On Type-Cases, Union Elimination, and Occurrence Typing

GIUSEPPE CASTAGNA, CNRS, Université de Paris, France
MICKAËL LAURENT, Université de Paris, France
KIM NGUYỄN, Université Paris-Saclay, France
MATTHEW LUTZE, Université de Paris, France

We extend classic union and intersection type systems with a type-case construction and show that the combination of the union elimination rule of the former and the typing rules for type-cases of our extension encompasses *occurrence typing*. To apply this system in practice, we define a canonical form for the expressions of our extension, called MSC-form. We show that an expression of the extension is typable if and only if its MSC-form is, and reduce the problem of typing the latter to the one of reconstructing annotations for that term. We provide a sound algorithm that performs this reconstruction and a proof-of-concept implementation.

CCS Concepts: • **Theory of computation** → *Type structures*; *Program analysis*; • **Software and its engineering** → *Functional languages*.

Additional Key Words and Phrases: subtyping, union types, intersection types, type-case, dynamic languages, type systems.

ACM Reference Format:
Giuseppe Castagna, Mickaël Laurent, Kim Nguyễn, and Matthew Lutze. 2022. On Type-Cases, Union Elimination, and Occurrence Typing. *Proc. ACM Program. Lang.* 6, POPL, Article 13 (January 2022), 31 pages. https://doi.org/10.1145/3498674

1 INTRODUCTION

TypeScript [Microsoft] and Flow [Facebook] are extensions of JavaScript that allow the programmer to specify in the code type annotations used to statically type-check the program. For instance, the following function definition is valid in both languages

```
function foo(x : number | string) {
   return (typeof(x) === "number")? x+1 : x.trim();
}
```
(1)

Apart from the type annotation (in red) of the function parameter, the above is standard JavaScript code defining a function that checks whether its argument is an integer; if it is so, then it returns the argument's successor (`x+1`), otherwise it calls the method `trim()` of the argument. The annotation specifies that the parameter is either a number or a string (the vertical bar denotes a union type). If this annotation is respected and the function is applied to either an integer or a string, then the application cannot fail because of a type error (`trim()` is a standard string method) and both TypeScript and Flow rightly accept this function and deduce that it will return either a number or a string, that is, a result of type `number|string`. This is possible because both type-checkers

Authors' addresses: Giuseppe Castagna, Mickaël Laurent, Matthew Lutze, Institut de Recherche en Informatique Fondamentale (IRIF), Université de Paris, CNRS, 8 place Aurélie Nemours, 75013 Paris, France; Kim Nguyễn, Laboratoire Méthodes Formelles (LMF), Université Paris-Saclay, CNRS, ENS Paris-Saclay, 91190, Gif-sur-Yvette, France.

2475-1421/2022/1-ART13
https://doi.org/10.1145/3498674

implement a specific type discipline called *occurrence typing* or *flow typing*:[1] as a matter of fact, standard type disciplines would reject this function. The reason for that is that standard type disciplines would try to type every part of the body of the function under the assumption that `x` has type `number | string` and they would fail, since the successor is not defined for strings and the method `trim()` is not defined for numbers. This is so because standard disciplines do not take into account the type test performed on `x`. Occurrence typing is the typing technique that uses the information provided by the type test to specialize—precisely, to *refine*—the type of the occurrences of `x` in the branches of the conditional: since the program tested that `x` is of type `number`, then we can safely assume that `x` is of type `number` in the "then" branch, and that it is *not* of type `number` (and thus deduce from the type annotation that it must be of type `string`) in the "else" branch.

