled by a finite set of type variables

$$d ::= c^L \mid (d,d)^L \mid \{(d,\partial),\ldots,(d,\partial)\}^L \qquad\qquad \partial ::= d \mid \Omega$$

with $L \in \mathscr{P}_{\text{fin}}(\mathscr{V})$, and interpret a type variable α by the set of all elements that are labeled by α, that is $[\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\}$ (where $\mathsf{tags}(c^L) = \mathsf{tags}((d,d')^L) = \mathsf{tags}(\{(d_1,\partial_1),\ldots,(d_n,\partial_n)\}^L) = L$). The interpretation of all other types disregards labels (e.g., the interpretation of `Int` is the set of all integer constants labeled by any set of variables). It is straightforward to modify Definition 3 to validate the equality $[\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\}$: it suffices to use the new domain and just add the clause

$$(d : \alpha) \quad = \quad \alpha \in \mathsf{tags}(d)$$

No further modification is needed (apart from adding labels in the first clauses) and Definition 4 is still valid.[9]

While the interpretation of type variables is not very intuitive, it is easy to check that it yields a subtyping relation that has all the sought properties. First and foremost, according to this interpretation a type is empty if and only if all its instances are empty. In particular, as expected, the interpretation of a type variable α is never empty (it contains all the elements tagged by α) insofar as α could be instantiated into a non-empty type. Also, the interpretation of a variable is contained in the interpretation of another variable if and only if the two variables are the same.[10]

[9] The reason why the interpretation thus obtained is convex is that every type is interpreted into an infinite set (even singletons types, since, e.g., $[\![\texttt{true}]\!] = \{\mathsf{true}^L \mid L \in \mathscr{P}_{\text{fin}}(\mathscr{V})\}$). See Castagna and Xu [9] to see how this implies convexity.

[10] It is important to avoid confusion between (sub)type equivalence and unification. There is a fundamental difference between being the same type (i.e., denoting the same set of values) and to be unifiable: two variables can always be unified, but if they are not the same, then it is not safe to use an expression whose type is one variable where an expression whose type is a different variables is expected. For instance, `fun(x:`α`→`α`, y:`β`)=xy` is not well typed (since

Finally, $\alpha \wedge t$ is empty if and only if t is empty since, otherwise, we could obtain a non-empty type by instantiating α with t. For instance, since $[\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\}$ and $[\![\mathtt{Int}]\!] = \{n^L \mid n \in \mathbb{Z}\}$, then $[\![\alpha \wedge \mathtt{Int}]\!] = \{n^L \mid n \in \mathbb{Z}, \alpha \in L\}$: we see that $\alpha \wedge \mathtt{Int}$ is not empty since it contains at least $\mathtt{42}^{\{\alpha\}}$. Likewise, since $[\![\alpha \wedge \mathtt{Int}]\!]$ contains both $\mathtt{42}^{\{\alpha\}}$ and $\mathtt{42}^{\{\alpha,\beta\}}$, then neither $\alpha \wedge \mathtt{Int} \leq \beta$ nor $\alpha \wedge \mathtt{Int} \leq \neg\beta$ hold, the former because $\mathtt{42}^{\{\alpha\}} \notin [\![\beta]\!]$ the latter because $\mathtt{42}^{\{\alpha,\beta\}} \notin [\![\neg\beta]\!]$ (by definition $[\![\neg\beta]\!] = \{d \mid \beta \notin \mathsf{tags}(d)\}$). The subtyping relation is again decidable (see [9] for a detailed description of the algorithm) and, although it is not evident from the interpretation, the subtyping relation is preserved by type substitutions, a property needed to ensure soundness for polymorphic type systems.

4 Languages

The natural candidate languages for the types we presented in the previous section are λ-calculi (functional languages) with at least pairs and type-cases. Intuitively, we need a λ-calculus because we have arrow types, we need pairs to inhabit product types, while type-cases are needed to define overloaded functions and thus inhabit any intersection of arrow types. We will see in Section 4.2 (cf. Section 4.2.1 in particular) that the correspondence between set-theoretic types and a language satisfying these criteria can be formally stated.

We start by describing a generic theoretical language that covers all the features we outlined in the motivation section. While theoretically interesting the language will not be effective: it is a language so generic and expressive that defining a reasonably complete type-inference algorithm seems very hard. We will then discuss some trade-offs and define three effective (sub-)systems for which type inference is possible, but each of which will be able to capture only a part of the examples of Section 2.

4.1 Theoretical framework

The expressions and values of our theoretical language are defined as follows:

$$\begin{array}{ll} \textbf{Expressions} & e ::= c \mid x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi_i e \mid (e\in t)\,?\,e:e \\ \textbf{Values} & v ::= c \mid \lambda x.e \mid (v,v) \end{array} \tag{2}$$

Expressions are an untyped λ-calculus with constants c, pairs (e,e), pair projections $\pi_i e$, and type-cases $(e\in t)\,?\,e:e$. A type-case $(e_0\in t)\,?\,e_1:e_2$ is a dynamic type test that first evaluates e_0 and, then, if e_0 reduces to a value v evaluates e_1 if v has type t or e_2 otherwise. Formally, the reduction semantics is that of a call-by-value pure λ-calculus with pairs and type-cases. The reduction is given by the following notions of reductions

the programmer wrote these variables, then they are considered monomorphic and it is unsound to use a β expression where an α expression is expected) while `fun(x:`$\alpha\to\alpha$`, y:`α`)=xy` is well typed, and so is `fun(x,y)=xy`. To type the latter the type system assigns the type α to `x` and β to `y` and then *unifies* α with $\beta\to\gamma$ which yields the type $((\beta\to\gamma)\times\beta)\to\gamma$ of the function.

(where $e\{v/x\}$ denotes the capture avoiding substitution of v for x in e)

$$(\lambda x.e)v \rightsquigarrow e\{v/x\} \qquad (v\in t)\,?\,e_1:e_2 \rightsquigarrow e_1 \quad \text{if } v\in t$$
$$\pi_1(v_1,v_2) \rightsquigarrow v_1 \qquad (v\in t)\,?\,e_1:e_2 \rightsquigarrow e_2 \quad \text{if } v\notin t$$
$$\pi_2(v_1,v_2) \rightsquigarrow v_2$$

together with the context rules that implement a leftmost outermost reduction strategy, that is, $E[e] \rightsquigarrow E[e']$ if $e \rightsquigarrow e'$ where the evaluation contexts $E[\cdot]$ are defined as $E ::= [\,] \mid vE \mid Ee \mid (v,E) \mid (E,e) \mid \pi_i E \mid (E\in t)\,?\,e:e$. In the reduction rules we used the notation $v \in t$ to indicate that the value v has type t. Here, this corresponds to deducing the judgment $\varnothing \vdash v : t$ using the rules given in Figure 1, rules that form the type-system of our language; but we will see that for the three system variations we present later on, the relation $v \in t$ can be defined without resorting to the type-system: this is an important property since we do not want to call the type-inference algorithm to decide at run-time the branching of a type-case.

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

$$[\to\mathrm{I}]\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad [\to\mathrm{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$[\times\mathrm{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1,e_2) : t_1 \times t_2} \qquad [\times\mathrm{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\times\mathrm{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \quad \Gamma, x : t_1 \vdash e : t \quad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t} \qquad [\leq]\ \frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\mathbb{0}]\ \frac{\Gamma \vdash e : \mathbb{0}}{\Gamma \vdash (e\in t)\,?\,e_1 : e_2 : \mathbb{0}} \qquad [\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e\in t)\,?\,e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e\in t)\,?\,e_1 : e_2 : t_2}$$

Fig. 1: Declarative type system

The rules in the first three rows of Figure 1 do not deserve any special comment: they are the standard typing rules for a simply-typed λ-calculus with pairs where, as customary, Γ ranges over type environments, that is, finite mappings from variables to types, $\Gamma, x : t$ denotes the extension of the environment Γ with the mapping $x \mapsto t$ provided that $x \notin \mathsf{dom}(\Gamma)$ (we will use $\varnothing$ to denote the empty type environment).

The rules in the fourth row are also standard. The first rule [$\wedge$] is the classic introduction rule for intersection: it states that if an expression has two types, then it has their intersection, too. The second rule [$\vee$] is the classic union elimination rule as it was first introduced by MacQueen et al. [41]: it states that given some expression (here, $e\{e'/x\}$) with a subexpression e' of type $t_1 \vee t_2$, if we can give to this expression the type t both under the hypothesis that e' produces a result of type t_1 and under the hypothesis that e' produces a result in t_2, then we can safely give this expression type t. The last rule of the row is the *subsumption* rule [$\leq$] that states that if an expression has some type t, then it has all super-types of t, too. Together, the rules in the first four lines of

Figure 1 form a well-known type-system, since they are the same rules as those in Definition 3.5 of the classic work on union and intersection types by Barbanera, Dezani, and de'Liguoro [1]. Although the rules are textually the same as in [1], there is an important difference between the system in [1] and the one in Figure 1, namely, that our types are a strict extension of those of [1] since we also have recursive types, negation types, and the empty type. As a consequence our subsumption rule uses the subtyping relation of Definition 4 which is more general than the one in [1] of which it is a conservative extension (cf. [19]).

Finally, the last three rules are specific to systems with set-theoretic types and type-cases. They are rather new (they were first introduced in [16]) and provide a natural and nifty way to type type-case expressions. The first rule states that if the tested expression e has the empty type (i.e., if e diverges, that is, it can only produce a value that is in the empty set), then so has the whole type-case expression. The second rule states that if e can only produce a result in t, then the type of $(e\in t)\,?\,e_1:e_2$ is the type of e_1. The third rule states that if e can only produce a result in $\neg t$, then the type of $(e\in t)\,?\,e_1:e_2$ is the type of e_2: since the negation type $\neg t$ is interpreted set-theoretically as the set of all values that are *not* of type t, this means that, in that case, e can only produce a result *not* of type t. The reader may wonder how we type a type-case expression $(e\in t)\,?\,e_1:e_2$ when the tested expression e is neither of type t nor of type $\neg t$. As a matter of fact, a type-case is interesting only if we cannot statically determine whether it will succeed or fail. For instance, when discussing occurrence typing, we informally described how to type the expression $(x\in\mathtt{Int})\,?\,(x+1):\neg x$ when x is of type $\mathtt{Int}\vee\mathtt{Bool}$, that is, in that case, when x is neither of type $\mathtt{Int}$ nor of type $\neg\mathtt{Int}$. Here is where the union elimination rule [$\vee$] shows its full potential. Even though the expression e tested in $(e\in t)\,?\,e_1:e_2$ has some type s that is neither contained in (i.e., subtype of) t nor in $\neg t$, we can use intersection and negation to split s into the union of two types that have this property, since $s\simeq(s\wedge t)\vee(s\wedge\neg t)$. We can thus apply the union rule and check the type-case under the hypothesis that the tested expression has type $(s\wedge t)$ and under the hypothesis that it has type $(s\wedge\neg t)$. For instance, for $(x\in\mathtt{Int})\,?\,(x+1):\neg x$ we check the type-case under the hypothesis that x has type $\mathtt{Int}$ (i.e., $(\mathtt{Int}\vee\mathtt{Bool})\wedge\mathtt{Int}$) and deduce the type $\mathtt{Int}$, and under the hypothesis that x has type $\mathtt{Bool}$ (i.e., $(\mathtt{Int}\vee\mathtt{Bool})\wedge\neg\mathtt{Int}$) and deduce the type $\mathtt{Bool}$ which, by subsumption, gives for the whole expression the type $\mathtt{Int}\vee\mathtt{Bool}$.

A final important remark is that the deduction system in Figure 1 is defined modulo α-conversion. This is crucial in systems with union types since the rule [$\vee$] breaks the α-invariance property (see Section 2.4 in [16] and Discussion 12.5 in [32]).

4.1.1 On deriving negation types. The language and the typing rules we just defined are expressive enough to cover all the examples we described in the first two sections. However, the rules of Figure 1 are yet not enough to cover the whole palette of application of set-theoretic types. The reason is that in the current system the only way to derive for an expression a negation type is to use the subsumption rule. For instance, we can deduce $42:\neg\mathtt{Bool}$ by subsumption, since $42:\mathtt{Int}$ and $\mathtt{Int}\leq\neg\mathtt{Bool}$ (since all integer constants are contained in the set of values that are not Booleans). But

while subsumption is sufficient for values formed only by constants,[11] it is not enough for values with functional components. For example, consider the successor function $\lambda x.(x+1)$. This function has type $\texttt{Int} \to \texttt{Int}$ but *not* type $\texttt{Bool} \to \texttt{Bool}$: it maps integers to integers but when applied to a Boolean it does not return a Boolean. Therefore, one would expect the type system to deduce for the successor function the type $\neg(\texttt{Bool} \to \texttt{Bool})$. However, in this case subsumption is of no use since $\texttt{Int}\to\texttt{Int}$ is *not* a subtype of $\neg(\texttt{Bool} \to \texttt{Bool})$, and rightly so since it is easy to find a value that is of the former type but not of the latter one: for instance, the identity function $\lambda x.x$ is a function that has type $\texttt{Int} \to \texttt{Int}$ but—since it is also of type $\texttt{Bool}\to\texttt{Bool}$—is *not* of type $\neg(\texttt{Bool} \to \texttt{Bool})$.

Intuitively, we would like the type-system to deduce for an expression e the type $\neg t$ whenever (i) e is typable with some type t' and (ii) it is not possible to deduce the type t for it. In a sense we would like to have a rule such as the pseudo-rule $[\neg]$ here below on the left:

$$[\neg]\ \frac{\Gamma \vdash e : t' \qquad \Gamma \not\vdash e : t}{\Gamma \vdash e : \neg t} \qquad\qquad [\neg^{(\to)}]\ \frac{\Gamma \vdash \lambda x.e : t' \qquad \Gamma \not\vdash \lambda x.e : t \to t''}{\Gamma \vdash \lambda x.e : \neg(t \to t'')}$$

This pseudo-rule, which puts in formulas what we explained in prose, deduces negation types for a generic expression e. However, from a practical perspective a less generic rule such as $[\neg^{(\to)}]$ above on the right would suffice: as a matter of fact, deciding negation types is useful in practice to evaluate type-cases and these are decided on values rather than generic expressions. So from a practical viewpoint it suffices to deduce negation types only for values rather than for all expressions and, in particular, for λ-abstractions, since for all the other values subsumption is enough (see Footnote 11). So instead of deducing generic negation types for generic expressions, it is enough to deduce negated arrow types for λ-abstractions, yielding the less general pseudo-rule $[\neg^{(\to)}]$ above.

A different motivation for deducing negation types is that, for the reasons we explain in Section 4.1.3, few practical systems implement the union elimination rule $[\vee]$ in its full generality, insofar as deciding when $[\vee]$ is to be applied is still an open problem (technically, this corresponds to determining an inversion lemma for the $[\vee]$ rule). Now, in the absence of a $[\vee]$ rule (e.g., in the systems in Section 4.2 and 4.3), the property of type preservation by reduction (also known as the property of *subject reduction*) requires the following property to hold:

For every type t and well-typed value v, either $\varnothing \vdash v : t$ or $\varnothing \vdash v : \neg t$ holds. (3)

[11] These are either constants or possibly nested pairs of constants. All these values have a smallest type deduced by the rules of Figure 1 and this smallest type is *indivisible* (i.e., its only proper subtype is the empty type: cf. [9]). An indivisible type acts like a point in the set-theory of types: it is either contained in a type or in its negation, and so are their values. This is the reason why subsumption suffices to determine the negation types of this sort of values: they are all the types that do not contain the minimal type of the value at issue. Functions, in general, do not have a smallest type.

To illustrate why this is required, consider the expression $\lambda x.(x,x)$ and the following typing derivation (for some arbitrary type t).

$$[\leq]\ \dfrac{[\wedge]\ \dfrac{\dfrac{\vdots}{\varnothing \vdash \lambda x.(x,x) : t \to (t \times t)} \quad \dfrac{\vdots}{\varnothing \vdash \lambda x.(x,x) : \neg t \to (\neg t \times \neg t)}}{\varnothing \vdash \lambda x.(x,x) : (t \to (t \times t)) \wedge (\neg t \to (\neg t \times \neg t))}}{\varnothing \vdash \lambda x.(x,x) : \mathbb{1} \to ((t \times t) \vee (\neg t \times \neg t))}$$

The subsumption rule can be applied because

$$(t \to (t \times t)) \wedge (\neg t \to (\neg t \times \neg t)) \leq \mathbb{1} \to ((t \times t) \vee (\neg t \times \neg t)) :$$

in general, it holds that $(t'_1 \to t_1) \wedge (t'_2 \to t_2) \leq (t'_1 \vee t'_2) \to (t_1 \vee t_2)$, and $t \vee \neg t \simeq \mathbb{1}$. Now consider an arbitrary type t and a well-typed value v. Since v has type $\mathbb{1}$ by subsumption, the application $(\lambda x.(x,x))v$ can be typed as $(t \times t) \vee (\neg t \times \neg t)$. This application reduces to (v,v). Therefore, either (v,v) has type $(t \times t) \vee (\neg t \times \neg t)$ or subject reduction does not hold. Since $t \times t$ and $\neg t \times \neg t$ are disjoint, to derive the union type for v we need either the system to have the [$\vee$] rule, or v to have either type t or type $\neg t$. This illustrates the need for the property above which in particular requires to be able to derive negation types for functions other than by subsumption: e.g., since we cannot derive for $\lambda x.x$ the type $\texttt{Int} \to \texttt{Bool}$, then we must be able to derive for it the type $\neg(\texttt{Int} \to \texttt{Bool})$.

However, both [$\neg$] and [$\neg^{(\to)}$], untamed, do not make much sense (which is why we called them pseudo-rules). First, their definitions are circular since they depend on the very relation they are defining. Furthermore, they cannot be used in a deduction system since they would correspond to non-monotone immediate-consequence operators for which a fix-point may not exist and thus cannot be used to define the typing relation by induction. Therefore, it is necessary to put some tight-knit restrictions on the inference of negation types. This requires a lot of care because the very presence of negation types may yield to paradoxes, as it can be evinced from considering the recursive function[12] $\texttt{let rec}\ f = \lambda x.(f \in \texttt{true} \to \texttt{true})\,\texttt{?false:true}$; it is easy to see that f maps $\texttt{true}$ to $\texttt{true}$ if and only if it does not have type $\texttt{true} \to \texttt{true}$.

As hard the inference of negation types is, it cannot be dismissed lightheartedly, since the definition of the relation $v \in t$ depends on it and so does the semantics of type-cases: if we perform a type test such as $v \in \neg(\texttt{Bool} \to \texttt{Bool})$, then we expect it to succeed at least for some functional values (e.g., the successor function). In the second part of this section we will show different solutions proposed in the literature to infer negation types in a controlled way.

4.1.2 On the feasibility of type-inference. Defining inference of negation types is not the only problem to be solved before obtaining a language usable in practice. The

[12] Thanks to recursive types it is easy to define a polymorphic fix-point combinator and thus define recursive functions: for every type t it is possible to define Curry's fix-point combinator $\mathbf{Z}^t : (t \to t) \to t$ as $\lambda f{:}t \to t.\Delta^t\Delta^t$ where $\Delta^t = \lambda x{:}\mu X.X \to t.f(xx)$. Since our calculus is strict, it is more interesting to define, for any type s and t the strict fix-point combinator $\mathbf{Z}^{s,t} : ((s \to t) \to s \to t) \to s \to t$ as $\lambda f{:}(s \to t) \to\ s \to t.\mathrm{E}^{s,t}\mathrm{E}^{s,t}$ where $\mathrm{E}^{s,t} = \lambda x{:}\mu X.X \to s \to t.f(\lambda v{:}s.xxv)$.

rules of Figure 1 are still a far cry from a practical system that can decide whether a program is well-typed or not. As customary, there are essentially two problems:

1. the rules are not *syntax directed*: given a term, to type it we can try to apply some elimination/introduction rule, but also to apply the intersection rule [∧], or the subsumption rule [≤], or the union rule [∨].
2. some rules are *non-analytic*:[13] if we use the [→I] rule to type some λ-abstraction we do not know how to determine the type t_1 in the premise; if we use the [∨] rule we know neither how to determine e' nor how to determine the types t_1 and t_2 that split the type of e'.