Occurrence typing was first defined and formally studied by Tobin-Hochstadt and Felleisen [2008] to statically type-check untyped Scheme programs, and later extended by Tobin-Hochstadt and Felleisen [2010] yielding the development of Typed Racket. To that end the authors define a system to deduce propositions, that express relations between variables and types, and these are attached to functional types (*à la* effect system) to describe facts about the result of applying the corresponding functions. In this work we argue that to capture occurrence typing—and, as we detail later on, much, much more—it is not necessary to resort to effect-like systems, perform flow analysis, or invent new expressive types. It just suffices to add to the language at issue a type-case expression and combine (a tailored definition of) its typing rules with the classic union elimination rule as it was first introduced by MacQueen et al. [1986]. More precisely, let t be a type and $(e{\in}t)\,?\,e_1 : e_2$ 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. Then we claim that all the situations in which occurrence typing is used are covered by these three typing rules.

$$\frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x{:}t_1 \vdash e : t \quad \Gamma, x{:}t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\,?\,e_1 : e_2 : t_1} \qquad \frac{\Gamma \vdash e : \neg t \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\,?\,e_1 : e_2 : t_2}$$

The first is the classic union elimination rule by MacQueen et al. [1986] where $e\{e'/x\}$ is the expression obtained from e by substituting e' for x and $t_1 \vee t_2$ is a *union type* (for union and intersection we respectively use `|` and `&` in code snippets and $\vee$ and $\wedge$ in formal types). If we interpret a type set-theoretically as the set of all values that have that type, then $t_1 \vee t_2$ is the type that contains all values of type t_1 and all values of type t_2. In a sound system an expression e is given a type t only if it can only produce a result in t; this implies that an expression of type $t_1 \vee t_2$ can produce a result either of type t_1 or of type t_2 . The union elimination rule 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 the e' produces a result in t_2, then we can safely give this expression type t.

The two other rules are new and provide a natural and nifty way to type type-case expressions. The first 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 second 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, the type-case in (1) tests whether `x` is of type `number`, but since `x` is of type `number|string`, then it is neither of type `number` nor of type `¬number`. Here, the combination of set-theoretic types and the union rule plays its

[1] TypeScript calls it "type guard recognition" while Flow uses the terminology "type refinements".

magic. Union and negation types, give intersection types for free: just define $t_1 \wedge t_2$ as $\neg(\neg t_1 \vee \neg t_2)$. Thus, even though the tested expression e 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 (1) we check the type-case under the hypothesis that `x` has type `number` (i.e., `(number∨string)∧number`) and deduce the type `number`, and under the hypothesis that `x` has type `string` (i.e., `(number∨string)∧¬number`) and deduce the type `string`, which by subsumption gives for the whole expression the expected type `number∨string`.

We see that our treatment of occurrence typing heavily depends on the properties of *set-theoretic types*: unions, intersections, and negations of types. This does not come as a surprise since from its inception occurrence typing was intimately tied to type systems with set-theoretic types. Union was the first type connective to appear, since it was already used in [Tobin-Hochstadt and Felleisen 2008] to characterize the different control flows of a type test, as our `foo` example shows: one flow for integer arguments and another for strings. Intersection types appear (in limited forms) combined with occurrence typing both in TypeScript and in Flow and serve to give, among other things, more precise types to functions such as `foo`. For instance, since `x + 1` evaluates to an integer and `x.trim()` to a string, then our function `foo` has type `(number|string)→(number|string)`. But it is clear that a more precise type would be one that states that `foo` returns a number when it is applied to a number and returns a string when it is applied to a string, so that the type deduced for, say, `foo(42)` would be `number` rather than the less precise `number|string`. This is exactly what the *intersection type*

```
(number→number) & (string→string)                    (2)
```

states (intuitively, an expression has an intersection of types, noted `&`, if and only if it has all the types of the intersection) and corresponds in Flow to declaring `foo` as follows:

```
var foo : (number => number) & (string => string) =  x => {
   return (typeof(x) === "number")? x+1 : x.trim();   (3)
}
```

For what concerns negation types, they are pervasive in the occurrence typing approach, even though they are used only at meta-theoretic level, in particular to determine the type environment when the type-case fails. We already saw negation types at work when we informally typed the "else" branch in `foo`, for which we assumed that x did *not* have type `number`—i.e., it had the (negation) type `¬number`—and deduced from it that x then had type `string`—i.e., `(number|string)&¬number` which is equivalent to the set-theoretic difference `(number|string)\number` and, thus, to `string`.