Notice that [∨] cumulates both problems. The problem that some rules are not syntax directed can be already solved in this system for at least two of the three rules at issue: for the rules [∧] and [≤] it is possible to eliminate them and refactor the use of intersections and subtyping in the remaining rules. This essentially amounts to resorting to some form of canonical derivations in which intersection [∧] and subsumption [≤] rules are used at specific places: it can be proved (cf. [16]) that a typing judgment is provable with the system of Figure 1 if and only if there exists a derivation for that typing judgment where (i) subsumption is only used on the left premise of an application or a type-case rule, on the right premises of the union rule, and on the premise of a projection rule and (ii) intersection is only used for expressions that are λ-abstractions, that is, all the premises of an intersection rule are the consequence of a [→I]. This yields an equivalent system formed by the rules in Figure 2, plus the rules [Const],

$$[\to\mathrm{I}^{(\wedge)}]\ \frac{\forall i\in I\quad \Gamma,x:s_i\vdash_{\mathrm{can}} e:t_i}{\Gamma\vdash_{\mathrm{can}} \lambda x.e:\bigwedge_{i\in I} s_i\to t_i}\qquad [\to\mathrm{E}^{(\leq)}]\ \frac{\Gamma\vdash_{\mathrm{can}} e_1:t\leq t_1\to t_2\quad \Gamma\vdash_{\mathrm{can}} e_2:t_1}{\Gamma\vdash_{\mathrm{can}} e_1e_2:t_2}$$

$$[\times\mathrm{E}_1^{(\leq)}]\ \frac{\Gamma\vdash_{\mathrm{can}} e:t\leq t_1\times t_2}{\Gamma\vdash_{\mathrm{can}} \pi_1 e:t_1}\qquad [\times\mathrm{E}_2^{(\leq)}]\ \frac{\Gamma\vdash_{\mathrm{can}} e:t\leq t_1\times t_2}{\Gamma\vdash_{\mathrm{can}} \pi_2 e:t_2}$$

$$[\vee^{(\leq)}]\ \frac{\Gamma\vdash e':t_1\vee t_2\qquad \Gamma,x:t_1\vdash e:s_1\leq t\qquad \Gamma,x:t_2\vdash e:s_2\leq t}{\Gamma\vdash e\{e'/x\}:t}$$

$$[\in_1^{(\leq)}]\ \frac{\Gamma\vdash_{\mathrm{can}} e:t_0\leq t\quad \Gamma\vdash_{\mathrm{can}} e_1:t_1}{\Gamma\vdash_{\mathrm{can}} (e\in t)\,?\,e_1:e_2:t_1}\qquad [\in_2^{(\leq)}]\ \frac{\Gamma\vdash_{\mathrm{can}} e:t_0\leq\neg t\quad \Gamma\vdash_{\mathrm{can}} e_2:t_2}{\Gamma\vdash_{\mathrm{can}} (e\in t)\,?\,e_1:e_2:t_2}$$

Fig. 2: Canonical typing rules

[Var], [×I], and [𝕆] of Figure 1, which do not change. We improved the situation on the syntax-directed front since we got rid of [∧] and [≤], but it looks as we worsened the non-analytic front since now *all* rules in Figure 2 are non-analytic. In particular, nothing tells us how to determine the larger types in the subtyping relations occurring at the premises of these rules. Actually, for the three elimination rules [...E$^{(\leq)}$] in Figure 2

[13] We consider *non-analytic* (or *synthetic*) a rule in which the input (i.e., Γ and e) of the judgment at the conclusion is not sufficient to determine the inputs of the judgments at the premises (cf. [42, 61]).

there exists a standard way to determine these larger types which resorts to using some type operators defined by Frisch et al. [24]. To understand it, consider the rule [→E] for applications in Figure 1. It essentially does three things: (*i*) it checks that the expression in the function position has a functional type; (*ii*) it checks that the argument is in the domain of the function, and (*iii*) it returns the type of the application. In systems without set-theoretic types these operations are straightforward: (*i*) corresponds to checking that the expression in the function position has an arrow type, (*ii*) corresponds to checking that the argument is in the domain of the arrow deduced for the function, and (*iii*) corresponds to returning the codomain of that arrow. With set-theoretic types things get more complicated, since in general the type of a function is not always a single arrow, but it can be a union of intersections of arrow types and their negations.[14] Checking that the expression in the function position has a functional type is easy since it corresponds to checking that it has a type subtype of $\mathbb{0}\to\mathbb{1}$, the type of all functions. Determining its domain and the type of the application is more complicated and needs the operators $\mathsf{dom}()$ and $\circ$ defined as $\mathsf{dom}(t) \stackrel{\text{def}}{=} \max\{u \mid t \leq u \to \mathbb{1}\}$ and $t \circ s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\}$. In short, $\mathsf{dom}(t)$ is the largest domain of any single arrow that subsumes t while $t \circ s$ is the smallest codomain of any arrow type that subsumes t and has domain s. Thus the non-analytic rule $[\to\mathrm{E}^{(\leq)}]$ in Figure 2 can be replaced by its analytic version $[\to\mathrm{E}^{(\wedge)}]$ below:

$$[\to\mathrm{E}^{(\wedge)}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t_1 \circ t_2}\ \begin{array}{l} t_1 \leq \mathbb{0} \to \mathbb{1} \\ t_2 \leq \mathsf{dom}(t_1) \end{array} \qquad [\times\mathrm{E}_i^{(\wedge)}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \pi_i e : \boldsymbol{\pi_i}(t)}\ \begin{array}{l} t \leq \mathbb{1} \times \mathbb{1} \\ i = 1,2 \end{array}$$

We need similar operators for projections since in the rules $[\times\mathrm{E}_i^{(\leq)}]$ $(i = 1,2)$ the type t of e in $\pi_i e$ may not be a single product type but, say, a union of products: all we know is that for the projection to be well-typed t must be a subtype of $\mathbb{1} \times \mathbb{1}$. So let t be a type such that $t \leq \mathbb{1} \times \mathbb{1}$, we define $\boldsymbol{\pi_1}(t) \stackrel{\text{def}}{=} \min\{u \mid t \leq u \times \mathbb{1}\}$ and $\boldsymbol{\pi_2}(t) \stackrel{\text{def}}{=} \min\{u \mid t \leq \mathbb{1} \times u\}$ and replace each non-analytic rule $[\times\mathrm{E}_i^{(\leq)}]$ with the corresponding analytic version $[\times\mathrm{E}_i^{(\wedge)}]$ above. All these type operators can be effectively computed (see [24]).

4.1.3 Practical systems. Although we showed how to handle some non-analytic and/or non-syntax-directed rules, filling the gap between our theoretical setting and practical languages still requires to solve three non-trivial problems:

1. how to infer the type of λ-abstractions since the rule $[\to \mathrm{I}^{(\wedge)}]$ in Figure 2 not only is non-analytic, but also states that we must be able to deduce for λ-abstractions intersection of arrows rather than just a single arrow;
2. how to deduce negation types for expressions or, at least, how to deduce negated arrow types for λ-abstractions.
3. how to infer the type of type-cases; this in particular implies to tame the union-elimination rule [∨] which, in its present formulation, is still too generic: to use this rule to type some expression e one has to guess which subexpression e' of e to

[14] This is the reason why, having eliminated in Figure 2 the subsumption rule from the system, we need in rule $[\to\mathrm{E}^{(\leq)}]$ to subsume the type t deduced for the function e_1 to an arrow type. Likewise for the type t of the expression e in $[\times\mathrm{E}_i]$ which might be different from a product.

single out, which occurrences of e' in e are to be tested by replacing them by x, and how to split the type of this e' in a union of types to be tested separately.

These three problems are tightly connected: the inference of negation types essentially concerns the typing of functions and it affects the semantics of type-cases; the typing of type-cases must be based on their semantics, and since this typing is essentially performed by the union elimination rule, the taming of this rule cannot thus be disconnected from the semantics of type-cases, ergo, from the deduction of negation types, ergo, from the typing of λ-abstractions.

Unfortunately, for these three problems there is not a one-size-fits-all solution, yet. In the next three subsections we are going to present three different ways to address these problems yielding to three different practical systems, and discuss their advantages and drawbacks. We can summarize these three solutions as follows:

Core CDuce. Everything is explicit: every function is explicitly annotated with its type and the use of union elimination is limited to type-cases and needs to explicitly specify a variable to capture the tested value. In summary, no type-reconstruction, no general occurrence typing, but intersection of arrows and negation types are inferred for the functions.

Type Reconstruction. Functions are implicitly typed, that is, they need no annotation: their type is reconstructed but an intersection type can be deduced only if the function is explicitly annotated with it. No general occurrence typing and type-cases can test functional values only in a limited form. Inference of negation types is not performed: it is only used for proving type soundness.

General Occurrence Typing. Everything can be implicit, full use of union elimination to implement occurrence typing, reconstruction of intersections of arrow types for functions, but no polymorphism or inference of negation types. The typing algorithm is sound but cannot be complete.

4.2 Core CDuce

The first solution for the three problems just described in Section 4.1.3 was given with the definition of the calculus introduced in [23, 24] to study semantic subtyping, calculus which constitutes the functional core of the programming language CDuce [2]. In this Core Calculus of CDuce (from now on, just CDuce for short), the problem of inferring intersection types for λ-abstractions is solved by annotating them with an intersection type. Annotations solve also the problem of inferring negated arrow types for λ-abstractions since, as we detail below, we can deduce any negated arrow for a λ-abstraction as long as the intersection of this type with the type annotating the function is not empty. Finally, for what concerns type-cases and union-elimination, CDuce restricts union elimination to type-cases expressions whose syntax is modified so that they specify the binding of the variable used in the union rule.

Before detailing these technical choices it is important to understand the reason that drove their definition. These choices were made to "close the circle":

4.2.1 Closing the circle.

All this presentation long we spoke of types as sets of values. In particular, we said that semantic subtyping consists in interpreting types as sets

of values and then defining one type to be subtype of another if and only if the interpretation of the one type is contained in the interpretation of the other type. Since a subtyping relation is a pre-order, then it immediately induces the notions of least upper bound and greatest lower bound of a set of types. It is then natural to use such notions—thus, the subtyping relation—to characterize, respectively, union and intersection types. This property was used in the context of XML processing languages by Hosoya, Pierce, and Vouillon [36, 34, 33, 35]: by combining union types with product and recursive types they encoded XML typing systems such as DTDs or XML Schemas. The work of Hosoya et al., however, had an important limitation, since it could not define the subtyping relation for functions types and, therefore, it could not be used to type languages with higher order functions. This impossibility resided in a circularity of the definition we already hinted at in Section 3.1: to define subtyping one needs to define the type of each value; for non functional values this can be done by induction on their structure, but with functional values—i.e., λ-abstractions—this requires to type the bodies of the functions which, in turn, needs the very subtyping relation one is defining.

The solution to this circularity problem was found by Frisch et al. [23, 24] and consisted of three steps: (I) interpret types as sets of elements of some domain $\mathscr{D}$ and use this interpretation to define a subtyping relation; (II) use the subtyping relation just defined to type a specifically tailored functional language and in particular its values; (III) show that if we interpret a type as the set of values of this language that have that type, then this new interpretation induces the same subtyping relation as the starting one (which interprets types into subsets of the domain $\mathscr{D}$).

Step (I) yielded the definition of the interpretation we gave in Definition 3 resulting in the subtyping relation of Definition 4. For step (II) we need to define a language such that its set of values satisfies the property in (III). For that, the language must provide enough values to separate every pair of distinct types. In other terms, whenever two types do not have the same interpretation in the denotational model, then there must exist a value in the language that is in one type but not in the other one. In order to provide enough values to distinguish semantically different types we need three ingredients, two of which are already present in our system: (*i*) the inference of intersection types for functions (*ii*) a type-case expression, and (*iii*) a random choice operator. We detail each of them in the next subsections.

4.2.2 Inferring Intersection Types for Functions. The first problem we encounter is how to deduce intersection types for functions. In particular, for every distinct pair of intersections of arrows, we want to be able to define a function that distinguishes them (an intersection of arrows is never empty since it contains at least the function that diverges on all arguments). This is difficult to do in practice unless functions are explicitly annotated. As a matter of fact, $\lambda x.x$ has type $t \to t$ for every type t, and thus it has all possible finite intersections of these types, thus providing an infinite search space for an intersection type, without a best solution (since we do not have infinite intersections). To address this problem one could be tempted to annotate the parameter of a function with the set of the domains of the intersection type we want to deduce. In other terms, one could try to explicitly list the set of the types s_i to be used by the rule $[\to \mathrm{I}^{(\wedge)}]$ so that, for instance, the type deduced for $\lambda x{:}\{\texttt{Int},\texttt{Bool}\}.x$

would be $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$. Still, this is not enough because it does not avoid the paradoxes we presented at the end of Section 4.1.1. To solve these problems, Frisch et al. [24] annotate λ-abstractions with their intersection types, thus providing also their return type(s). So the identity function for integer and Booleans is written in the syntax of [24] as $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$ and the system deduces for it the type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$. Using the right annotations it is then easy to define a value that distinguishes two functional types that have different interpretations.

The first modification to the system of Section 4.1 is then to adopt for functions the syntax and typing rules of Frisch et al. [24], that is, we replace in (2) the production for λ-abstractions by the production $e ::= \lambda^{\wedge_{i\in I}s_i\to t_i}x.e$ (where I is finite) and replace in Figure 2 the rule $[\to \mathrm{I}^{(\wedge)}]$ by the following one:

$$[\to\mathrm{I}^{(\text{CDUCE})}]\ \frac{\forall i\in I\quad \Gamma,x:s_i\vdash e:t_i}{\Gamma\vdash\lambda^{\wedge_{i\in I}s_i\to t_i}x.e:t\wedge t'}\ \begin{array}{l} t=\wedge_{i\in I}(s_i\to t_i)\\ t'=\wedge_{j\in J}\neg(s'_j\to t'_j)\\ t\wedge t'\not\simeq\mathbb{0}\end{array}$$

This rule (taken verbatim from [24]) checks whether a λ-abstraction has all the arrow types listed in its annotation t and deduces for the term this type t intersected with an arbitrary finite number of negated arrow types. These negated arrow types can be chosen freely provided that the type $t\wedge t'$ remains non-empty. This rule ensures that given a function $\lambda^t x.e$ (where t is an intersection type), for every type $t_1\to t_2$, either $t_1\to t_2$ can be obtained by subsumption from t or $\neg(t_1\to t_2)$ can be added to the intersection t. In turn this ensures that, for any function and any type t either the function has type t or it has type $\neg t$ (see [49, Sections 3.3.2 and 3.3.3] for a thorough discussion on this rule). The consequences of this may look surprising. For example, it allows the system to type $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$ as $(\texttt{Int}\to\texttt{Int})\wedge\neg(\texttt{Bool}\to\texttt{Bool})$ (notice the negation) even though, disregarding its annotation, the function *does* map Booleans to Booleans. But the language is explicitly typed, and thus we cannot ignore the annotations: indeed, the function does not have type $\texttt{Bool}\to\texttt{Bool}$ insofar as its application to a Boolean does not return another Boolean but an error Ω. The point is that the theory of semantic subtyping defined by [24] gives expressions an *intrinsic semantics* (in the sense of Reynolds [54]) since the semantics of λ-abstractions depends on their explicit type annotations. This aspect is apparent when one studies the denotational semantics of CDuce (see Section 5.3 and Lanvin's PhD dissertation [40, Chapter 11]): in particular, notice that according to rule $[\to\mathrm{I}^{(\text{CDUCE})}]$ we have $\lambda^{\texttt{Int}\to\texttt{42}}x.42:\texttt{Int}\to\texttt{42}$ while $\lambda^{\texttt{Int}\to\texttt{Int}}x.42:\neg(\texttt{Int}\to\texttt{42})$ (notice the difference in the annotations). Therefore, $\lambda^{\texttt{Int}\to\texttt{Int}}x.42$ and $\lambda^{\texttt{Int}\to\texttt{42}}x.42$ must have different denotations since they may yield different results for a type-case on the type $\texttt{Int}\to\texttt{42}$.

The purpose of the rule $[\to\mathrm{I}^{(\text{CDUCE})}]$ is to ensure that, given any function and any type t, either the function has type t or it has type $\neg t$. This property not only matches the view of types as sets of values that underpins semantic subtyping, but also it is necessary to ensure subject reduction, as we explained in Section 4.1.1 (see [24] for details).[15]

[15] Although by this rule it is possible to deduce infinitely many distinct types for the same expression, the system still has a notion of principality, obtained by the introduction of *type schemes*: see Section 6.12 in [24].

4.2.3 Type-cases. The second ingredient to obtain the system of Frisch et al. [24] is the modification of the syntax for type-case expressions by adding an explicit binding. Formally, we replace the type-case expression in (2) by the following production:

$$e ::= (x{=}e{\in}t)\,?\,e{:}e$$

The expression $(x{=}e{\in}t)\,?\,e_1{:}e_2$ binds the value produced by e to the variable x, checks whether this value is of type t, if so it reduces to e_1, otherwise it reduces to e_2. Formally:

$$\begin{array}{ll}(x{=}v{\in}t)\,?\,e_1{:}e_2 \rightsquigarrow e_1\{v/x\} & \text{if } v \in t\\ (x{=}v{\in}t)\,?\,e_1{:}e_2 \rightsquigarrow e_2\{v/x\} & \text{if } v \notin t\end{array}$$

Since functions are explicitly annotated by their types, it is now possible to define the relation $v \in t$ without using the type-deduction system.[16] It is easy to prove that for a well typed value v and type t that $v \in t$ is decidable (cf. Lemma 6.41 in [24])) and that we have $v \in t \iff \vdash v : t \iff \not\vdash v : \neg t \iff v \notin \neg t$ (cf. Lemma 6.22 in [24]).

Type-case expressions are needed to define full-fledged overloaded functions as opposed to having just "coherent overloading" as found in Forsythe [53]. Indeed, the rule $[{\to}\mathrm{I}^{(\mathrm{CDUCE})}]$ we added in the previous subsection, when it is not coupled with a type-case expression, allows the system to type only a limited form of *ad hoc* polymorphism known as coherent overloading [51, 53]. In languages with coherent overloading, such as Forsythe or the system by Barbanera et al. [1] (or our system without type-case expressions), it is not possible to distinguish $(s_1{\to}t_1)\wedge(s_2{\to}t_2)$ from $(s_1{\vee}s_2){\to}(t_1{\wedge}t_2)$, in the sense that they both type exactly the same set of expressions.[17] The equivalence (or indistinguishability) of the two types above states that it is not possible to have a function with two distinct behaviors chosen according to the type of the argument: the behavior is the same for inputs of type s_1 or s_2 and the intersection of the arrow types is just a way to "refine" this behavior for specific cases. In the subtyping relation of Definition 4, instead, the relation

$$s_1 \vee s_2 \to t_1 \wedge t_2 \leq (s_1 \to t_1) \wedge (s_2 \to t_2) \tag{4}$$

is strict (i.e., the converse does not hold). Therefore, for the step (III) of [24] to hold, the language must provide a λ-abstraction that is in the larger type but not in the smaller one, for instance because for some argument in s_1 the λ-abstraction returns a result that is in t_1 but not in t_2. In general this may require the use of a type-case in the body of the function, as for $\lambda^{(\mathtt{Int}\to\mathtt{Bool})\wedge(\mathtt{Bool}\to\mathtt{Int})}x.\,(y{=}x{\in}\mathtt{Int})\,?\,(y{=}{=}1){:}42$ which is a function that has type $(\mathtt{Int}{\to}\mathtt{Bool})\wedge(\mathtt{Bool}{\to}\mathtt{Int})$ but not $\mathtt{Int}{\vee}\mathtt{Bool}\to\mathtt{Int}{\wedge}\mathtt{Bool}$: since $\mathtt{Int}{\wedge}\mathtt{Bool} = \mathbb{0}$, then the second type contains only functions that diverge on arguments in $\mathtt{Int}\vee\mathtt{Bool}$, which is not the case for the function above. Thanks to the presence of type-cases we can thus distinguish these two types by a value; without type-cases, the only functions in $(\mathtt{Int}{\to}\mathtt{Bool})\wedge(\mathtt{Bool}{\to}\mathtt{Int})$ we could define would

[16] We have $v \in t \overset{\text{def}}{\iff} \exists s{\in}\mathsf{typeof}(v).\,s \leq t$ where $\mathsf{typeof}(v)$ is inductively defined as: $\mathsf{typeof}(c) \overset{\text{def}}{=} \{b_c\}$, $\mathsf{typeof}(\lambda^{\bigwedge_{i\in I} s_i\to t_i}x.e) \overset{\text{def}}{=} \{t \mid t \simeq (\bigwedge_{i\in I} s_i \to t_i) \wedge (\bigwedge_{j\in J}\neg(s'_j \to t'_j)), t \not\leq \mathbb{0}\}$, $\mathsf{typeof}((v_1,v_2)) \overset{\text{def}}{=} \mathsf{typeof}(v_1)\times\mathsf{typeof}(v_2)$.

[17] It is not possible to prove that the two types are equivalent in the system of [53] but this can be done for the system of [1].

be those that (disregarding their annontations) could be typed also by $\texttt{Int} \vee \texttt{Bool} \to \mathbb{0}$ and, thus, they would diverge on all their arguments.

The typing rules for type-case expressions are, once again, taken verbatim from [24]

$$[\textsc{Case}]\ \frac{\Gamma \vdash e : t' \quad \Gamma, x : t' \wedge t \vdash e_1 : s \quad \Gamma, x : t' \wedge \neg t \vdash e_2 : s}{\Gamma \vdash (x{=}e{\in}t)\,?\,e_1 : e_2 : s} \qquad [\textsc{Efq}]\ \frac{}{\Gamma, x : \mathbb{0} \vdash e : t}$$

and they replace the rules $[\vee^{(\leq)}]$, $[\in_1^{(\leq)}]$, and $[\in_2^{(\leq)}]$ of Figure 2 (thus solving the last problem listed in Section 4.1.3). The [Case] rule infers the type t' of the tested expression e, and then infers the types of the branches by taking into account the outcome of the test. Namely, it infers the type of e_1 under the hypothesis that x is bound to a value that was produced by e (i.e., of type t') and passed the test (i.e., of type t): that is, a value of type $t \wedge t'$; it infers the type of e_2 under the hypothesis that x is bound to a value that was produced by e (i.e., of type t') and did *not* pass the test (i.e., of type $\neg t$). The reader will surely have recognized that the rule [Case] is nothing but a specific instance of the union-elimination rule [$\vee$] for a type-case expression, where the expression e' of [$\vee$] is the expression tested by the type-case and the bind for the variable x is explicitly given by the syntax of the expression. Finally, rule [Efq] (ex falso quodlibet) is used for when in the rule [Case] either $t \wedge t'$ or $\neg t \wedge t'$ is empty: this means that the corresponding branch cannot be selected whatever the result of e is and therefore, thanks to [Efq] the branch is not typed (it is given any type, in particular the type of the other branch). For more discussion on the [Case] rule and its various implications, the reader can refer to Section 3.3 of [24] or Section 3.3 of [7] (see also the related work section of [5]).

4.2.4 Random choice. The very last ingredient to obtain the system of Frisch et al. [24] is the addition of expressions for random choice.

Formally we add to the previous grammar the production $e ::= \mathsf{choice}(e,e)$. The semantics of this expression is just a random choice of one of its arguments:

$$\mathsf{choice}(e_1,e_2) \rightsquigarrow e_1$$
$$\mathsf{choice}(e_1,e_2) \rightsquigarrow e_2$$

The need for a choice operator can be evinced by considering the interpretation of function spaces given in Definition 3. Notice indeed that functions are interpreted as finite relations, but we do not require them to be deterministic, that is, in a finite relation there may be two pairs with the same first projection but different second projections. More concretely, if $e_1 : t_1$ and $e_2 : t_2$ then $\mathsf{choice}(e_1,e_2)$ allows us to define a value that separates the type $s \to t_1 \vee t_2$ from the type $(s{\to}t_1) \vee (s{\to}t_2)$ (in Definition 3 the interpretation of the latter type is strictly contained in the interpretation of the former type), since $\lambda x.\mathsf{choice}(e_1,e_2)$ is a value in the first type that it is not in the second type. This is formalized by the following straightforward typing rule.

$$[\textsc{Choice}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \mathsf{choice}(e_1,e_2) : t_1 \vee t_2}$$

The complete deduction system for Core CDuce is summarized in Figure 3. It is formed

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

$$[\to\mathrm{I}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : t \wedge t'}\ \begin{array}{l} t = \wedge_{i\in I}(s_i \to t_i) \\ t' = \wedge_{j\in J} \neg(s'_j \to t'_j) \\ t \wedge t' \not\simeq \mathbb{0} \end{array} \qquad [\to\mathrm{E}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t_1 \circ t_2}\ \begin{array}{l} t_1 \leq \mathbb{0} \to \mathbb{1} \\ t_2 \leq \mathsf{dom}(t_1) \end{array}$$

$$[\times\mathrm{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\mathrm{E}_i]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \pi_i e : \boldsymbol{\pi_i}(t)}\ \begin{array}{l} t \leq \mathbb{1} \times \mathbb{1} \\ i = 1, 2 \end{array}$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash e : t' \quad \Gamma, x : t \wedge t' \vdash e_1 : s \quad \Gamma, x : \neg t \wedge t' \vdash e_2 : s}{\Gamma \vdash (x{=}e{\in}t)\,?\,e_1 : e_2 : s} \qquad [\textsc{Efq}]\ \frac{}{\Gamma, x : \mathbb{0} \vdash e : t}$$

$$[\textsc{Choice}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \mathsf{choice}(e_1, e_2) : t_1 \vee t_2}$$

Fig. 3: Algorithmic system for the core calculus of CDuce

by the choice rule plus the rules [Case] and [Efq] of Section 4.2.3, the rule $[\to \mathrm{I}^{(\wedge)}]$ of Section 4.2.2, the rules $[\to\mathrm{E}^{(\mathrm{A})}]$ and $[\times\mathrm{E}_i^{(\mathrm{A})}]$ of Section 4.1.2, and the rules [Const], [Var], and $[\times\mathrm{I}]$ of Figure 1. The resulting deduction system is algorithmic: it is syntax-directed and formed by analytic rules (with a small caveat for $[\to\mathrm{I}]$, see Footnote 15). The complete definition of the core calculus for CDuce is summarized for the reader's convenience in Appendix A. Finally, the resulting system has enough points to distinguish all types that have different interpretations. In particular, the value interpretation of types for this language, defined as $[\![t]\!]_{\mathscr{V}} = \{v \mid \varnothing \vdash v : t\}$, induces the same subtyping relation as the interpretation $[\![\cdot]\!]$ of Definition 3: the circle is closed.

As a final remark, it is important to stress that all these types and typing rules can also be used for languages that do not have all the constructs described above and that, thus, do not "close the circle". The only drawback is that the type system will not be used at its full potential and result more restrictive than needed. For instance, if a language does not include overloaded functions, then we we could safely consider the relation in (4) to be an equivalence rather than a simple containment and thus safely allow expressions of the type on the right-hand side to be used where expressions of the type on the left-hand side are expected. This is forbidden by the current definition of the subtyping relation which, for such a language, would thus be more restrictive than it should be. Although this looks as a minor problem, it is however possible to modify the definition of the subtyping relation to validate the equality (and close the circle) as it is shown in Section 5.9 of Frisch's PhD thesis [22].

4.2.5 Polymorphic language. Hitherto, the system presented in this section is monomorphic. Although we did not explicitly state it, the meta-variable t used so far ranged over the monomorphic types of Definition 1, which did not include type-variables. In particular, Theorem 5.2 of [24] that states the equivalence of type containment in the value interpretation and in the domain $\mathscr{D}$ and, thus, "closed the circle", is valid only for monomorphic types: it is not possible to give a value interpretation to polymorphic types

insofar as there is no value whose type is a type variable, even though type-variables are *not* empty types. Likewise, the property that for every value v and type t either $v : t$ or $v : \neg t$ no longer holds if type variables may occur in types: for instance, 42 is neither of type α nor of type $\neg\alpha$.

Nevertheless, if we want the function `flatten` in the introduction to be applicable to any well-typed argument, then we need to add polymorphic types to CDuce, since the monomorphic version of this function requires a different implementation of `flatten` for each ground instantiation of the type `Tree(`α`)→List(`α`)`. A similar argument holds for the function `balance` in Section 2.

The extension of CDuce with polymorphic types is as conceptually simple as its practical implementation is difficult. To add polymorphism to CDuce it suffices to take the grammar of the monomorphic expressions of CDuce as is and use polymorphic types wherever monomorphic ones were used, with a single exception: since the property that a value v has either type t or type $\neg t$, no longer holds for every type t, but just for closed types, then we restrict type-case expressions to test only closed types, that is:

Types	$t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} \mid \alpha$
Test Types	$\tau ::= b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0}$
Expressions	$e ::= c \mid x \mid \lambda^{\wedge_{i\in I} t_i \to t_i} x.e \mid ee \mid \pi_i e \mid (e,e) \mid (x{=}e{\in}\tau)\,?\,e : e \mid \mathsf{choice}(e,e)$

To type these expressions all we need to do is to add a single typing rule to account for the fact that if an expression has a polymorphic type, then it has also all the instances of this type; and since there are multiple instances of a type, then it has also their intersection. In other terms we add to Figure 3 the following rule

$$[\textsc{Inst}^{(\wedge)}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : \bigwedge_{i\in I} t\sigma_i}$$

where I is a finite set, σ_i's denote type substitutions, that is, finite mappings from variable to types, and $t\sigma_i$ is their application to a type t. No other modification is necessary.

Thanks to these modifications it is now possible to define in CDuce, say, the polymorphic identity function $\lambda^{\alpha\to\alpha}x.x$ which is of type $\alpha \to \alpha$. By an application of the $[\textsc{Inst}^{(\wedge)}]$ rule we can deduce for it the type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ and, thanks to this deduction it is possible to infer for the application $(\lambda^{\alpha\to\alpha}x.x)(\mathsf{choice}(3,\texttt{true}))$ the type $\texttt{Int} \vee \texttt{Bool}$.[18]

Why then is the practical implementation of this system so difficult? The reader will have noticed that since we added $[\textsc{Inst}^{(\wedge)}]$ to the deduction system in Figure 3, then the system is no longer algorithmic. The new rule is neither syntax-directed (it applies to a generic expression e) nor analytic (it is not clear how to determine the set of type substitutions $\{\sigma_i\}_{i\in I}$ to apply in the rule). The latter, that is determining type substitutions, is the real challenge for implementing polymorphic CDuce. We will not explain the details about how to do it: this has needed two distinct papers (part 1 [10] and part 2 [12]) to which the reader can refer for all details. Bottom line, all this

[18] As a side note, even if property (3) does not hold in this system (e.g., `42` is neither of type α nor of type $\neg\alpha$) this does not hinder the soundness of system since subject reduction holds for all well typed terms with ground types, that is for all ground instances of a program. This is enough to prove the soundness of the system.

complexity is hidden to the programmer: CDuce does it for her. Nevertheless, we want to outline some aspects that can give the reader a flavor of the complexity that underlies this implementation.

A first ingredient that is necessary in all languages with implicit parametric polymorphism (also known as *prenex* or *second-order* polymorphism) is type unification: to type the application of a (polymorphic) function of type $t \to t'$ to an argument of type t'' one has to unify the type of the argument with the domain of the function, viz., to find a type substitution σ such that $t\sigma = t''\sigma$. However, in a polymorphic language with subtyping this may not be enough since, in general, we need a type-substitution that makes the type of the argument *a subtype* of the domain of the function. In other terms we need to solve the *type tallying problem* [12], that is, given two types t and t' find all type substitutions σ such that $t\sigma \leq t'\sigma$. For instance, consider the types we defined at the end of Section 2: if we want to apply a function whose domain is `RBTree(`α`)` (a red-black tree with generic labels) to an argument of type `RTree(Int)` (a *red* tree with integer labels), then we need the substitution $\sigma = \{\alpha \mapsto \texttt{Int}\}$ since (`RTree(Int)`)σ = `RTree(Int)`$\leq$ (`BTree(Int)`$\vee$`RTree(Int)`) = (`RBTree(`α`)`)σ (notice that the subtyping relation in the middle is strict). The type tallying problem is decidable for the polymorphic types of Section 3. In Castagna et al. [12, Appendix C] we defined an algorithm that returns a set of type-substitutions that is sound and complete with respect to the tallying problem: every substitution in the set is a solution, and every solution is an instance of the substitutions in the set. The reason why the tallying problem admits as solution a principal set of substitutions—rather than a single principal substitution—is due to the presence of set-theoretic types. For instance the problem of finding a substitution σ such that $(\alpha_1 \times \alpha_2)\sigma \leq (\beta_1 \times \beta_2)\sigma$ admits three incomparable solutions: (i) $\{\alpha_1 \mapsto \mathbb{0}\}$, (ii) $\{\alpha_2 \mapsto \mathbb{0}\}$, and (iii) $\{\alpha_1 \mapsto \beta_1, \alpha_2 \mapsto \beta_2\}$.

While the capacity of solving the type tallying problem is necessary to type the applications of polymorphic functions, this capacity alone is not sufficient. The reason is that functions can be typed not only by instantiating their types, but also by what is commonly called *expansion*: as stated by rule [Inst$^{(\wedge)}$] an expression, thus a function, can be typed by any intersection of instantiations of its type. Consider the function:

```
let even : (Int→Bool) & (α\Int→α\Int) =
  fun x -> (x∈Int) ? ((x mod 2)==0) : x
```

or, in Core CDuce syntax, $\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})}x.\,(y{=}x{\in}\texttt{Int})\,?\,((y \bmod 2){=}{=}0)\,{:}\,y$. The function is polymorphic: if applied to an integer it returns a Boolean (i.e., whether the argument is even or not), otherwise it returns the argument. Notice that the type of this function is not weird since it follows the same pattern as the type of the `balance` function we defined in Section 2. Next consider the classic `map` function:

```
let rec map : (α→β)→List(α)→List(β) =
  fun f l -> match l with                                   (5)
    |  [] -> []
    |  h::t ->  (f h)::(map f t)
```

and the partial application `map even` for which polymorphic CDuce infers the type

```
map even : ( List(Int) → List(Bool) ) ∧
           ( List(γ\Int) → (List(γ\Int)) ∧                  (6)
           ( List(γ∨Int)) → List((γ\Int)∨Bool) )
```

stating that `map even` returns a function that when applied to a list of integers it returns a list of Booleans; when applied to a list that does not contain any integer, then it returns a list of the same type (actually, the same list); and when it is applied to a list that may contain some integers (e.g., 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 with reals that are not integers). The typing of `map even` shows that the sole tallying is not sufficient to obtain such a precise type: the result is obtained by inferring three different instantiations[19] of the type of `map`, taking their intersection and tallying it with the type of `even`. This is obtained by the CDuce type-checker by trying different expansions of the types of the function and of the argument, implementing a dove-tail search. For a detailed explanation the reader can refer to Castagna et al. [12].

The language presented in this subsection is the core of the polymorphic version of CDuce implemented in the development branch of the language. It is possible to define in it the functions `flatten`, `balance`, `map`, and `even` of Sections 1, 2 and here above as long as they are explicitly typed: CDuce requires every function to be annotated with its type. CDuce also performs occurrence typing, but it requires the tested expression either to be a variable or to be explicitly bound to a variable on which the union elimination rule is applied.

In the next section we show how to get rid of the mandatory annotations for functions (alas at the expense of inferring intersection types for them), while in Section 4.4 we present a language in which union elimination is implemented without any restriction and intersection types for functions are inferred without need of annotations (alas at the expense of polymorphism).

4.3 An Implicitly-Typed Polymorphic Language with Set-Theoretic Types

The polymorphic language of the previous section requires to explicitly annotate every function with its type. While for top-level functions this may be often advisable and sometime necessary—e.g., for documentation purposes or for exporting the functions in a library—, it is in general an annoying burden for the programmer, especially for local functions many of which seldom require to be documented with a precise type. Besides, determining the right annotation may be mind-boggling if not impossible, even for very simple functions: for instance, consider the function $\lambda x.(\lambda y.(x,y))x$ which clearly has the type $(\texttt{Int}\to\texttt{Int}\times\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool}\times\texttt{Bool})$ since it always returns a pair obtained by duplicating the function's argument; as an exercise the reader may try to annotate it without polymorphic types (and without reading the solution in the footnote), so as to deduce the above intersection type.[20]

[19] For `map even` we need to infer just two instantiations, namely, $\{\alpha\mapsto(\gamma\setminus\texttt{Int}),\beta\mapsto(\gamma\setminus\texttt{Int})\}$ and $\{\alpha\mapsto(\gamma\vee\texttt{Int}),\beta\mapsto(\gamma\setminus\texttt{Int})\vee\texttt{Bool}\}$. The type in (6) is redundant since the first type of the intersection is an instance (e.g., for γ=`Int`) of the third. We included it just for the sake of the presentation.

[20] This is impossible since the type to give to the local function $\lambda y.(x,y)$ depends on the hypothesis on x: annotating the inner function with the above intersection type would not work since when x is of type `Int`, then the local function does not have type `Bool` → `Bool`×`Bool` and similarly for the case when x is of type `Bool`. The only solution is to annotate both functions

To obviate these problems we studied how to type the implicitly-typed language of grammar (2) in Section 4.1, whose λ-abstractions are not annotated, using the polymorphic types of Section 3.2. The first results of this study were presented in Castagna et al. [13] whose system was later greatly improved and superseded by Petrucciani's Ph.D. dissertation [49, see Part 2] on which the rest of this subsection is largely based. In order to make type-inference for the implicitly-typed λ-abstractions in (2) feasible, we define a system that imposes several restrictions that are absent from the explicitly-typed polymorphic CDuce we described in Section 4.2.5, namely:

1. the system implements the so-called *let-polymorphism*, characteristic of languages of the ML-family or Hindley-Milner systems, according to which the type system can only instantiate the type of expressions[21] that are bound in a let construct. This contrasts with the system in Section 4.2.5 where the type of every expression can be instantiated.
2. type-case expressions can test only types that do not have any functional subcomponent other than $\mathbb{0} \to \mathbb{1}$;
3. the type-system does not infer negated arrow types for functions;
4. the reconstruction algorithm does not infer intersection types for functions.

Obviously, to implement let-polymorphism we need to extend the grammar (2) with a let-expression. The language thus has the following definition:

$$\begin{array}{ll} \textbf{Test Types} & \tau ::= b \mid \tau \times \tau \mid \mathbb{0} \to \mathbb{1} \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \\ \textbf{Expressions} & e ::= c \mid x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi_i e \mid (e \in \tau)\,?\,e : e \mid \texttt{let}\, x = e\, \texttt{in}\, e \end{array}$$

As anticipated type-cases cannot test arbitrary types, since they use the restricted grammar for test types τ. There are two restrictions with respect to the types in (1): types must be ground (as in Section 4.2.5, α does not appear in the definition of τ) and the only arrow type that can appear is $\mathbb{0}\to\mathbb{1}$, that is, the type of all functions. This means that type-cases can distinguish functions from non-functions but cannot distinguish, say, the functions that have type $\texttt{Int}\to\texttt{Int}$ from those that do not. Type-cases of this form have the same expressiveness as the type-testing primitives of dynamic languages like JavaScript and Racket. The definitions of values and of the reduction semantics rules given in Section 4.1 do not change. To account for the new let-expressions we add to these definitions the notion of reduction $\texttt{let}\, x = v\, \texttt{in}\, e \rightsquigarrow e\{v/x\}$ together with the new evaluation context $\texttt{let}\, x = E\, \texttt{in}\, e$. As for CDuce, the relation $v \in t$ (actually, $v \in \tau$) used in the reduction semantics of type-cases can be defined independently from the type system. Here the definition is even simpler than the one given in Section 4.2.3 (cf. Footnote 16) since we have $v \in t \overset{\text{def}}{\iff} \mathsf{typeof}(v) \leq t$ and $v \notin t \overset{\text{def}}{\iff} \mathsf{typeof}(v) \leq \neg t$ where $\mathsf{typeof}(c) \overset{\text{def}}{=} b_c$, $\mathsf{typeof}(\lambda x.e) \overset{\text{def}}{=} \mathbb{0}\to\mathbb{1}$, and $\mathsf{typeof}((v_1, v_2)) \overset{\text{def}}{=} \mathsf{typeof}(v_1) \times \mathsf{typeof}(v_2)$. Note that typeof maps every λ-abstraction to $\mathbb{0}\to\mathbb{1}$. This approximation is allowed by the restriction on test types in type-cases.

The most important changes with respect to the theoretical framework of Section 4.1 are to be found in the type-system, defined in Figure 4 where, as anticipated, t, t', t_1, and t_2 range over the polymorphic types defined in Section 3.2 grammar (1). The type

with the type $\alpha \to \alpha\times\alpha$ and deduce the intersection type by applying rule [Inst$^{(\wedge)}$]. See also Section 4.4 which introduces more expressive annotations that can type this example.

[21] In practice, values, see the so called *value restriction* suggested by Wright [64].