Since set-theoretic types play such a pivotal role in our treatment the system we study in this work is a conservative extension of the standard type assignment system for union and intersection types, which was defined by Barbanera et al. [1995, Definition 3.5]. To cope with occurrence typing we extend, in a nutshell, the system of [Barbanera et al. 1995] with three simple ingredients:

(1) we add to its expressions the type-case expression $(e \in t)\,?\,e_1 : e_2$;
(2) we add to its types the negation connective and the empty type;
(3) we add to its deduction rules the typing rules for type-cases we showed before.

The resulting system, presented in Section 2, is a type-assignment system for an untyped λ-calculus with constants, pairs, and type-cases.

We said earlier that the combination of the union rule and set-theoretic types gives us much more than what current formalizations of occurrence typing can capture. The approaches cited above essentially focus on refining the type of variables that occur in an expression whose type is being tested. They do it when the variable occurs at top-level in the test (i.e., the variable is the

expression being tested) or under some specific positions such as in nested pairs or at the end of a path of selectors. The union elimination rule of MacQueen et al. [1986] does not have such limitations: it can refine the type of *any* expression e' occurring in *any* position of the current expression e, by splitting the type of e' into a union of types that are tested separately for e. The separation of these tests combines with our new rules for type-cases to yield a system in which different branches of the same type-case are typed under different typing hypotheses, which is the essence of occurrence typing. In this work we aim at exploiting this power of the union rule and refine the type of any expression that occurs in a tested expression (or elsewhere). Of particular interest will be the expressions occurring in applications since the refinement of their types is pivotal in deducing and exploiting intersection types for functions. For example, let x_1 be a variable of type $(\mathsf{Int}\to\mathsf{Int}) \wedge (\mathsf{String}\to\mathsf{String})$ (e.g., x_1 binds the function `foo` defined in (3)) and x_2 be of type $\mathsf{Int} \vee \mathsf{String}$. If `@` denotes string concatenation, then we can use the three typing rules above to deduce that the following expression has type $\mathsf{Int} \vee \mathsf{String}$.

$$(x_1 x_2 \in \mathsf{Int})\,?\,(x_2 + 1) : ((x_1 x_2)\,@\,x_2) \tag{4}$$

This is done by splitting the type of x_2: if x_2 is of type Int, then $x_1 x_2$ is also of type Int, thus the first branch is selected and the addition in it is well typed with type Int; if x_2 is of type String, then $x_1 x_2$ is also of type String, thus the second branch is selected and the concatenation in it is well typed with type String. This is an example of typing that is out of reach for all the previously cited approaches. This is possible because our approach does not perform occurrence typing using only the information given by type-cases: it also uses all type information provided by applications, even more when functions are overloaded (i.e., typed by an intersection of arrows, as x_1 in (4)).

One of the most important consequences of such a thorough analysis is that we can use its results to infer intersection types for functions, even in the absence of precise annotations such as the one given in the definition of `foo` in (3): we split the type of the function parameter (initially supposed to be the top type `Any`) and deduce a distinct arrow for each split of the input type, discarding from the domain the split types for which the inference fails. To put it simply, we can infer the type (2) for the unannotated pure JavaScript code of `foo` (i.e., no type annotation at all), while in TypeScript and Flow (and any other formalism we are aware of) this requires an explicit and full type annotation as the one given in (3). This creates a virtuous circle since, as the program in (4) shows, determining intersection types for functions is crucial to refine the type of expressions in applications, which allows to deduce more precise types for functions and so forth. Thanks to this virtuous circle our system can type a whole class of functions that other systems fail to type (even when given explicit full type annotations) and must then hard-code to retain sufficient expressiveness. And for well-typed programs our approach needs, in general, fewer annotations. For instance, we can type all the 14 paradigmatic examples of Tobin-Hochstadt and Felleisen [2010] without any annotation, whereas they need to specify annotations for 5 of them (see Section 6).