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : t\{\vec{\alpha}\mapsto\vec{t}\}}\ \Gamma(x) = \forall\vec{\alpha}.t$$

$$[\to\text{I}]\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$[\times\text{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\text{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\times\text{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash e : t' \quad \text{either } t' \leq \neg t \text{ or } \Gamma \vdash e_1 : s \quad \text{either } t' \leq t \text{ or } \Gamma \vdash e_2 : s}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 : s}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : \forall\vec{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : t}\ \vec{\alpha} \sharp \Gamma$$

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\leq]\ \frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'}$$

Fig. 4: Typing rule for let-polymorphism

system described in Figure 4 is very similar to a standard Hindley-Milner system: the differences are just the addition of subtyping and intersection introduction, as well as a rule for type-cases. As in Hindley-Milner type systems, we introduce a notion of *type scheme* separate from that of types. A type scheme, denoted by $\forall\alpha_1, ..., \alpha_n.t$ and abbreviated as $\forall\vec{\alpha}.t$, binds the type variables α_1, ...,α_n in t. We view types as a subset of type schemes identifying $\forall\vec{\alpha}.t$ with t if $\vec{\alpha}$ is empty. Type environments map variables into type-schemes and we write $\alpha \sharp \Gamma$ for the property that α does not occur free in Γ (type schemes are considered equivalent modulo α-renaming of the type variables).

If we compare the rules in Figure 4 with those of the theoretical framework in Figure 1 we will notice several differences. Foremost, the union elimination rule [$\vee$] is no longer present. Since this rule played a key role in typing type-case expressions, then the three rules [$\mathbb{0}$], [$\in_1$], and [$\in_2$] for type cases are replaced in Figure 4 by a single rule [Case] that skips the typing of a branch when this is not selectable. The only other difference is in the rule [Var]. This is classic in Hindley-Milner system: type environments maps variables into type-schemes and [Var] instantiates these variables with a set of types $\vec{t}$. This rule is coupled with the new rule [Let] that infers the type t_1 of the argument of the let-expression and generalizes this type by binding in the type of x all variables $\vec{\alpha}$ that are not free in Γ. A final observation, while by the rule [$\wedge$] it is possible to deduce intersection types for functions, it is not possible to deduce negation types (other than by subsumption). Nevertheless the system is sound, but the proof needs to deduce these negation types which, since the λ-abstractions are not annotated, is not straightforward: see Petrucciani [49, §3.3].

4.3.1 Type reconstruction. The next problem is to define an algorithm of *type reconstruction*[22] for this implicitly-typed language with set-theoretic types (that we dub Implicit CDuce, for short). The algorithm defined by Petrucciani [49, Chapter 4] does not attempt to infer intersection types: that would complicate type inference because we cannot easily know how many types we should infer and intersect for a given expression, notably for a function. Therefore, the algorithm of type reconstruction is sound with respect to the type system in Figure 4 and complete with respect the same system without the rule [$\wedge$]. The algorithm follows a pattern that is common with Hindley-Milner system and consists in producing sets of structured constraints, that are then simplified into sets of subtyping constraints to be solved by the tallying algorithm we hinted at in Section 4.2.5. For space reasons we just outline its main characteristics and some specificities of type reconstruction for set-theoretic types.

The main difference with respect to Hindley-Milner systems is that we reduce type reconstruction to solving sets of constraints that are subtyping constraints (rather than type equality constraints) that we then solve by using tallying (rather than unification).

A subtlety of the Hindley-Milner type system is in generalization: to type e_2 in $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$, we assign to x the type scheme obtained from the type of e_1 by quantifying over all type variables *except those that are free in the type environment*. This restriction is needed to ensure soundness. Therefore, whether the binding for a variable x of a let-expression is polymorphic or not (and if it is, which type variables we can instantiate) depends on a comparison of the type variables that appear syntactically in the type of the bound expression and in the type environment. This is problematic with semantic subtyping: we want to see types up to the equivalence relation $\simeq$ (that is, to identify types with the same set-theoretic interpretation), but two types can be equivalent while having different type variables in them. For instance, $\alpha \wedge \mathbb{0}$ and $\alpha \setminus \alpha$ are both equivalent to $\mathbb{0}$, but α occurs in them and not in $\mathbb{0}$. This means that we cannot see type environments up to equivalence of their types, since the type schemes in them were generalized according to the variables that syntactically occurred in the environment. The absence of this property is problematic during constraint resolution, in particular when applying type substitutions. The solution to this problem is to adopt a technique akin to the *reformulated typing rules* of Dolan and Mycroft [20]. First of all, note that our current type environments Γ bring two different kinds of hypotheses: they map λ-abstracted variables into types (that cannot be instantiated: the variables have monomorphic types) and let-abstracted variables into type schemes (which can be instantiated by replacing the quantified variables by types: the variables have polymorphic types). As a first step, let us separate these two kinds of hypotheses and replace our type environment Γ by a monomorphic type environment M for λ-abstracted variables and a polymorphic type environment P for let-abstracted variables: for clarity we distinguish the latter variables by superposing a hat on them, such as in $\hat{x}$ and $\hat{y}$. The second step is to notice that type schemes are obtained by generalizing the type variables that do not occur in the *monomorphic* part of the type environment. This observation allows us to get rid of type schemes and generalization by replacing them with *typing schemes* [20]

[22] We use this term in the sense of Pierce [50], that is, reconstructing the type information in an implicitly-typed language. Sometimes the terms *type inference* or *type assignment system* are equivalently used in the literature.

that record how a polymorphic type depends on the current monomorphic type environment. So while a monomorphic type environment M still maps a λ-abstracted variable x into a type t, a polymorphic environment P maps a let-abstracted variable $\hat{x}$ into a *typing scheme* $\langle M \rangle t$ where M is a monomorphic type environment and t a type. For instance, consider the expression $\lambda x.(\mathtt{let}\,\hat{x}\mathtt{=}\lambda y.(x,y)\,\mathtt{in}\,e)$. With type schemes we would choose α as the type of x, type $\lambda y.(x,y)$ as $\beta \to \alpha \times \beta$, and then, to type e, we would assign to $\hat{x}$ the type scheme $\forall\beta.\beta \to \alpha \times \beta$ (we generalized β but not α). In the reformulated system, in contrast, $\hat{x}$ is assigned the *typing scheme* $\langle x\colon \alpha \rangle(\beta \to \alpha \times \beta)$ where all type variables are implicitly quantified (the reconstruction algorithm will be allowed to instantiate all of them: *cf.*, rule [$\hat{x}$] in Figure 6) and can be α-renamed: we could equivalently choose for $\hat{x}$ the typing scheme $\langle x\colon \gamma \rangle(\delta \to \gamma \times \delta)$, since we do not care which type variables we use, but only that the dependency is recorded correctly. Using this system, the previous difficulties with generalization do not arise: we can give $\hat{x}$ the type $\alpha \vee (\gamma \setminus \gamma)$ equivalent to α as long as this does not capture an implicitly quantified type variable of the typing scheme (in the present case γ).

Once we have fixed this point, then reconstruction consists in constraint generation and constraint solving. On the lines of Pottier and Rémy [52], Petrucciani [49, Chapter 4] introduces two notions of constraint. The first, *type constraints* $(t_1 \prec t_2)$, constrain a solution (a type substitution σ) to satisfy subtyping between two types (that is, to satisfy $t_1\sigma \leq t_2\sigma$). In the absence of let-polymorphism, the type inference problem can be reduced to solving such type constraints, as done by Wand [63] for unification. In our setting, as for type inference for ML, it would force us to mix constraint generation with constraint solving. Therefore, we introduce *structured constraints*, which allow us to keep the two phases of constraint generation and constraint solving separate. These constraints can mention expression variables and include binders to introduce new variables. Structured constraints are closely related to those in the work of Pottier and Rémy [52] on type inference for ML and are defined as follows:

$$C ::= (t \prec t) \mid (x \dot{\leq} t) \mid (\hat{x} \dot{\leq} t) \mid C \wedge C \mid C \vee C \mid \exists\vec{\alpha}.C \mid \mathsf{def}\, x\colon t\ \mathsf{in}\ C \mid \mathsf{let}\, \hat{x}\colon \forall\alpha[C]\ \mathsf{in}\ C$$

Structured constraints include type constraints $(t \prec t)$ but also several other forms. The two forms $(x \dot{\leq} t)$ and $(\hat{x} \dot{\leq} t)$ constrain the type or typing scheme of the variable. Constraints include conjunction and disjunction. The existential constraint $\exists\vec{\alpha}.C$ introduces new type variables (it simplifies freshness conditions). Finally, the def and let constraints introduce the two forms of expression variables and are used to describe the constraints for λ-abstractions and let-expressions, respectively. Their meaning can be evinced from the definition of the *constraint generation* function $\langle\!\langle \cdot : \cdot \rangle\!\rangle$ that, given an expression e and a type t, yields a structured constraint $\langle\!\langle e : t \rangle\!\rangle$. This constraint expresses the conditions under which e has type $t\sigma$ for some type substitution σ. It is defined in Figure 5 where $\alpha, \alpha_1, \alpha_2$ do not occur in t. The constraints for variables and constants are straightforward. To type $\lambda x.e$ with type t, the system generates two fresh variables α_1 and α_2, generates the constraint for e to be of type α_2 under the hypothesis that x is of type α_1, and adds the constraint that t subsumes $\alpha_1 \to \alpha_2$. Note that the constraint generation associates $\lambda x.e$ to a single arrow $\alpha_1 \to \alpha_2$ since, as anticipated, it does not attempt to infer intersection types. The constraints for applications, pairs and projections are self-explaining. For type-cases, the system generates the constraint for

$$
\begin{aligned}
\langle\!\langle \hat{x} : t \rangle\!\rangle &= (\hat{x} \dot{\leq} t)\\
\langle\!\langle x : t \rangle\!\rangle &= (x \dot{\leq} t)\\
\langle\!\langle c : t \rangle\!\rangle &= (\boldsymbol{b}_c \lessdot t)\\
\langle\!\langle (\lambda x.e) : t \rangle\!\rangle &= \exists\alpha_1,\alpha_2.(\mathsf{def}\, x\colon \alpha_1 \,\mathsf{in}\, \langle\!\langle e : \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \to \alpha_2 \lessdot t)\\
\langle\!\langle e_1 e_2 : t \rangle\!\rangle &= \exists\alpha.\langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 : t \rangle\!\rangle\\
\langle\!\langle (e_1, e_2) : t \rangle\!\rangle &= \exists\alpha_1,\alpha_2.\langle\!\langle e_1 : \alpha_1 \rangle\!\rangle \wedge \langle\!\langle e_2 : \alpha_2 \rangle\!\rangle \wedge (\alpha_1 \times \alpha_2 \lessdot t)\\
\langle\!\langle \pi_i e : t \rangle\!\rangle &= \exists\alpha_1,\alpha_2.\langle\!\langle e : \alpha_1 \times \alpha_2 \rangle\!\rangle \wedge (\alpha_i \lessdot t)\\
\langle\!\langle (\texttt{(}e_0 \in \tau\texttt{)?}\, e_1 \texttt{:}\, e_2) : t \rangle\!\rangle &= \exists\alpha.\langle\!\langle e_0 : \alpha \rangle\!\rangle \wedge \big((\alpha \lessdot \neg\tau) \vee \langle\!\langle e_1 : t \rangle\!\rangle\big) \wedge \big((\alpha \lessdot \tau) \vee \langle\!\langle e_2 : t \rangle\!\rangle\big)\\
\langle\!\langle (\texttt{let}\, \hat{x} \texttt{=} e_1 \,\texttt{in}\, e_2) : t \rangle\!\rangle &= \mathsf{let}\, \hat{x}\colon \forall\alpha[\langle\!\langle e_1 : \alpha \rangle\!\rangle] \,\mathsf{in}\, \langle\!\langle e_2 : t \rangle\!\rangle
\end{aligned}
$$

Fig. 5: Constraint generation

the tested expression to be of type α, for a fresh α, and then it types the two branches provided that they can be selected, viz., either the constraint that e_1 is of type t must be satisfied or the first branch cannot be selected since α is a subtype of $\neg\tau$, and similarly for the second branch. Finally, for let-expressions it generates the constraints for e_1 remembering that the type α of e_1 can be generalized, and under this hypothesis generates the constraints for e_2 to be of type t.

Once the structured constraints are generated for a given expression they are simplified to obtain a set of *type constraints* whose solution yields the type of the expression. This is done by an algorithm that takes as input a polymorphic type environment P and a structured constraint C and produces a set of type constraints D (which is then solved by tallying), a monomorphic type-environment M (which collects the constraints $x \dot{\leq} t$ in C) and a set of variables $\vec{\alpha}$ (that collects the type variables introduced during the simplification of C). This is written as $P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}$ and defined by the deduction rules in Figure 6.

$$
[\lessdot]\ \frac{}{P \vdash (t_1 \lessdot t_2) \rightsquigarrow \{t_1 \lessdot t_2\} \mid \varnothing \mid \varnothing} \qquad [x]\ \frac{}{P \vdash (x \dot{\leq} t) \rightsquigarrow \varnothing \mid (x\colon t) \mid \varnothing}
$$

$$
[\hat{x}]\ \frac{}{P \vdash (\hat{x} \dot{\leq} t) \rightsquigarrow \{t_1\{\vec{\alpha}\mapsto\vec{\beta}\} \lessdot t\} \mid M_1\{\vec{\alpha}\mapsto\vec{\beta}\} \mid \vec{\beta}}\ \begin{cases} P(\hat{x}) = \langle M_1 \rangle t_1 \\ \vec{\alpha} = \mathsf{tvar}(\langle M_1 \rangle t_1) \\ \vec{\beta} \sharp t \end{cases}
$$

$$
[\wedge]\ \frac{P \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1 \qquad P \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2}{P \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2 \mid M_1 \wedge M_2 \mid \vec{\alpha}_1 \cup \vec{\alpha}_2}\ \begin{cases} \vec{\alpha}_1 \sharp \vec{\alpha}_2, C_2 \\ \vec{\alpha}_2 \sharp C_1 \end{cases}
$$

$$
[\vee]\ \frac{P \vdash C_i \rightsquigarrow D \mid M \mid \vec{\alpha}}{P \vdash C_1 \vee C_2 \rightsquigarrow D \mid M \mid \vec{\alpha}} \qquad [\exists]\ \frac{P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}'}{P \vdash \exists\vec{\alpha}.C \rightsquigarrow D \mid M \mid \vec{\alpha}' \cup \vec{\alpha}}\ \vec{\alpha}' \sharp \vec{\alpha}
$$

$$
[\textsc{Def}]\ \frac{P \vdash C \rightsquigarrow D \mid M \mid \vec{\alpha}}{P \vdash \mathsf{def}\, x\colon t \,\mathsf{in}\, C \rightsquigarrow D \cup \{t \lessdot M(x)\} \mid M \backslash x \mid \vec{\alpha}}\ \vec{\alpha} \sharp t
$$

$$
[\textsc{Let}]\ \frac{\begin{array}{c} P \vdash C_1 \rightsquigarrow D_1 \mid M_1 \mid \vec{\alpha}_1 \\ (P, \hat{x}\colon \langle M_1\sigma_1 : \alpha\sigma_1 \rangle) \vdash C_2 \rightsquigarrow D_2 \mid M_2 \mid \vec{\alpha}_2 \end{array}}{P \vdash \mathsf{let}\, \hat{x}\colon \forall\alpha[C_1] \,\mathsf{in}\, C_2 \rightsquigarrow D_2 \mid M_1\sigma_1\{\vec{\alpha}\mapsto\vec{\beta}\} \wedge M_2 \mid \vec{\alpha}_2 \cup \vec{\beta}}\ \begin{cases} \sigma_1 \in \mathsf{tally}(D_1) \\ \vec{\alpha} = \mathsf{tvar}(M_1\sigma_1) \\ \vec{\alpha}_1 \sharp \alpha \\ \vec{\beta} \sharp C_1, \vec{\alpha}_2 \end{cases}
$$

Fig. 6: Constraint simplification rules

A type constraint yields the singleton containing the type constraint itself (rule [$\lhd$]) while a constraint for a λ-abstracted variable returns the corresponding monomorphic environment without any other constraint (rule [x]). The first interesting rule is the one for the constraint of a let-abstracted variable (rule [$\hat{x}$]), since it performs the instantiation: if the typing scheme of $\hat{x}$ is $\langle M_1 \rangle t_1$, then the simplification instantiates *all* the type variables in the typing scheme (i.e., $\mathsf{tvar}(M_1\sigma_1)$)by some fresh variables $\vec{\beta}$ (precisely, some variables $\vec{\beta}$ not occurring in t, noted $\vec{\beta} \sharp t$), and returns the constraint that the type of $\hat{x}$ so instantiated is subsumed by t, the monomorphic environment M_1 of the constraint so instantiated, and the set $\vec{\beta}$ of fresh variables used for this instantiation. The rule for conjunction [$\wedge$] requires all the constraints to be satisfied and merges the monomorphic environments (where $M_1 \wedge M_2$ denotes the pointwise intersection of the environments[23]). Rule [$\vee$] non-deterministically chooses a constraint, while [$\exists$] ensures that the constraint uses fresh variables and records them. Rule [Def] simplifies the constraint C and adds a new type constraint $t \lhd M(x)$ (notice the contravariance, since t is the type hypothesis of a λ-abstracted variable it must be *smaller* than the type $M(x)$ needed to type the body of the function) to remove the binding of x from M, so that the domain of a monomorphic environment obtained by simplifying a constraint C is always the set of λ-abstracted variables free in C. Finally, because of let-polymorphism, the simplification algorithm uses the tallying algorithm internally to simplify let-constraints. This is done in [Let] where $\mathsf{tally}(D)$ denotes the set of type-substitutions that solve the set of type constraints D. The rule first simplifies the structured constraint C_1 and solves the resulting D_1 using tallying. Then it non-deterministically chooses a solution σ_1 of D_1 to obtain the typing scheme for $\hat{x}$, and simplifies C_2 in the expanded environment. The final monomorphic environment returned is the intersection of M_2 and a fresh renaming of $M_1\sigma_1$. In most rules, the side conditions force the choice of fresh variables.

The type reconstruction algorithm is sound: let e be a program (i.e., a closed expressions) and α a type variable, if $\varnothing \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D \mid \varnothing \mid \alpha$ and $\sigma \in \mathsf{tally}(D)$, then $\varnothing \vdash e : \alpha\sigma$ is derivable by the system in Figure 4. The algorithm is also complete with respect to the system without the intersection rule, viz., if a type t can be deduced for an expression e without using the rule [$\wedge$], then $\varnothing \vdash \langle\!\langle e : \alpha \rangle\!\rangle \rightsquigarrow D \mid \varnothing \mid \alpha$ for some D and there exists $\sigma \in \mathsf{tally}(D)$, such that t is an instance of $\alpha\sigma$. The system we presented here is a simplification of the one by Petrucciani [49]. In particular we glossed over how to handle non-determinism (disjunctive constraints and multiple solutions of $\mathsf{tally}(D)$ are the two sources of non-determinism for the algorithm) and how the introduction of fresh variables during tallying is addressed. The reader can find these details in Petrucciani [49, Chapter 4].

In this system we can now write the `map` function defined in (5) without specifying its type in the annotation: the type reconstruction algorithm will deduce it for us. This same type is deduced for the `map` function by any language of the ML-family. But of course, the use of set-theoretic types goes beyond what can be reconstructed in ML. We already gave an example in Section 2 that shows that, thanks to set-theoretic types, pattern matching can be typed to ensure exhaustivity. A second example is given by the

[23] In this rule and in the rule [Def] we suppose that $M(x) = \mathbb{1}$ for $x \notin \mathsf{dom}(M)$. Thus if $x \notin \mathsf{dom}(M)$, then $(M \wedge M')(x) = M'(x)$ and $(t \lhd M(x)) = (t \lhd \mathbb{1})$.

function `f` below which returns true for the pair of *tags* (akin to user-defined constants) `(`A,`B)`, and false for the symmetric pair:

```
let f = function                    let g = function
  | (`A,`B) -> true                   | `A -> `B                    (7)
  | (`B,`A) -> false                  |  x -> x
```

the type returned by the reconstruction algorithm for implicit CDuce and the one by OCaml (where this kind of tags are called *polymorphic variants*) are given below.

	Implicit CDuce	OCaml
`f`:	$\texttt{(`A,`B)}\vee\texttt{(`B,`A)} \to \texttt{Bool}$	$\texttt{(`A}\vee\texttt{`B , `A}\vee\texttt{`B)} \to \texttt{Bool}$
`g`:	$\forall\alpha.\texttt{`A}\vee\texttt{`B}\vee(\alpha\setminus(\texttt{`A}\vee\texttt{`B})) \to \texttt{`B}\vee\alpha$	$\forall(\alpha \geq \texttt{`A}\vee\texttt{`B}).\,\alpha \to \alpha$

While OCaml states that the function `f` can be applied to any pair of tags `` `A `` or `` `B `` (but the type-checker warns that pattern matching may not be exhaustive since it fails for, say, the pair `(`A,`A)`) the reconstruction in implicit CDuce bars out all pairs that would make pattern matching fail. But even when exhaustivity is not an issue, implicit CDuce can return more precise types, as the function `g` defined in (7) shows. The type returned by OCaml states that the function `g` will return either `` `A `` or `` `B `` or any other value that is passed to the function.[24] The type inferred by implicit CDuce states that the function `g` will return either `` `B `` or any other value passed to the function provided that it is neither `` `A `` or `` `B ``: contrary to OCaml, it correctly detects that `g` will never return a tag `` `A ``.

Finally, to understand how the reconstruction algorithm works in the presence of subtyping, consider the following OCaml code snippet (that does not involve any pattern matching or fancy data type: just products) that OCaml fails to type:

```
fun x -> if (fst x) then (1 + snd x) else x
```

Our reconstruction algorithm deduces for this function the type

$$\texttt{(Bool}\times\texttt{Int)} \to \texttt{(Int | (Bool}\times\texttt{Int))}$$

To that end, the constraint generation and simplification systems assign to the function the type $\alpha \to \beta$ and, after simplification, generates a set of four constraints: $\{(\alpha \lessdot \texttt{Bool}\times\mathbb{1}), (\alpha \lessdot \mathbb{1}\times\texttt{Int}), (\texttt{Int} \lessdot \beta), (\alpha \lessdot \beta)\}$. The first constraint is generated because `fst x` is used in a position where a Boolean is expected; the second comes from the use of `snd x` in an integer position; the last two constraints are produced to type the result of an `if_then_else` expression (with a supertype of the types of both branches). To compute the solution of two constraints of the form $\alpha \lessdot t_1$ and $\alpha \lessdot t_2$, the tallying algorithm must compute the greatest lower bound of t_1 and t_2 (or an approximation thereof); likewise for two constraints of the form $s_1 \lessdot \beta$ and $s_2 \lessdot \beta$ the best solution is the least upper bound of s_1 and s_2. This yields $\texttt{Bool} \times \texttt{Int}$ for the domain — i.e., the intersection of the upper bounds for α— and `(Int | (Bool×Int))` for the codomain—i.e., the union of the lower bounds for β.

This last example further witnesses the interest of having set-theoretic types exposed to the programmer rather than just as meta-operations implemented by the type checker. To perform type reconstruction in the presence of subtyping, one must be able

[24] In OCaml this value can only be another polymorphic variant.

to compute unions and intersections of types. In some cases, as for the domain in the example above, the solution of these operations is a type of ML (or of the language at issue): then the operations can be meta-operators computed by the type-checker but not exposed to the programmer. In other cases, as for the codomain in the example, the solution is a type which might not already exist in the language: therefore, the only solution to type the expression precisely is to add the corresponding set-theoretic operations to the types of the language.

4.3.2 Adding type annotations. The type reconstruction algorithm we just described cannot infer intersection types for functions. However it is possible to explicitly annotate functions (actually, any expression) with an intersection type and the system will *check* whether the function has that type [see 49, Chapter 5]. For instance, we can specify for the functions `f` and `g` in (7) the following annotations.

$$\texttt{f} : ((\texttt{`A},\texttt{`B}) \to \texttt{true}) \wedge ((\texttt{`B},\texttt{`A}) \to \texttt{false})$$
$$\texttt{g} : \forall\alpha.(\texttt{`A} \to \texttt{`B}) \wedge ((\alpha\setminus\texttt{`A}) \to (\alpha\setminus\texttt{`A}))$$

and the type system will accept both of them. With these explicit annotations we almost recover all the expressiveness of the system in Section 4.2.5 (it just lacks the possibility of testing function types other than $\mathbb{0}\to\mathbb{1}$). So for instance, the application of the (implicitly-typed) `map` to the function `g` explicitly annotated with the type above, will return in the system of Petrucciani [49, Chapter 5] exactly the same type as as the type of `map even` given in (6) where `` `A `` is replaced for `Int` and `` `B `` is replaced for `Bool`.

Adding annotations requires few modifications to the previous system. First of all we have, of course, to add annotations to our syntax. Here we present a simplified setting in which annotations are added only to let-expressions (see [49, Chapter 5] for the system where annotations can be added to any expressions anywhere in a program), which corresponds to adding the following production:

$$e ::= \texttt{let}\,\hat{x} : \forall\vec{\alpha}.t = e\,\texttt{in}\,e$$

A let-abstracted variable can now be annotated with an annotation $\forall\vec{\alpha}.t$ which specifies the type t to check for the expression bound to the variable, as well as the type variables $\vec{\alpha}$ that are polymorphic in t. Like in the annotation given above to the function `g`, we can specify all the variables occurring in t, but we can also omit some, meaning that they will be considered monomorphic. For instance, $\texttt{let}\,\hat{x} : \forall\alpha.\alpha\to\alpha = \lambda x.x\,\texttt{in}\,\hat{x}\texttt{3}$ is well typed, because α is bound in the let and can be instantiated in the body of the let. Instead, $\texttt{let}\,\hat{x} : \alpha\to\alpha = \lambda x.x\,\texttt{in}\,\hat{x}\texttt{3}$ is ill-typed, because α is not bound in the let and cannot be instantiated when typing the body $\hat{x}\texttt{3}$—in practice, this means that α is bound in some outer scope and is polymorphic only outside that scope.

The addition of annotations has as a consequence that now expressions may have some free *type* variables (e.g., $\mathsf{tvar}(\texttt{let}\,\hat{x} : \forall\vec{\alpha}.t = e_1\,\texttt{in}\,e_2) = ((\mathsf{tvar}(t)\cup\mathsf{tvar}(e_1))\setminus\vec{\alpha})\cup \mathsf{tvar}(e_2)$) which are monomorphic and, thus, cannot be instantiated. To cope with this fact all the constructions we introduced previously in this section must be enriched by a set Δ of monomorphic type variables that cannot be instantiated. So for instance the typing rule for let-expressions has Δ as extra hypothesis and becomes:

$$[\textsc{Let}]\ \frac{\Gamma;\Delta\cup\vec{\alpha}\vdash e_1 : t_1 \le t' \qquad \Gamma,x:\forall\vec{\alpha}.t_1;\Delta\vdash e_2 : t}{\Gamma;\Delta\vdash \texttt{let}\,\hat{x} : \forall\vec{\alpha}.t' = e_1\,\texttt{in}\,e_2 : t}\ \vec{\alpha}\sharp\Gamma,\Delta$$

The set Δ must also be added as a parameter of constraint generation. Furthermore, constraint generation must be modified to exploit type annotations. In particular, we want to generate different constraints for an $\hat{x}$ variable or a function when we know the type it should have. For instance, if a function $\lambda x.e$ is annotated by an intersection type $\bigwedge_{i\in I} t'_i \to t_i$, then we want to generate separate constraints from e for each arrow: we break up the intersection into the set $\{t'_i \to t_i \mid i \in I\}$ and generate a def-constraint for each element in the set. If the type in the annotation is not syntactically an intersection of arrows, we can still try to rewrite it to an equivalent intersection (as a trivial example, we could treat the annotation $(t' \to t) \vee \mathbb{0}$ like $t' \to t$). Formally, we need a function $\mathsf{d}^{\Delta}(t)$ that given a type t and a set of monomorphic variables Δ returns a set of arrow types such that, if it is not empty, then it satisfies (i) $t \simeq \bigwedge_{t'\in \mathsf{d}^{\Delta}(t)} t'$; (ii) $\mathsf{var} \bigwedge_{t'\in \mathsf{d}^{\Delta}(t)} t' \subseteq \Delta$; (iii) for all $t_1 \to t_2 \in \mathsf{d}^{\Delta}(t)$, $t_1 \not\simeq \mathbb{0}$. Essentially, $\mathsf{d}^{\Delta}(t)$ decomposes the type t into an equivalent intersection of arrow types such that these arrows are not of the form $\mathbb{0} \to s$ (which not only would be redundat but also problematic [see 49, Section 5.2.2]) and do not contain monomorphic variables. If this decomposition is not possible $\mathsf{d}^{\Delta}(t)$ returns the empty set. Once we have a function satisfying these properties (its definition is not important), then we can modify the constraint generation function so that it takes into account the monomorphic variables Δ and the annotations. The crucial modifications are the following ones.

$$
\begin{array}{ll}
\langle\!\langle \hat{x} : t \rangle\!\rangle^{\Delta} = \bigwedge_{i\in I}(\hat{x} \dot{\leq} t) & \text{if } t \simeq \bigwedge_{i\in I} t_i \\
\langle\!\langle (\lambda x.e) : t \rangle\!\rangle^{\Delta} = \exists \alpha_1, \alpha_2.(\mathsf{def}\, x\colon \alpha_1 \,\mathsf{in}\, \langle\!\langle e : \alpha_2 \rangle\!\rangle^{\Delta}) \wedge (\alpha_1 \to \alpha_2 \prec t) & \text{if } \mathsf{d}^{\Delta}(t) = \varnothing \\
\langle\!\langle (\lambda x.e) : t \rangle\!\rangle^{\Delta} = \bigwedge_{t_1\to t_2 \in \mathsf{d}^{\Delta}(t)} (\mathsf{def}\, x\colon t_1 \,\mathsf{in}\, \langle\!\langle e : t_2 \rangle\!\rangle^{\Delta}) & \text{otherwise} \\
\langle\!\langle (\mathtt{let}\, \hat{x} : \forall \vec{\alpha}.t' = e_1 \,\mathtt{in}\, e_2) : t \rangle\!\rangle^{\Delta} = \mathsf{let}\, \hat{x}\colon \forall \vec{\alpha}, \alpha[\langle\!\langle e_1 : t' \rangle\!\rangle^{\Delta\cup\vec{\alpha}} \wedge (t' \prec \alpha)] \,\mathsf{in}\, \langle\!\langle e_2 : t \rangle\!\rangle^{\Delta} &
\end{array}
$$

with the conditions $\alpha_1, \alpha_2 \sharp t, e, \Delta$ in the second line and $\alpha, \vec{\alpha} \sharp e_1, \Delta$ in the last one. If a let-abstracted variable is typed by an intersection, then we generate the constraints for each type in the intersection separately and take their conjunction. If the type of a λ-abstraction can be decomposed into an intersection of arrows, then we generate the constraints for each single arrow separately and take their conjunction; otherwise we proceed as before (in this case the type t is likely to be a type variable). For annotated let-expressions we generate the constraint that expresses the conditions under which e_1 has the type in the annotation, adding the variables $\vec{\alpha}$ to those that cannot be instantiated when typing e_1. Note that the freshness conditions now regard both Δ and the subexpressions of the program (since free type variables may occur in them). Similar modifications must be done on the freshness conditions of the remaining generation rules.

Finally, the constraint simplification rules must also take into account the set of monomorphic variables. Thus, for instance, we have to modify the simplification rule for let-abstracted variables, so that the fresh instantiation does not use variables in Δ,

and likewise for let-expressions:

$$[\hat{x}]\ \frac{}{P;\Delta \vdash (\hat{x} \dot{\leq} t) \ \rightsquigarrow\ \{t_1\{\vec{\alpha}\mapsto\vec{\beta}\} \lessdot t\} \mid M_1\{\vec{\alpha}\mapsto\vec{\beta}\} \mid \vec{\beta}}\ \begin{cases} P(\hat{x}) = \langle M_1\rangle t_1 \\ \vec{\alpha} = \mathsf{tvar}(\langle M_1\rangle t_1) \\ \vec{\beta} \sharp t, \Delta \end{cases}$$

$$[\textsc{Let}]\ \frac{\begin{array}{c} P;\Delta \cup \vec{\alpha} \vdash C_1 \ \rightsquigarrow\ D_1 \mid M_1 \mid \vec{\alpha}_1 \\ (P,\hat{x}\colon \langle M_1\sigma_1 : \alpha\sigma_1\rangle);\Delta \vdash C_2 \ \rightsquigarrow\ D_2 \mid M_2 \mid \vec{\alpha}_2 \end{array}}{P;\Delta \vdash \mathsf{let}\,\hat{x}\colon \forall\vec{\alpha},\alpha[C_1]\ \mathsf{in}\ C_2 \ \rightsquigarrow\ D_2 \mid M_1\sigma_1\{\vec{\beta}\mapsto\vec{\gamma}\} \wedge M_2 \mid \vec{\alpha}_2 \cup \vec{\beta}}\ \begin{cases} \sigma_1 \in \mathsf{tally}_{\Delta\cup\vec{\alpha}}(D_1) \\ \vec{\alpha} \sharp \Delta, M_1\sigma_1 \\ \vec{\beta} = \mathsf{tvar}(M_1\sigma_1)\backslash\Delta \\ \vec{\alpha}_1 \sharp \alpha \\ \vec{\gamma} \sharp C_1, \vec{\alpha}_2, \Delta \end{cases}$$

notice in the last rule that the appropriate set of monomorphic variables is now passed to tally, so that it will not instantiate them to solve the constraints (see Petrucciani [49] for details).

4.3.3 Occurrence typing As a final remark, notice that since the type-system in Figure 4 does not include any form of a union elimination rule, this system cannot perform occurrence typing. It is possible to proceed as in Section 4.2.5 and change the syntax of type-cases so as they specify a binding for the tested expression obtaining the same limited form of occurrence typing present in the CDuce language.

4.4 Occurrence Typing and Reconstruction of Intersections

The two systems described in the preceding sections—i.e., the explicitly-typed version and the implicitly-typed version of CDuce—present two limitations with respect to the theoretical framework of Section 4.1:

1. *No occurrence typing:* neither system includes the union elimination rule [∨] of Figure 1 which, combined with the rules [$\in_i$], implements occurrence typing.
2. *No reconstruction for intersection types:* in both systems the only way to deduce an intersection type for a function is to explicitly annotate it with the sought type.

The approach we describe next, proposed by Castagna et al. [16], targets precisely these two problems but, for the time being, at the expense of polymorphism. The work studies whether it is possible to define a type-inference algorithm for the system of the theoretical framework, *as is*: we use the language defined in (2) with the type-system defined by rules in Figure 1 and the monomorphic types of Definition 1. The technical problems to solve in order to define a typing algorithm for this system are those evoked in Sections 4.1.2 and 4.1.3, namely, (i) how to determine the arrows that form the intersection type of a λ-abstraction that is not annotated, (ii) how to deduce negation types for a function, (iii) which expressions and which occurrences of these expressions should the system choose when it applies an instance of the rule [∨], and (iv) how to determine the union of types into which the system should split the type of an expression chosen for [∨].

We have seen that the previous two systems simply avoided the technical problems (iii) and (iv) by excluding the rule [∨] and by typing type-case expressions with custom rules (possibly adding an explicit binding to the syntax of the type-cases so as to have a limited form of occurrence typing). The system described in [16], instead, follows the

opposite approach: it keeps the rule [∨] as is and introduces specific sound (though, not complete) algorithmic solutions for these two technical problems. For (iii) it virtually applies the [∨] rule to *all* subexpressions of a program and for each such subexpression it takes into account *all* its occurrences in the program. For (iv) it uses the type-cases and the applications of overloaded functions that occur in the program to determine the split in union types: for instance, if $e_1 : (\texttt{Int} \to \texttt{Char}) \wedge (\texttt{Bool} \vee \texttt{Char} \to \texttt{Bool})$, $e_2 : \texttt{Int} \vee \texttt{Bool}$ and there is in the program a type-case of the form $(e_1 e_2 \in \texttt{Bool})\,\texttt{?}\,\cdots\,\texttt{:}\,\cdots$, then the system splits the type of $e_1 e_2$ (which is $\texttt{Char} \vee \texttt{Bool}$) into two separates types, `Char` and `Bool`, since they yield different results for the type-case; but the system will also split the type of e_2 into `Int` and `Bool` since they yield two distinct result types for the application of the overloaded function e_1 (and incidentally for the type-case at issue). The same solution as for technical problem (iv) is also used for the technical problem (i), viz., given a function with a certain domain the system uses the type-cases and the applications of overloaded functions occurring in the program to determine how to split the function's domain into a union of types to be checked separately and, thus, deduce an intersection type for the function: for instance, if e_1 has the same type as above and it is applied to the parameter x of some function—e.g., $\lambda x.\ldots e_1(x)\ldots$—, then the system will deduce that the domain of the function is (a subtype of) $\texttt{Int} \vee \texttt{Bool} \vee \texttt{Char}$ and split this domain in two, that is, it tries to type the body of the function under the hypothesis $x : \texttt{Int}$ and under the hypothesis $x : \texttt{Bool} \vee \texttt{Char}$ to deduce for the function a type of the form $(\texttt{Int} \to \ldots) \wedge (\texttt{Bool} \vee \texttt{Char} \to \ldots)$. Finally, the system in Castagna et al. [16] avoids technical problem (ii) in the same way as implicit CDuce does: negation types are not inferred, but type-cases cannot test functional types other than $\mathbb{0} \to \mathbb{1}$. This of course implies that property (3) in Sectionn 4.1.1—i.e., that every value has a type or its negation—does not hold. But this does not hinder the property of type preservation since, as we explained in Section 4.1.1, the presence of the union elimination rule [∨] suffices for it (even though it holds only for *ad hoc* parallel reductions: cf. Barbanera et al. [1] and Castagna et al. [16]).

To obtain a type-inference algorithm with the characteristics outlined above, Castagna et al. [16] proceed in four steps, that we describe next.

First, we introduce an intermediate language that adds to the theoretical framework's original language defined in (2) (henceforth, the *source language*) a "bind" construct that factors out common subexpressions. The type system of this new intermediate language limits the introduction of intersection and union types in the rules for typing functions and bind forms, respectively. Typeability in the source and the intermediate language coincide up to refactoring with bind.

Second, we introduce a syntactic restriction on terms of the intermediate language dubbed *maximal-sharing canonical form* (MSC-form), reminiscent of an aggressive A-normal form [55]. A MSC-form is essentially a list of bindings from variables to *atoms*. An atom is either an expression of our source language in which all subexpressions are variables, or it is a λ-abstraction whose body is a MSC-form. These forms are called *maximal-sharing* forms because they must satisfy the property that there cannot be two distinct bindings for the same atom. This is a crucial property because it ensures that every expression of the source language (i) is equivalent to a unique (modulo the order of bindings) MSC-form and (ii) is well-typed if and only if its MSC-form is. For

instance, consider the expression

$$(a_1 a_2 \in \texttt{Int})\,\texttt{?}\,(a_2 + 1)\,\texttt{:}\,((a_1 a_2)\,@\,a_2) \tag{8}$$

where a_1 and a_2 are generic atoms of type $t_1 = (\texttt{Int}\to\texttt{Int}) \wedge (\texttt{String}\to\texttt{String})$ and $t_2 = \texttt{Int} \vee \texttt{String}$, respectively, and `@` denotes string concatenation. This expression is well-typed with type `Int`∨`String`. Its MSC-form will look like the term in Table 1. Notice that this term satisfies the maximal sharing property because the two occurrences of the application $a_1 a_2$ in the source language expression (8) are bound by the same variable x_3. Essentially MSC-forms are our solution to technical problem (*iii*) we evoked at the beginning of this section, namely, which subexpressions and which occurrences of these subexpressions should the system choose for applying [∨]: the fact that all proper subexpressions of an atom are variables means that the system chooses *all* subexpressions, while the maximal sharing property means that the system chooses *all* occurrences of each subexpression since it replaces all of them by the same variable.

```
bind x1 = a1 in
bind x2 = a2 in
bind x3 = x1x2 in
bind x4 = x2 + 1 in
bind x5 = x3@x2 in
bind x6 = (x3∈Int)?x4 : x5
  in x6
```

Table 1: Pure MSC-form

```
bind x1 : {t1} = a1 in
bind x2 : {Int, String} = a2 in
bind x3 : {x2:Int▷Int, x2:String▷String} = x1x2 in
bind x4 : {Int} = x2 + 1 in
bind x5 : {String} = x3@x2 in
bind x6 : {t2} = (x3∈Int)?x4 : x5
  in x6
```

Table 2: Annotated MSC-form

Third, we prove that an MSC-form is well-typed if and only if it is possible to explicitly annotate all the bindings of variables so that the MSC-form type-checks. The annotations essentially define how to split the type of the bound variables into a union of types (when the variable is bound by a λ this corresponds to splitting the type of the λ-abstraction into an intersection, when the variable is bound by a bind this corresponds to splitting the argument of the bind-expression into a union) and the annotated MSC-form type-checks if the rest of the expression type-checks for each of the splits specified in its annotations. Table 2 gives the annotations for the MSC-form of Table 1. The important annotations are those of the variables x_2 and x_3. The first states that to type the expression, the type `Int`∨`String` of a_2 must be split and the expression must be checked separately for x_2 : `Int` and x_2 : `String`. The annotation of x_3 states that when x_2 has type `Int` then x_3 must be assumed to be of type `Int` and when x_2 has type `String` so must have x_3. Since we can effectively transform a source language expression into its MSC-form, then we have a method to check the well-typedness of an expression of the source language: transform it into its MSC-form and infer all the annotations of its variables, if possible. Inferring the annotations of a MSC-form boils down to deciding how to split the types of its variables.

Fourth, we define an algorithm which infers how to split the types of atoms. It starts from a MSC-form in which all variables are annotated with the top type `Any` and performs several passes to refine these annotations. Each pass has three possible outcomes: either (*a.*) the MSC-form type-checks with its current annotations and the algorithm stops with a success, or (*b.*) the MSC-form does not type-check, the pass

proposes a new version of the same MSC-form but with refined annotations, and a new pass is started, or $(c.)$ the MSC-form does not check and it is not possible to further refine the annotations so that the form may become typable, then the algorithm stops with a failure. The algorithm refines the annotations differently for variables that are bound by lambdas and by binds. For the variables in binds the algorithm produces a set of disjoint types so that their union is the type of the atom in the bind; for lambdas the algorithm splits the type of the parameter into a set of disjoint types and rejects the types in this set for which the function does not type-check, thus determining the domain of the function. The very last point that remains to explain is how to determine the split of a type: as a matter of fact, in general there are infinitely many different ways to split a type. The split of the types is driven by the types tested in type-cases and the operations applied to their components. For instance, the split of the type of a_2 for the variable x_2 in Tables 1 and 2 is determined by the test $x_3{\in}\texttt{Int}$: the algorithm will propose to split the type t_3 of x_3 into $t_3 \wedge \texttt{Int}$ and $t_3 \wedge \neg\texttt{Int}$. Since t_3 is $\texttt{Int}\vee\texttt{String}$, the split proposed for x_3 is actually $\texttt{Int}$ or $\texttt{String}$. This split triggers in the subsequent pass the split for the type of x_2 since x_3 is defined as x_1x_2 and x_3 can be of type $\texttt{Int}$ only if x_2 is of type $\texttt{Int}$ and it can be of type $\texttt{String}$ only if x_2 is of type $\texttt{String}$. We just got the expected annotations. Essentially, this fourth step is our solution to the technical problems (iv) and (i) we evoked at the beginning of this section, namely, how to split the type of a subexpression chosen to apply [∨] into a union of types, and how to split the type of an implicitly-typed function into an intersection of arrows: we split these types by analyzing the type-cases and the overloaded function applications occurring in the program.

Formally, [16] defines the following intermediate language

$$\textbf{Intermediate exp } \boldsymbol{e} ::= c \mid x \mid \lambda x.\boldsymbol{e} \mid \boldsymbol{e}\boldsymbol{e} \mid (\boldsymbol{e},\boldsymbol{e}) \mid \pi_i\boldsymbol{e} \mid (\boldsymbol{e}{\in}\tau)\,\texttt{?}\,\boldsymbol{e}\,\texttt{:}\,\boldsymbol{e} \mid \texttt{bind}\,x\texttt{=}\boldsymbol{e}\,\texttt{in}\,\boldsymbol{e} \quad (9)$$

with the typing rules given in Figure 7 (where we omitted the rules for constants, variables, and pairs since they are the same as in Figure 1).[25] A well-typed expression of

$$[{\to}\mathrm{I}]\ \frac{(\forall j \in J) \qquad \Gamma, x : t_j \vdash_{\mathscr{I}} \boldsymbol{e} : s_j}{\Gamma \vdash_{\mathscr{I}} \lambda x.\boldsymbol{e} : \bigwedge_{j\in J} t_j \to s_j}\ J \neq \varnothing \qquad [{\to}\mathrm{E}]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e}_1 : t_1 \quad \Gamma \vdash_{\mathscr{I}} \boldsymbol{e}_2 : t_2}{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e}_1\boldsymbol{e}_2 : t_1 \circ t_2}\ \begin{array}{l} t_1 \leq \mathbb{0} \to \mathbb{1} \\ t_2 \leq \mathsf{dom}(t_1) \end{array}$$

$$[{\times}\mathrm{E}_1]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e} : t \leq (\mathbb{1} \times \mathbb{1})}{\Gamma \vdash_{\mathscr{I}} \pi_1\boldsymbol{e} : \pi_1(t)} \qquad [{\times}\mathrm{E}_2]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e} : t \leq (\mathbb{1} \times \mathbb{1})}{\Gamma \vdash_{\mathscr{I}} \pi_2\boldsymbol{e} : \pi_2(t)} \qquad [\mathbb{0}]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e} : \mathbb{0}}{\Gamma \vdash_{\mathscr{I}} (\boldsymbol{e}{\in}t)\,\texttt{?}\,\boldsymbol{e}_1\,\texttt{:}\,\boldsymbol{e}_2 : \mathbb{0}}$$

$$[{\in}_1]\ \frac{\Gamma \vdash \boldsymbol{e} : t_0 \leq t \qquad \Gamma \vdash \boldsymbol{e}_1 : t_1}{\Gamma \vdash (\boldsymbol{e}{\in}t)\,\texttt{?}\,\boldsymbol{e}_1\,\texttt{:}\,\boldsymbol{e}_2 : t_1}\ t_0 \not\simeq \mathbb{0} \qquad [{\in}_2]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e} : t_0 \leq \neg t \qquad \Gamma \vdash \boldsymbol{e}_2 : t_2}{\Gamma \vdash_{\mathscr{I}} (\boldsymbol{e}{\in}t)\,\texttt{?}\,\boldsymbol{e}_1\,\texttt{:}\,\boldsymbol{e}_2 : t_2}\ t_0 \not\simeq \mathbb{0}$$

$$[\vee_1]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e}_2 : s}{\Gamma \vdash_{\mathscr{I}} \texttt{bind}\,x\texttt{=}\boldsymbol{e}_1\,\texttt{in}\,\boldsymbol{e}_2 : s}\ x{\notin}\mathsf{dom}(\Gamma) \qquad [\vee_2]\ \frac{\Gamma \vdash_{\mathscr{I}} \boldsymbol{e}_1 : \bigvee_{j\in J} t_j \quad {}^{(\forall j\in J)}\ \Gamma, x{:}t_j \vdash_{\mathscr{I}} \boldsymbol{e}_2 : s_j}{\Gamma \vdash_{\mathscr{I}} \texttt{bind}\,x\texttt{=}\boldsymbol{e}_1\,\texttt{in}\,\boldsymbol{e}_2 : \bigvee_{j\in J} s_j}\ J{\neq}\varnothing$$

Fig. 7: Intermediate typing rules

[25] Notice that we do not define a reduction semantics for the intermediate language since the sole purpose of the intermediate expressions is to encode typing derivations. But a call-by-need semantics for the new bind-expressions would be appropriate [16, see Appendix A.6].

the intermediate language is typed by derivations in which every instance of the [∨] rule (here declined in two forms) corresponds to a bind-expression. Any such derivation corresponds to a canonical derivation (Figure 2) for a particular expression of the source language in (2). This expression can be obtained from the intermediate language expression by unfolding its bindings. Formally, this is obtained by the *unwinding* operation, noted $\lceil . \rceil$ and defined for binding expressions as $\lceil \mathtt{bind}\, x = \boldsymbol{e}_1 \,\mathtt{in}\, \boldsymbol{e}_2 \rceil \stackrel{\text{def}}{=} \lceil \boldsymbol{e}_2 \rceil \{ \lceil \boldsymbol{e}_1 \rceil / x \}$, as the identity for constants and variables, and homomorphically for all the other expressions. It is possible to prove that the problem of typing an expression of our source language is equivalent to the problem of finding a typable intermediate expression whose unwinding is that declarative expression. In other terms, a declarative expression is typable if and only if we can enrich it with bindings so that it becomes a typable intermediate expression.

The definition of the intermediate expressions is a step forward in solving the problem of typing a declarative expression, but it also brings a new problem, since we now have to decide where to add the bindings in a declarative expression so as to make it typable in the intermediate system. We get rid of this problem by defining the *maximal sharing canonical forms* (*MSC-form* for short). The idea is pretty simple, and consists in adding a new binding for every *distinct* (modulo α-conversion) sub-expressions of a declarative expression. Formally, this transformation yields a MSC-form:

Definition 5 (MSC Forms). *An intermediate expression* $\boldsymbol{e}$ *is a* maximal sharing canonical form *if it is produced by the following grammar:*

$$\begin{array}{ll} \textbf{\textit{Atomic expressions}} & \boldsymbol{a} ::= c \mid \lambda x.\boldsymbol{\kappa} \mid (x,x) \mid xx \mid (x{\in}\tau)\,?\,x : x \mid \pi_i x \\ \textbf{\textit{MSC-forms}} & \boldsymbol{\kappa} ::= x \mid \mathtt{bind}\, x = \boldsymbol{a}\, \mathtt{in}\, \boldsymbol{\kappa} \end{array} \tag{10}$$

and is α*-equivalent to an expression* $\boldsymbol{\kappa}$ *that satisfies the following properties: (1) if* $\mathtt{bind}\, x_1 = \boldsymbol{a}_1\, \mathtt{in}\, \boldsymbol{\kappa}_1$ *and* $\mathtt{bind}\, x_2 = \boldsymbol{a}_2\, \mathtt{in}\, \boldsymbol{\kappa}_2$ *are distinct sub-expressions of* $\boldsymbol{\kappa}$*, then* $\lceil \boldsymbol{a}_1 \rceil \not\equiv_\alpha \lceil \boldsymbol{a}_2 \rceil$*; (2) if* $\lambda x.\boldsymbol{\kappa}_1$ *is a sub-expression of* $\boldsymbol{\kappa}$ *and* $\mathtt{bind}\, y = \boldsymbol{a}\, \mathtt{in}\, \boldsymbol{\kappa}_2$ *a sub-expression of* $\boldsymbol{\kappa}_1$*, then* $fv(\boldsymbol{a}) \not\subseteq fv(\lambda x.\boldsymbol{\kappa}_1)$*; (3) if* $\mathtt{bind}\, x = \boldsymbol{a}\, \mathtt{in}\, \boldsymbol{\kappa}'$ *is a sub-expression of* $\boldsymbol{\kappa}$*, then* $x \in fv(\boldsymbol{\kappa}')$.

MSC-forms, ranged over by $\boldsymbol{\kappa}$, are variables preceded by a list of bindings of variables to atoms. Atoms are either λ-abstractions whose body is a MSC-form or any other expression in which all proper sub-expressions are variables. Therefore, bindings can appear in a MSC-form either at top-level or at the beginning of the body of a function. Definition 5 ensures that given an expression e of the source language (2) there exists a unique (modulo α-conversion and the order of bindings) MSC-form whose unwinding is e: we denote this MSC-form by $\mathsf{MSC}(e)$ and it is easy to effectively produce it from e (roughly, visit e bottom up and generate a distinct binding for each distinct sub-expression). Furthermore, e is typable if and only if $\mathsf{MSC}(e)$ is: we reduced the problem of typing e to the one of typing $\mathsf{MSC}(e)$, a form that we can effectively produce from e and for which we have the syntax-directed type system of Figure 7.

The type system of Figure 7 is syntax directed, but it still includes non-analytic rules for functions and bind-expressions. Thus, the next step consists in adding annotations to intermediate expressions, so as to make these rules analytic: we consider expressions of the form $\lambda x{:}A.\boldsymbol{e}$ and $\mathtt{bind}\, x{:}A = \boldsymbol{e}\, \mathtt{in}\, \boldsymbol{e}$, where A ranges over annotations of the form $\{\Gamma \triangleright t, \ldots, \Gamma \triangleright t\}$. Our annotations are, thus, finite relations between type environments and types. An annotation of the form $x : \{\Gamma_i \triangleright t_i\}_{i \in I}$ indicates that under the

hypothesis Γ_i the variable x must be supposed to be of type t_i. We write $\{t_1,\ldots,t_n\}$ for $\{\varnothing\triangleright t_1,\ldots,\varnothing\triangleright t_n\}$ and just t for $\{\varnothing\triangleright t\}$. So for instance we write $\lambda x{:}t.\boldsymbol{e}$ for $\lambda x{:}\{\varnothing\triangleright t\}.\boldsymbol{e}$ while, say, $\mathtt{bind}\,x{:}\{t_1,\ldots,t_n\}\mathtt{=}\boldsymbol{e}_1\,\mathtt{in}\,\boldsymbol{e}_2$ stands for $\mathtt{bind}\,x{:}\{\varnothing\triangleright t_1,\ldots,\varnothing\triangleright t_n\}\mathtt{=}\boldsymbol{e}_1\,\mathtt{in}\,\boldsymbol{e}_2$.

In this system terms encode derivations. Terms with simple annotations such as $\lambda x{:}t.\boldsymbol{e}$ represent derivations as they can be found in the simply-typed λ-calculus: in other terms, to type the function the system must look for a type s such that $\lambda x{:}t.\boldsymbol{e}$ is of type $t\to s$. When annotations are sets of types, such as in $\lambda x{:}\{t_1,\ldots,t_n\}.\boldsymbol{e}$, then the term represents a derivation for an intersection type, such as the derivations that can be found in semantic subtyping calculi: in other terms, to type the function the system look for a set of types $\{s_1,\ldots,s_n\}$ such that $\lambda x{:}\{t_1,\ldots,t_n\}.\boldsymbol{e}$ has type $\bigwedge_{i=1}^n t_i\to s_i$. Finally, the reason why we need the more complex annotations of the form $\{\Gamma_1\triangleright t_1,\ldots,\Gamma_1\triangleright t_1\}$ can be shown by an example. Consider $\lambda x.((\lambda y.(x,y))x)$: in the declarative system we can deduce for it the type $(\mathtt{Int}\to\mathtt{Int}\times\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}\times\mathtt{Bool})$. We must find the annotations A_1 and A_2 such that $\lambda x{:}A_1.((\lambda y{:}A_2.(x,y))x)$ has type $(\mathtt{Int}\to\mathtt{Int}\times\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}\times\mathtt{Bool})$. Clearly $A_1=\{\mathtt{Int},\mathtt{Bool}\}$. However, the typing of the parameter y depends on the typing of x: when $x{:}\mathtt{Int}$ then y must have type $\mathtt{Int}$ (the type of y must be larger than the one of x—the argument it will be bound to—, but also smaller than $\mathtt{Int}$ so as to deduce that $\lambda y.(x,y)$ returns a pair in $\mathtt{Int}\times\mathtt{Int}$). Likewise when $x{:}\mathtt{Bool}$, then y must be of type $\mathtt{Bool}$, too. Therefore, we use as A_2 the annotation $\{x{:}\mathtt{Int}\triangleright\mathtt{Int},x{:}\mathtt{Bool}\triangleright\mathtt{Bool}\}$, which precisely states that when $x{:}\mathtt{Int}$, then we must suppose that y (the variable annotated by A_2) is of type $\mathtt{Int}$, and likewise for $\mathtt{Bool}$.

$$[{\to}\mathrm{I}]\ \frac{(\forall j\in J)\qquad \Gamma,x:t_j\vdash_{\mathscr{A}}\kappa:s_j}{\Gamma\vdash_{\mathscr{A}}\lambda x{:}\{\Gamma_i\triangleright t_i\}_{i\in I}.\kappa:\bigwedge_{j\in J}t_j\to s_j}\ J=\{i\in I\mid\Gamma\le\Gamma_i\}\neq\varnothing$$

$$[\vee_1]\ \frac{\Gamma\vdash_{\mathscr{A}}\kappa:s}{\Gamma\vdash_{\mathscr{A}}\mathtt{bind}\,x{:}\{\Gamma_i\triangleright t_i\}_{i\in I}\mathtt{=}a\,\mathtt{in}\,\kappa:s}\ \begin{array}{c}x\notin\mathsf{dom}(\Gamma)\\ \{i\in I\mid\Gamma\le\Gamma_i\}=\varnothing\end{array}$$

$$[\vee_2]\ \frac{\Gamma\vdash_{\mathscr{A}}a:\bigvee_{j\in J}t_j\qquad(\forall j\in J)\qquad\Gamma,x:t_j\vdash_{\mathscr{A}}\kappa:s_j}{\Gamma\vdash_{\mathscr{A}}\mathtt{bind}\,x{:}\{\Gamma_i\triangleright t_i\}_{i\in I}\mathtt{=}a\,\mathtt{in}\,\kappa:\bigvee_{j\in J}s_j}\ J=\{i\in I\mid\Gamma\le\Gamma_i\}\neq\varnothing$$

Fig. 8: Algorithmic typing rules

The type system for annotated terms is given by the rules for abstractions and binding in Figure 8 plus all the other rules of the intermediate type system (specialized for MSC-forms, i.e., where every subexpression is a variable). The system is algorithmic since it is syntax-directed and uses only analytic rules.

The main interest of this algorithmic system is that a well-typed annotated term univocally encodes a type derivation for a MSC-form and, therefore, it also encodes a particular canonical derivation for an expression of the source language. All this gives us a procedure to check whether an expression e of the source language (2) is well typed or not: produce $\mathsf{MSC}(e)$ and look for a way to annotate it so that it becomes a well-typed annotated expression. If we find such annotations, then e is well typed. If such annotations do not exist, then e is not well-typed.

The last final step is then to define an algorithm to find whether there exists a way to annotate an MSC-form to make it well-typed. Different algorithms are possible. Castagna et al. [16, Section 5] describe an algorithm that starts by annotating all

bound variables with `Any` and then performs several passes in which it analyses the type-cases and overloaded applications of the term to determine how to split the types of the concerned expression and thus refining the annotations of their bindings. The reader may refer to [16, Section 5] for the details of this algorithm. Here we just stress that, contrary to the previous systems presented here, the algorithm is able to deduce the precise intersection types for the (non-annotated) functions `not_`, `and_` (both versions), and `or_` we gave at the end of the introduction, as well as reconstruct the type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ for the function $\lambda x.(x\in\texttt{Int})\,?\,(x+1)\,:\,\neg x$ given in Section 2.

4.5 Summary

In this section we presented three practical variations of the theoretical language we defined in Section 4.1. The ultimate goal of our research is to have a unique language that covers the characteristics of the three of them. The current implementation of polymorphic CDuce corresponds to the language we presented in Section 4.2.5 but work is in progress to merge it with the implicitly-typed language of Petrucciani's dissertation [49, Part I] that we presented in Section 4.3. The idea is to completely move to the constraint generation and resolution we surveyed in Section 4.3.1 and consider the current version of polymorphic CDuce as the special case of the annotated expressions presented in Section 4.3.2. This may require to change the notation of explicitly-typed polymorphic functions, which is the reason why a polymorphic version of CDuce was not released, yet. The only problem to solve to obtain a conservative extension of (both monomorphic and polymorphic) CDuce will then be how to deal in Petrucciani's system with the type-cases of values with functional components (since in CDuce you can test whether a function has a given arrow type). The final step will be then to integrate the resulting system with the general usage of the union elimination rules on the lines of the system we described in Section 4.4.

5 Further Features

In this section we briefly overview few extra features that were developed in the context of the study of set-theoretic types.

5.1 Pattern Matching

Several examples presented in this article use pattern matching. Furthermore, in Section 2 we cited the precise typing of pattern matching expressions as one of the main motivations of using set-theoretic types. However, the languages we formalized in Section 4 do not include pattern matching expressions: they just have type-case expressions which, in their binding variant of Section 4.2.3, may be considered a very simplistic version of pattern matching. Here, we outline how full-fledged pattern matching can be added to the languages presented in Section 4. Similar formalizations of pattern matching for set-theoretic type systems have been described by Frisch [22, Chapter 6], Castagna et al. [12, Appendix E], and Castagna et al. [13].

For simplicity, we only consider two-branch pattern matching. We extend the syntax with the match construct and with patterns:

$$e ::= \cdots \mid \mathtt{match}\, e\, \mathtt{with}\, p \to e \mid p \to e \qquad p ::= \tau \mid x \mid (p,p) \mid p \,\&\, p \mid p \mid p\,,$$

where τ are the test types defined in Section 4 (ground types for CDuce and its explicit polymorphic variant, ground non-functional types for implicitly-typed CDuce and the variant for occurrence typing) and with some restrictions on the variables that can appear in patterns: in (p_1,p_2) and $p_1 \,\&\, p_2$, p_1 and p_2 must have distinct variables; in $p_1 \mid p_2$, p_1 and p_2 must have the same variables.

A more familiar syntax for patterns is $p ::= \_ \mid c \mid x \mid (p,p) \mid p \,\mathsf{as}\, x \mid p \mid p$, with wildcards and constants instead of τ types and with as-patterns "$p \,\mathsf{as}\, x$" (in OCaml syntax; $x\mathtt{@}p$ in Haskell) instead of conjunction. We can encode $\_$ and c as $\mathbb{1}$ and $\boldsymbol{b}_c$ (both are in the grammar for τ), while "$p \,\mathsf{as}\, x$" is $p \,\&\, x$, as will soon be clear.

$$\begin{aligned}
v/x &= \{v/x\} \\
v/\tau &= \{\} && \text{if } v \in \tau \\
v/(p_1,p_2) &= \varsigma_1 \cup \varsigma_2 && \text{if } v = (v_1,v_2),\ v_1/p_1 = \varsigma_1, \text{ and } v_2/p_2 = \varsigma_2 \\
v/p_1 \,\&\, p_2 &= \varsigma_1 \cup \varsigma_2 && \text{if } v/p_1 = \varsigma_1 \text{ and } v/p_2 = \varsigma_2 \\
v/p_1 \mid p_2 &= v/p_1 && \text{if } v/p_1 \neq \mathsf{fail} \\
v/p_1 \mid p_2 &= v/p_2 && \text{if } v/p_1 = \mathsf{fail} \\
v/p &= \mathsf{fail} && \text{otherwise}
\end{aligned}$$

Fig. 9: Semantics of patterns

To describe the semantics of pattern matching, we define a function $(\cdot)/(\cdot)$ that, given a value v and a pattern p, yields a result v/p which is either fail or a substitution ς mapping the variables in p to values (subterms of v). This function is defined in Figure 9. Then, we augment the reduction rules with

$$\begin{aligned}
(\mathtt{match}\, v\, \mathtt{with}\, p_1 \to e_1 \mid p_2 \to e_2) &\rightsquigarrow e_1\varsigma && \text{if } v/p_1 = \varsigma \\
(\mathtt{match}\, v\, \mathtt{with}\, p_1 \to e_1 \mid p_2 \to e_2) &\rightsquigarrow e_2\varsigma && \text{if } v/p_1 = \mathsf{fail} \text{ and } v/p_2 = \varsigma
\end{aligned}$$

and add $\mathtt{match}\, E\, \mathtt{with}\, p \to e \mid p \to e$ to the grammar of evaluation contexts.

Given each pattern p, we can define a type $\lbag p \rbag$ that describes exactly the values that match the pattern:

$$\lbag \tau \rbag = \tau \qquad \lbag x \rbag = \mathbb{1}$$
$$\lbag (p_1,p_2) \rbag = \lbag p_1 \rbag \times \lbag p_2 \rbag \qquad \lbag p_1 \,\&\, p_2 \rbag = \lbag p_1 \rbag \wedge \lbag p_2 \rbag \qquad \lbag p_1 \mid p_2 \rbag = \lbag p_1 \rbag \vee \lbag p_2 \rbag$$

It can be shown that, for every well-typed value v and every pattern p, we have $v/p \neq \mathsf{fail}$ if and only if $\varnothing \vdash v : \lbag p \rbag$. This allows us to formalize purely at the level of types the exhaustiveness and redundancy checks that are often performed on pattern matching. The typing rule for match is the following.

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : t_0 \\ \text{either } t_0 \leq \neg \lbag p_1 \rbag \text{ or } \Gamma, (t_0 \wedge \lbag p_1 \rbag)/p_1 \vdash e_1 : t \\ \text{either } t_0 \leq \lbag p_1 \rbag \text{ or } \Gamma, (t_0 \setminus \lbag p_1 \rbag)/p_2 \vdash e_2 : t\end{array}}{\Gamma \vdash \mathtt{match}\, e_0\, \mathtt{with}\, p_1 \to e_1 \mid p_2 \to e_2 : t}\ t_0 \leq \lbag p_1 \rbag \vee \lbag p_2 \rbag$$

The "either … or …" conditions have the same purpose as for type-case rule [CASE] in Figure 4, namely, they skip the typing of branches that cannot be selected. The side condition $t_0 \leq \lceil p_1 \rfloor \vee \lceil p_2 \rfloor$ ensures that matching is exhaustive: any value produced by e_0 has type t_0 and therefore matches either p_1 or p_2. When a branch is selectable it is typed under the hypothesis Γ extended with a type environment produced by applying the operator t/p, which given a type t and a pattern p with $t \leq \lceil p \rfloor$ produces the type environment that can be assumed for the variables in p when a value of type t is matched against p and matching succeeds. Thus e_1 is typed under the hypothesis obtained supposing that p_1 was matched against a value produced by e_0 (i.e., in t_0) and accepted by p_1 (i.e., in $\lceil p_1 \rfloor$), while the hypotheses for e_2 are obtained supposing that p_2 was matched against a value produced by e_0 (i.e., in t_0) and *not* accepted by p_1 (i.e., in $\neg \lceil p_1 \rfloor$: remind that $t_0 \setminus \lceil p_1 \rfloor = t_0 \wedge \neg \lceil p_1 \rfloor$). The operator is defined as follows

$$\begin{aligned} t/\tau &= \varnothing \\ t/x &= x : t \\ t/(p_1, p_2) = t/p_1 \,\&\, p_2 &= (t/p_1) \cup (t/p_2) \\ t/p_1 \mid p_2 &= ((t \wedge \lceil p_1 \rfloor)/p_1) \cup ((t \setminus \lceil p_1 \rfloor)/p_2) \end{aligned}$$

and satisfies the property that for every t, p, and v, if $\varnothing \vdash v : t$ and $v/p = \varsigma$, then, for every variable x in p, the judgment $\varnothing \vdash x\varsigma : (t/p)(x)$ holds.

Finally, we said that the condition $t_0 \leq \lceil p_1 \rfloor \vee \lceil p_2 \rfloor$ in the typing rule for **match**-expressions ensures the exhaustiveness of pattern matching, but what about redundancy? When $t_0 \leq \neg \lceil p_1 \rfloor$ should not the system return a warning that e_1 cannot be selected and likewise for e_2 when $t_0 \leq \lceil p_1 \rfloor$? In general it should not, since skipping the typing of some branches is necessary for inferring intersection types for overloaded functions. For instance, consider again the function that we defined in Section 2 $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} x.\,(x \in \texttt{Int})\,?\,(x+1)\,:\,\neg x$ whose definition with pattern matching would be:

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} x.\,\texttt{match}\, x\, \texttt{with}\, \texttt{Int} \to (x+1) \mid \mathbb{1} \to \neg x$$

When typing the body of the function under the hypothesis $x : \texttt{Int}$ it is important not to check the type of the second branch (since $\neg x$ would be ill typed) and under the hypothesis $x : \texttt{Bool}$ it is important not to check the type of the first branch (since $x+1$ would be ill typed). However, neither of the branches is redundant because each of them is type-checked *at least once*. Redundancy corresponds to branches that are never type-checked, as, for instance, the second branch in the following definition

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} x.\,\texttt{match}\, x\, \texttt{with}\, (\texttt{Int} \vee \texttt{Bool}) \to x \mid \mathbb{1} \to \neg x$$

which is skipped both under the hypothesis $x : \texttt{Int}$ and under the hypothesis $x : \texttt{Bool}$ and, therefore, must be fingered as redundant (but the function is well-typed). In conclusion, as it is the case for exhaustiveness, redundancy of pattern matching, too, can be characterized in terms of a type system that includes set-theoretic types.

5.2 Gradual Typing

Gradual typing is an approach proposed by Siek and Taha [56] to combine the safety guarantees of static typing with the programming flexibility of dynamic typing. The idea

is to introduce an *unknown* (or *dynamic*) type, denoted $?$, used to inform the compiler that some static type-checking can be omitted, at the cost of some additional runtime checks. The use of both static typing and dynamic typing in a same program creates a boundary between the two, where the compiler automatically adds—often costly [58]—dynamic type-checks to ensure that a value crossing the barrier is correctly typed.

Occurrence typing—that we discussed in Sections 2 and 4.4—and gradual typing often have common use cases. For instance the example we gave for occurrence typing in Section 2, $\lambda x.(x{\in}\mathtt{Int})\,?\,(x+1)\,:\,{\neg}x$, can also be typed by gradual typing as follows:

$$\lambda x : ?.(x{\in}\mathtt{Int})\,?\,(x+1)\,:\,{\neg}x \tag{11}$$

"Standard" or "safe" gradual typing inserts two dynamic checks since it compiles the code above into $\lambda x : ?.(x{\in}\mathtt{Int})\,?\,(\mathtt{x}\langle\mathtt{Int}\rangle+1)\,:\,{\neg}(\mathtt{x}\langle\mathtt{Bool}\rangle)$, where $e\langle t\rangle$ is a type-cast that dynamically checks whether the value returned by e has type t.[26] The type deduced for the function in (11) is $? \to \mathtt{Int}{\vee}\mathtt{Bool}$ meaning that it is a function that can be applied to any argument (which may have its type dynamically checked if needs to be) and will return either an integer or a Boolean (or a cast exception if a dynamic check fails). This type is not very precise since it allows the function to be applied to any argument, even if we already know that it will fail with a cast exception for arguments that are neither integers nor Booleans. Whence the interest of having full-fledged set-theoretic types thanks to which the programmer can shrink the domain of the function as follows:

$$\lambda x : (?\wedge(\mathtt{Int}{\vee}\mathtt{Bool})).(x{\in}\mathtt{Int})\,?\,(x+1)\,:\,{\neg}x \tag{12}$$

Intuitively, this annotation means that the function above accepts for x a value of any type (which is indicated by $?$), as long as this value is also either an `Int` or a `Bool`. So the type-casts will never fail. This was the initial motivation of our study of integrating gradual and set-theoretic types [8, 15]. Of course, the example above does not need gradual typing if the systems provides occurence typing: this provides a better solution since, as we showed before, it returns a more precise type ($(\mathtt{Int}{\to}\mathtt{Int})\wedge(\mathtt{Bool}{\to}\mathtt{Bool})$) and avoids the insertion of superfluous run-time checks. But there are some cases in which the occurrence typing analysis may fail to type-check, since either they are too complex or they are not covered by the formalism (e.g., when polymorphic types are needed, which are not captured by the system presented in Section 4.4). In those cases gradual typing is a viable alternative to no typing at all. In a sense, occurrence typing is a discipline designed to push forward the frontiers beyond which gradual typing is necessary, thus reducing the amount of runtime checks needed (see Castagna et al. [16, Section 3.3] for more a detailed treatment).

But the interest of integrating gradual and set-theoretic types is not limited to having a more precise typing of some applications like the ones above. The main interest of this integration is that the introduction of set-theoretic types allows us to give a semantic foundation to gradual typing, and explain the type-theory of gradual types only in terms of non-gradual ones. The core of the type-systems for gradually-typed expressions such

[26] Intuitively, $e\langle t\rangle$ is syntactic sugar for, say, in JavaScript `(typeof(`e`)==="`t`")?`e`:(throw "Type error")`. Not exactly though, since to implement compilation *à la* sound gradual typing it is necessary to use casts on function types that need special handling.

as (11) or (12) is the definition of a *precision* relation $\preccurlyeq$ on types [57, 25] (called *naive subtyping* in [62]). In the cited works the definition of this relation is very simple: given two gradual types τ_1, τ_2, the type τ_1 is less precise than τ_2, written $\tau_1 \preccurlyeq \tau_2$, if and only if τ_2 is obtained from τ_1 by replacing some occurrences of $?$ by some types (we use τ to range over *gradual types* to distinguish them from types in which $?$ does not occur and that are called *static types*). So for instance $?\to?\times? \preccurlyeq \mathtt{Int}\to?\times\mathtt{Bool} \preccurlyeq \mathtt{Int}\to\mathtt{Int}\times\mathtt{Bool}$. Intuitively, the precision relation indicates in which types a gradual type may "materialize" (i.e., turn out to be) at runtime. So the type $?\to?$ of a function materializes into $\mathtt{Int}\to?$ if this function happens at runtime to be applied to an integer. Castagna et al. [15] demonstrate that to extend with gradual typing an existing statically-typed language all is needed is (i) to add $?$ to the types (as a new basic type), (ii) define the precision relation $\preccurlyeq$ (and, if used, the subtyping relation $\leq$) on the new types, and (iii) add the subsumption-like materialization rule here on the right to the existing typing rules. Of course, this does not immediately yield an effective implementation (one has to find a type-inference algorithm, define the language with the explicit casts and the compilation of well-typed terms into it, compilation that, roughly, must insert a dynamic type-cast wherever the typing algorithm had to use a materialization rule) but conceptually this is all is needed. The difficult point is to define the precision relation (but also to extend an existing subtyping relation to gradual types). The simple definition of the precision relation we gave above is syntax based and, as such, it shows its limits as soon as we add type connectives. In semantic subtyping equivalence between types plays a central role: two types are equivalent if and only if they represent the same set of values, and this makes them to behave identically in every context. So one would expect equivalent types to materialize in the same set of types, but this is not the case: consider for example the types $\mathtt{Int}\vee?$ and $?\vee\mathtt{Int}$; although they are equivalent, the former materializes into $\mathtt{Int}\vee\mathtt{Bool}$ while the latter does not. The latter does, however, materialize into $\mathtt{Bool}\vee\mathtt{Int}$ which is equivalent to $\mathtt{Int}\vee\mathtt{Bool}$. A similar reasoning can be done for $?$ and $\neg?$ which intuitively behave in exactly the same way. We thus need a more robust, syntax independent characterization of the precision relation.

$$[\preccurlyeq]\ \frac{\Gamma \vdash e : \tau \qquad \tau \preccurlyeq \tau'}{\Gamma \vdash e : \tau'}$$

In his Ph.D. dissertation, Lanvin [40] showed that this characterization can be given just in terms of static types (i.e., the types without any occurrence of $?$ in them). Take as static types either the set-theoretic types of Definition 1 or their polymorphic extension given by grammar (1), together with their respective subtyping relations. To obtain gradual types add to the grammars of these types the production $t ::= ?$ (still, we use τ to range over gradual types and reserve t for static types). Given a gradual type τ, the set of all static types it materializes to forms a complete lattice, with a maximum and a minimum static type that we denote by $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$, respectively. So we have that for all τ, if $\tau \preccurlyeq t$, then $\tau^{\Downarrow} \leq t \leq \tau^{\Uparrow}$, where $\leq$ is the subtyping relation for the static types. It is very easy to derive the materialization extrema $\tau^{\Uparrow}$ and $\tau^{\Downarrow}$ from τ: you get $\tau^{\Uparrow}$ by replacing in τ every covariant occurrence of $?$ by $\mathbb{1}$ and every contravariant occurrence of $?$ by $\mathbb{0}$; $\tau^{\Downarrow}$ is obtained in the same way, by replacing in τ every covariant occurrence of $?$ by $\mathbb{0}$ and every contravariant one by $\mathbb{1}$. The definition of the minimal and maximal static materializations together with the subtyping relation on static types is all is

needed to define the precision relation *and* the subtyping relation on the newly defined gradual types. Lanvin [40] shows that it is possible to define the precision relation $\preccurlyeq$ and the subtyping relation $\dot{\leq}$ on gradual types as follows:

$$\tau_1 \preccurlyeq \tau_2 \quad \stackrel{\text{def}}{\iff} \quad {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \text{ and } {\tau_2}^{\Uparrow} \leq {\tau_1}^{\Uparrow} \tag{13}$$

$$\tau_1 \dot{\leq} \tau_2 \quad \stackrel{\text{def}}{\iff} \quad {\tau_1}^{\Downarrow} \leq {\tau_2}^{\Downarrow} \text{ and } {\tau_1}^{\Uparrow} \leq {\tau_2}^{\Uparrow} \tag{14}$$

where $\leq$ denotes the subtyping relation given on static types—e.g., the two subtyping relations induced by interpretations of types given in Sections 3.1 and 3.2—.[27]

Equation (13) conveys a strong message: any gradual type can be seen as an *interval* of *possible types*, where $?$ denotes the interval of all types, and a type τ denotes the interval ranging from $\tau^{\Downarrow}$ to $\tau^{\Uparrow}$ (or, more precisely, the sub-lattice of the types included between the two). Semantic materialization then allows us to reduce this interval, by going to any type τ' such that $\tau^{\Downarrow} \leq {\tau'}^{\Downarrow}$ and ${\tau'}^{\Uparrow} \leq \tau^{\Uparrow}$, possibly until reaching a static type (that is, a type τ such that $\tau^{\Downarrow} = \tau^{\Uparrow}$).

Equation (14) extends this interval interpretation to the subtyping relation of gradual types, stating that a type τ_1 is a subtype of τ_2 if the interval denoted by τ_1 only contains subtypes of elements of the interval denoted by τ_2.

Notice that these two definitions also provide an effective way to decide precision and subtyping for gradual types: generate the gradual extrema and check on them the subtyping relations for static types according to (13) or (14).

Lanvin justifies these definitions by giving a semantic interpretation of all gradual types (i.e., not just of the static ones) and proving all the needed properties. In particular, he proves a series of properties that show the robustness of the relations defined in (13) and (14). First, for all *gradual* types τ and τ' such that $\tau \preccurlyeq \tau'$ we have $\tau^{\Downarrow} \dot{\leq} \tau' \dot{\leq} \tau^{\Uparrow}$, that is, all the materializations of a gradual type form a complete sub-lattice (not just the static materializations). More surprisingly, for every gradual type τ we have $\tau \dot{\simeq} \tau^{\Downarrow} \vee (? \wedge \tau^{\Uparrow})$, where $\dot{\simeq}$ is the symmetric closure of the gradual subtyping relation $\dot{\leq}$. According to this last property, every gradual type τ is equivalent to the $?$ type as long as we bound it with the two extrema $\tau^{\Downarrow}$ and $\tau^{\Uparrow}$, thus strengthening the intuition of gradual types as intervals of static types. Therefore, every gradual type can be represented by a pair of static types, and to add gradual typing to a system, it suffices to augment the types with a single constant $?$ that only needs to appear at top level, that is, under neither an arrow nor a product. This characterization can then be used to define the gradual counterparts $\widetilde{\mathsf{dom}}(.)$, $\tilde{\circ}$, and $\widetilde{\boldsymbol{\pi}_i}(.)$ of the type operators $\mathsf{dom}(.)$, $\circ$, and $\boldsymbol{\pi}_i(.)$ we defined at the end of Section 4.1.2, thus providing a further proof of the robustness of the definitions in (13) and (14). So we have:

$$\begin{aligned}
\widetilde{\mathsf{dom}}(\tau) &\stackrel{\text{def}}{=} \mathsf{dom}(\tau^{\Uparrow}) \vee (? \wedge \mathsf{dom}(\tau^{\Downarrow})) \\
\tau \,\tilde{\circ}\, \tau' &\stackrel{\text{def}}{=} (\tau^{\Downarrow} \circ {\tau'}^{\Uparrow}) \vee (? \wedge (\tau^{\Uparrow} \circ {\tau'}^{\Downarrow})) \\
\widetilde{\boldsymbol{\pi}_i}(\tau) &\stackrel{\text{def}}{=} (\boldsymbol{\pi}_i(\tau^{\Downarrow})) \vee (? \wedge (\boldsymbol{\pi}_i(\tau^{\Uparrow})))
\end{aligned}$$

for which we can prove that $\widetilde{\mathsf{dom}}(\tau) = \max\{\tau' \mid \tau \dot{\leq} \tau' \to \mathbb{1}\}$, $\tau_1 \,\tilde{\circ}\, \tau_2 = \min\{\tau \mid \tau_1 \dot{\leq} \tau_2 \to \tau\}$, $\widetilde{\boldsymbol{\pi}_1}(\tau) = \min\{\tau' \mid \tau \dot{\leq} \tau' \times \mathbb{1}\}$, and $\widetilde{\boldsymbol{\pi}_2}(\tau) = \min\{\tau' \mid \tau \dot{\leq} \mathbb{1} \times \tau'\}$.

[27] Strictly speaking, it is necessary slightly to modify these interpretations so that all the types of the form $\mathbb{0} \to t$ are not all equivalent: see Lanvin [40, Section 6.1.2].

5.3 Denotational Semantics

We have seen in Section 3 that the essence of semantic subtyping is to interpret types as sets of values. However, for the circularity problem described in Section 4.2.1 this cannot be done directly on the values of some language, but must pass via an interpretation in a domain $\mathscr{D}$ whose elements represent these values. Furthermore, for cardinality problems, functional values cannot be represented directly as elements of the domain, and one has to interpret types as sets containing only functions with finite graphs. Even if at the end one obtains the same subtyping relation as if we had considered infinite functions (cf., Section 3.1) this solution has been making readers uneasy. The fact of using finite graph functions to define a relation for general function spaces looked more as a technical trick than as a theoretical breakthrough. Pierre-Louis Curien suggested that the construction was a *pied de nez* to (it cocked a snook at) denotational semantics, insofar as it used a semantic construction to define a language for which a denotational semantics was not known to exist. The common belief was that the solution worked because considering *all* finite functions in the interpretation of a function space was equivalent to give the finite approximations of the non-finite functions in that space, in the same way as, say, Scott domains are built by giving finite approximations of the functions therein.

Very recently, Lanvin's Ph.D. dissertation [40, Part 2] has formalized this intuition and defined a denotational semantics for a language with semantic subtyping (actually, the language of Sections 4.2.2–4.2.4), in which functions are interpreted as the infinite set of their finite approximations. This yields a model with a simple *inductive* definition, which does not need isomorphisms or the solution of domain equation. The idea is to interpret not only types but also terms in the domain $\mathscr{D}$ of Definition 2. Unfortunately, the domain $\mathscr{D}$ cannot be used as is,[28] but it must be slightly modified to account for the fact that functions map finite approximations (rather single denotations) into denotations. In practice, one has to modify the domain as follows

$$d ::= c \mid (d,d) \mid \{(S,\partial),\ldots,(S,\partial)\} \qquad S ::= \{d,\ldots,d\} \qquad \partial ::= d \mid \Omega$$

where, thus, the domain now includes finite maps from *finite and non-empty* sets of elements (ranged over by S) into other elements or Ω. The interpretation of types must also be slightly modified to make the interpretation of arrows satisfy the following equation:

$$[\![t_1 \to t_2]\!] = \{R \in \mathscr{P}_{\text{fin}}(\mathscr{P}_{\text{fin}}(\mathscr{D}) \times \mathscr{D}_\Omega) \mid \forall (S,\partial) \in R\,.\, S \cap [\![t_1]\!] \neq \varnothing \implies \partial \in [\![t_2]\!]\}\}$$

modification that yields the same subtyping relation as the one produced by the interpretation of Definition 3. In this domain it is then not very difficult to interpret every term of the language into the (possibly infinite) *set* of its finite approximations. For instance, a constant c will be interpreted as the singleton $\{c\}$. The only delicate part is the interpretation of λ-abstractions, that is defined as follows:

$$[\![\lambda x{:}t.e]\!]_\rho = \{R \in \mathscr{P}_{\text{fin}}(\mathscr{P}_{\text{fin}}(\mathscr{D}) \times \mathscr{D}_\Omega) \mid \forall (S,\partial) \in R, \text{ either } S \subseteq [\![t]\!] \text{ and } \partial \in [\![e]\!]_{\rho,x\mapsto S} \text{ or } S \subseteq [\![\neg t]\!] \text{ and } \partial = \Omega\}$$

[28] Actually, it can but it yields a weak property of computational soundness: cf. [40, Chapter 9].

where ρ is a semantic environment that maps variables into approximations, that is, into sets in $\mathscr{P}_{\text{fin}}(\mathscr{D})$. The definition above states that a λ-abstraction is interpreted as the set of all finite approximations that map any approximation in the domain of the function to the interpretation of the body where the parameter is associated to that approximation, and any approximation that is outside this domain to the failure Ω. Notice that, as we anticipated in Section 4.2.2, the denotation of a function depends on its type annotation.

For this interpretation it is possible to prove three fundamental properties:

1. Type soundness: if $\Gamma \vdash e : t$, then $[\![e]\!]_\rho \subseteq [\![t]\!]$, for every $\rho \in [\![\Gamma]\!]$.[29]
2. Computational soundness: if $\Gamma \vdash e : t$, and $e \rightsquigarrow e'$, then $[\![e]\!]_\rho = [\![e']\!]_\rho$, for all $\rho \in [\![\Gamma]\!]$.
3. Computational adequacy: $[\![e]\!]_\rho = \varnothing$, for every well-typed closed *diverging* term e.

This interpretation works only for a language without type-cases and overloaded functions (notice the syntax of annotations which are just on the parameter of λ-abstractions and not on the full term). The reader can refer to Lanvin [40, Part 2] for all details and a denotational semantics of the whole Core CDuce language.

6 Conclusion

In this essay I tried to survey the multiple advantages and usages of set-theoretic types in programming. Set-theoretic types are sometimes the only way to type some particular functions, sometimes as simple as the `flatten` function of the introduction. This is so because set-theoretic types provide a suitable language to describe many non-conventional, but not uncommon, programming patterns. This is demonstrated by the fact that the need of set-theoretic types naturally arises when trying to fit type-systems on dynamic languages: union and negations become necessary to capture the nature of branching and of pattern matching, intersections are often the only way to describe the polymorphic use of some functions whose definition lacks the uniformity required by parametric polymorphism. The development of languages such as Flow, TypeScript, and Typed Racket is the latest witness of this fact. I also showed that even when set-theoretic types are not exposed to the programmer, they are often present at meta level since they provide the basic tools to precisely type some program constructions such as type-cases and pattern matching. Finally, set-theoretic types provide a powerful theoretic toolbox to explore, understand, and formalize existing type disciplines: I demonstrated this with gradual types which, thanks to set-theoretic types, can be understood as intervals of static types, an analogy that we can use to rethink both their theory (see Lanvin's dissertation [40]) and their practice (e.g., the implementation of gradual virtual machines as in [14]).

This survey is necessarily incomplete. For instance I barely spoke of XML types and XML programming even though they were the first motivation for developing the theory of semantic subtyping and to design and implement programming languages such as XDuce and CDuce. Also, I completely swept under the carpet how to handle features that are common in modern programming languages such as the use of abstract types—whose integration with structural subtyping and polymorphism may result delicate—and the presence of side-effects. The latter is particular sensitive for the

[29] Where $[\![\Gamma]\!] = \{\rho \mid \forall x \in \mathsf{dom}(\Gamma)\,.\,\rho(x) \subseteq [\![\Gamma(x)]\!]\}$.

language presented in Section 4.4, insofar as the use of MSC-forms is sound only for pure expressions. Nevertheless, I hope I gave a good idea of the potentiality of having set-theoretic types in a programming language and how the addition of these types can be done.

It is not all a bed of roses though. From a formal point of view we did not succeed, yet, to define a unique formalism that mixes implicitly and explicitly typed functions, reconstruction of intersection types, and an advanced use of occurrence typing. But we are not far from it. From a practical viewpoint even more work is needed. We have seen that parametric polymorphism with set-theoretic types implies constraint generation and constraint resolution (i.e., structured and (sub-)typing constraints in the implicitly-typed language and only the latter in the explicitly-typed one). This has several drawbacks. Foremost, because of the presence of unions and of subtyping, constraint solving is a potential source of computational explosion that we do not master well yet. Furthermore, constraint solving makes the generation of informative error messages very difficult for the case when it fails, but even pretty printing the deduced types in a form easily understandable by the programmer may sometimes happen to be challenging. So the positive message with which I want to conclude this presentation is that, all in all, the research of set-theoretic types still is a very nice playground that reserves us several interesting and challenging problems yet to be solved.

Acknowledgements: The work presented here is the result of many collaborations with many coauthors that I listed in the first page of this article. The "we" I used all the presentation long and abandoned from the conclusion must be intended as inclusive of all of them. Not only the results exposed here were first presented in articles I co-authored or dissertations I supervised but, in some cases, I also adapted or reused verbatim the text of these works. Thus several parts of this presentation are borrowed from Castagna et al. [16]. Section 2 and the beginning of Section 3 faithfully reproduce and extend Section 1.1 of Tommaso Petrucciani PhD thesis [49] whose reading I warmly recommend. Sections 3.1 and 3.2 follow a presentation that can be found in several papers I co-authored. Most of Section 4.2 comes from some unpublished notes that Victor Lanvin and I wrote on the denotational semantics of CDuce, while Section 5.2 reuses parts of Lanvin [40, Chapter 6]. James Clark, Guillaume Duboc, Victor Lanvin, Mickaël Laurent, Matt Lutze, Kim Nguyen, and José Valim provided useful feedback on an early version of this manuscript.

References

1. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types. *Inf. Comput.*, 119(2):202–230, June 1995. ISSN 0890-5401. https://doi.org/10.1006/inco.1995.1086.
2. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press. https://doi.org/10.1145/944705.944711.
3. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985. https://doi.org/10.1145/6041.6042.

4. Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994. https://doi.org/10.1006/inco.1994.1013.
5. G. Castagna, M. Laurent, V. Lanvin, and K. Nguyen. Revisiting occurrence typing. *Science of Computer Programming*, 217:102781, mar 2022. ISSN 0167-6423. https://doi.org/10.1016/j.scico.2022.102781. Preprint available at **https://arxiv.org/abs/1907.05590**.
6. Giuseppe Castagna. Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science*, 16(1):15:1–15:58, 2020. https://doi.org/10.23638/LMCS-16(1:15)2020. New and extended version available at the author's web page: **https://www.irif.fr/~gc/papers/covcon-again.pdf**.
7. Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming,* pages 198-208, ACM Press (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming,* Lecture Notes in Computer Science n. 3580, pages 30-34, Springer (summary), Lisboa, Portugal, July 2005. https://doi.org/10.1145/1069774.1069793. Joint ICALP-PPDP keynote talk.
8. Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, 1, Article 41(ICFP'17), September 2017. https://doi.org/10.1145/3110285.
9. Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*, pages 94–106, 2011. https://doi.org/10.1145/2034773.2034788.
10. Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'14, pages 5–17, January 2014. https://doi.org/10.1145/2676726.2676991.
11. Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* Castagna et al. [12], pages 289–302. https://doi.org/10.1145/2676726.2676991.
12. Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'15, pages 289–302, January 2015. https://doi.org/10.1145/2676726.2676991.
13. Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming*, pages 378–391, September 2016. https://doi.org/10.1145/2951913.2951928.
14. Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, IFL '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450375627. https://doi.org/10.1145/3412932.3412940.
15. Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: a new perspective. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. https://doi.org/10.1145/3290329.
16. Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. On type-cases, union elimination, and occurrence-typing. *Proc. ACM Program. Lang.*, 6(POPL), 2022. https://doi.org/10.1145/3498674. To appear.

17. CDuce. The CDuce Compiler. **https://www.cduce.org**.
18. Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):48:1–48:30, October 2017. ISSN 2475-1421. https://doi.org/10.1145/3133872.
19. Mariangiola Dezani-Ciancaglini, Alain Frisch, Elio Giovannetti, and Yoko Motohama. The relevance of semantic subtyping. *Electronic Notes in Theoretical Computer Science*, 70(1): 88 – 105, 2003. ISSN 1571-0661. https://doi.org/10.1016/S1571-0661(04)80492-4. ITRS '02, Intersection Types and Related Systems.
20. Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72. ACM, 2017. ISBN 978-1-4503-4660-3. https://doi.org/10.1145/3009837.3009882.
21. Facebook. Flow. **https://flow.org/**.
22. Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, Université Paris 7 – Denis Diderot, 12 2004. **http://www.cduce.org/papers/frisch_phd.pdf**.
23. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002. https://doi.org/10.1109/LICS.2002.1029823.
24. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):19:1–19:64, September 2008. ISSN 0004-5411. https://doi.org/10.1145/1391289.1391293.
25. Ronald Garcia. Calculating threesomes, with blame. In *ICFP '13: Proceedings of the International Conference on Functional Programming*, 2013. https://doi.org/10.1145/2500365.2500603.
26. Nils Gesbert, Pierre Genevès, and Nabil Layaïda. Parametric polymorphism and semantic subtyping: the logical connection. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 107–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. https://doi.org/10.1145/2034773.2034789.
27. Nils Gesbert, Pierre Genevès, and Nabil Layaïda. A logical approach to deciding semantic subtyping. *ACM Transactions on Programming Languages and Systems*, 38(1):3, 2015. https://doi.org/10.1145/2812805.
28. Google. Dart programming language specification. **https://dart.dev/guides/language/spec**.
29. Michael Greenberg. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, 2019. ISBN 978-3-95977-113-9. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6.
30. Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, 1978. https://doi.org/10.1109/SFCS.1978.3.
31. Robert Harper. *Programming Languages: Theory and Practice*. Carnegie Mellon University, 2006. Available on the web: **http://fpl.cs.depaul.edu/jriely/547/extras/online.pdf**.
32. J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators An Introduction*. Cambridge University Press, 2008.
33. Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

34. Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *POPL'01, 25th ACM Symposium on Principles of Programming Languages*, 2001. https://doi.org/10.1145/360204.360209.
35. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003. https://doi.org/10.1145/767193.767195.
36. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP'00*, volume 35(9) of *SIGPLAN Notices*, 2000. https://doi.org/10.1145/351240.351242.
37. Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. *ACM Transactions on Programming Languages and Systems*, 32(1):1–56, 2009. https://doi.org/10.1145/1596527.1596529.
38. JetBrains. Kotlin documentation. Available at **http://kotlinlang.org/docs/reference**, 2018.
39. Gavin King. The ceylon language specification, version 1.3. Available at **https://ceylon-lang.org/documentation/1.3/spec**, 2017.
40. Victor Lanvin. *A Semantic Foundation for Gradual Set-Theoretic Types*. PhD thesis, Université de Paris, November 2021. URL **https://tel.archives-ouvertes.fr/tel-?????????**
41. David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1):95–130, 1986. ISSN 0019-9958. https://doi.org/10.1016/S0019-9958(86)80019-5.
42. Per Martin-Löf. *Analytic and Synthetic Judgements in Type Theory*, pages 87–99. Springer Netherlands, Dordrecht, 1994. ISBN 978-94-011-0834-8. https://doi.org/10.1007/978-94-011-0834-8_5.
43. Microsoft. TypeScript. **https://www.typescriptlang.org/**.
44. Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):112:1–112:29, October 2018. ISSN 2475-1421. https://doi.org/10.1145/3276482.
45. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. https://doi.org/10.1017/CBO9780511530104.
46. Chris Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999.
47. David J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Verification, Model Checking, and Abstract Interpretation*, pages 335–354. Springer, 2013.
48. David J. Pearce and Lindsay Groves. Whiley: a platform for research in software verification. In *Software Language Engineering*, pages 238–248, Cham, 2013. Springer International Publishing. ISBN 978-3-319-02654-1.
49. Tommaso Petrucciani. *Polymorphic Set-Theoretic Types for Functional Languages*. PhD thesis, Joint Ph.D. Thesis, Università di Genova and Université Paris Diderot, March 2019. URL **https://tel.archives-ouvertes.fr/tel-02119930**.
50. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
51. Benjamin Crawford Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, USA, 1992. URL **https://www.cis.upenn.edu/~bcpierce/papers/thesis.pdf**.
52. François Pottier and Didier Rémy. *The essence of ML type inference*, chapter 10, pages 389–489. MIT Press, 2005.
53. John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.
54. John C. Reynolds. *Programming Methodology*, chapter What do types mean? – From intrinsic to extrinsic semantics. Monographs in Computer Science. Springer, 2003.

55. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, page 288–298, New York, NY, USA, 1992. Association for Computing Machinery. https://doi.org/10.1145/141471.141563.
56. Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
57. Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. Technical Report CU-CS-1039-08, University of Colorado at Boulder, January 2008.
58. Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '16, pages 456–468. ACM, 2016. ISBN 978-1-4503-3549-2. https://doi.org/10.1145/2914770.2837630. URL **http://doi.acm.org/10.1145/2914770.2837630**.
59. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. https://doi.org/10.1145/1328438.1328486. URL **http://doi.acm.org/10.1145/1328438.1328486**.
60. Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. https://doi.org/10.1145/1863543.1863561. URL **http://doi.acm.org/10.1145/1863543.1863561**.
61. Types. What exactly should we call syntax-directed inference rules? Discussion on the Types mailing list, jun 2019. **http://lists.seas.upenn.edu/pipermail/types-list/2019/002138.html**.
62. Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP '09, pages 1–16. Springer, 2009. https://doi.org/10.1007/978-3-642-00590-9_1.
63. Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticæ*, 10:115–122, 1987. https://doi.org/10.3233/FI-1987-10202.
64. Andrew K. Wright. Simple imperative polymorphism. *LISP Symb. Comput.*, 8(4):343–355, 1995. https://doi.org/10.1007/BF01018828.

A Core Calculus of CDuce [24]

Syntax

Types $t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$
Expressions $e ::= c \mid x \mid \lambda^{\wedge_{i\in I} t_i \to t_i} x.e \mid ee \mid \pi_i e \mid (e,e) \mid (x{=}e{\in}t)\,?\,e\,{:}\,e \mid \mathsf{choice}(e,e)$
Values $v ::= c \mid x \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid (v,v)$

Reduction semantics

$$\begin{array}{rl}
(\lambda^{\wedge_{i\in I} s_i\to t_i} x.e)v \rightsquigarrow e\{v/x\} & \\
\pi_i(v_1,v_2) \rightsquigarrow v_i & i=1,2\\
\mathsf{choice}(e_1,e_2) \rightsquigarrow e_i & i=1,2\\
(x{=}v{\in}t)\,?\,e_1\,{:}\,e_2 \rightsquigarrow e_1\{v/x\} & \text{if } v\in t\\
(x{=}v{\in}t)\,?\,e_1\,{:}\,e_2 \rightsquigarrow e_2\{v/x\} & \text{if } v\notin t
\end{array}$$

where $v \in t \overset{\text{def}}{\iff} \exists s{\in}\mathsf{typeof}(v).s \le t$ with

$$\begin{array}{rcl}
\mathsf{typeof}(c) & \overset{\text{def}}{=} & \{b_c\}\\
\mathsf{typeof}(\lambda^{\wedge_{i\in I} s_i\to t_i} x.e) & \overset{\text{def}}{=} & \{t \mid t \simeq (\bigwedge_{i\in I} s_i \to t_i) \wedge (\bigwedge_{j\in J} \neg(s'_j \to t'_j)), t \not\leq \mathbb{0}\}\\
\mathsf{typeof}((v_1,v_2)) & \overset{\text{def}}{=} & \mathsf{typeof}(v_1) \times \mathsf{typeof}(v_2)
\end{array}$$

plus the standard context rule implementing a leftmost outermost strategy, namely, $E[e] \rightsquigarrow E[e']$ if $e \rightsquigarrow e'$, where

$$E ::= [\,] \mid Ee \mid vE \mid (E,e) \mid (v,E) \mid \pi_i E \mid (x{=}E{\in}t)\,?\,e\,{:}\,e$$

Type-system

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b_c}} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

$$[{\to}\text{I}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i \quad t = \wedge_{i\in I}(s_i \to t_i) \quad t' = \wedge_{j\in J}\neg(s'_j \to t'_j) \quad t \wedge t' \not\simeq \mathbb{0}}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i\to t_i} x.e : t \wedge t'}$$

$$[{\to}\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad t_1 \le \mathbb{0} \to \mathbb{1} \quad t_2 \le \mathsf{dom}(t_1)}{\Gamma \vdash e_1 e_2 : t_1 \circ t_2}$$

$$[{\times}\text{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1,e_2) : t_1 \times t_2} \qquad [{\times}\text{E}_i]\ \frac{\Gamma \vdash e : t \quad t \le \mathbb{1}\times\mathbb{1}}{\Gamma \vdash \pi_i e : \boldsymbol{\pi_i}(t)}\ i=1,2$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash e : t' \quad \Gamma, x : t\wedge t' \vdash e_1 : s \quad \Gamma, x : \neg t \wedge t' \vdash e_2 : s}{\Gamma \vdash (x{=}e{\in}t)\,?\,e_1\,{:}\,e_2 : s} \qquad [\textsc{Efq}]\ \frac{}{\Gamma, x : \mathbb{0} \vdash e : t}$$

$$[\textsc{Choice}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \mathsf{choice}(e_1,e_2) : t_1 \vee t_2}$$