Having three typing rules that allow us to type all the examples of occurrence typing (and even more) is still a far cry from a practical system that can decide whether a program of our source language (the untyped λ-calculus with constants, pairs, and type-cases) is well-typed or not. The culprit is, of course, the powerful union rule: to use this rule to type some expression e one has to guess a subexpression e' of e to single out, the occurrences of e' to be tested, and how to split the type of this e' in a union of types to be tested separately. This is no simple feat: according to Mariangiola Dezani, arguably the best expert in union and intersection type systems, determining an inversion (a.k.a., generation) lemma for this union rule is the most important open problem in this field of research [Dezani-Ciancaglini 2020]. And an inversion lemma is an important aid to define a type-inference algorithm, since it tells us when and how to apply the rule.

```
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
```

Fig. 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
```

Fig. 2. Annotated MSC-form

In order to tackle this last problem of deciding whether an expression is well typed or not, we transform it into an equivalent problem in which the range of possible choices is much more restricted. To that end we introduce a canonical form for the expressions of our source language that we call *maximal-sharing canonical form* (MSC-form). 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. We call these forms *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 some trivial syntactic conversions) MSC-form and (ii) is well-typed if and only if its MSC-form is. For instance, consider the expression in (4) where x_1 and x_2 rather than being variables are generic atoms of type $t_1 = (\mathsf{Int}\to\mathsf{Int}) \wedge (\mathsf{String}\to\mathsf{String})$ and $t_2 = \mathsf{Int} \vee \mathsf{String}$, that is $(a_1a_2 \in \mathsf{Int})?(a_2 + 1) : ((a_1a_2)\,@\,a_2)$. Its MSC-form will look like the term in Figure 1. Notice that this term satisfies the maximal sharing property because the two occurrences of the application a_1a_2 in the source language expression are bound by the same variable x_3. The other crucial property that we prove is 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. These annotations essentially define how the type of the variables must be split and the annotated MSC-form type-checks if the rest of the expression type-checks for each of the splits specified in its annotations. Figure 2 gives the annotations for the previous MSC-form. The important annotations are those of the variables x_2 and x_3. The first states that to type the expression, the type $\mathsf{Int}\vee\mathsf{String}$ of a_2 must be split and the expression must be checked separately for $x_2 : \mathsf{Int}$ and $x_2 : \mathsf{String}$. 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 (see Section 4 for details).

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 atoms. This is done by an algorithm we present in Section 5 which 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 (i) the MSC-form type-checks with its current annotations and the algorithm stops with a success, or (ii) 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 (iii) 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 algorithms 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 Figures 1 and 2 is determined by the test $x_3 \in \mathsf{Int}$: the algorithm will propose to split the type t_3 of x_3 into $t_3 \wedge \mathsf{Int}$ and $t_3 \wedge \neg\mathsf{Int}$. Since t_3 is $\mathsf{Int} \vee \mathsf{String}$, the split proposed for x_3 is actually Int or String. This split triggers in the subsequent pass the split for the type of x_2 since x_3 is defined as $x_1 x_2$ and x_3 can be of type Int only if x_2 is of type Int and it can be of type String only if x_2 is of type String. We just got the expected annotation.

Contributions. This work provides four main technical contributions. (*i*) We show how to extend the system of Barbanera et al. [1995] with negation types and type-cases and prove its soundness (§2); (*ii*) we define MSC-forms and prove that the problem of typing a term of the previous system is equivalent to the one of typing its MSC-form (§3); (*iii*) we prove that the latter problem is equivalent to adding type annotations in a MSC-form so that it becomes a well-typed term and that checking well-typedness of an annotated MSC-form is decidable (§4); (*iv*) we define a sound reconstruction algorithm for the annotations of a MSC-form (§5) and provide an implementation (§6).

More generally, this work provides a novel formal lens for viewing the conceptual core of occurrence typing, whose essence it reveals. It reframes occurrence typing in the more standard and general setting of classic union and intersection type systems and, in doing so, it removes several current limitations of existing approaches. The removal of these limitations makes it possible for our system not only to type several examples that are out of reach of existing approaches but also to deduce precise intersection types for completely unannotated functions, something that, in our ken, no other system is currently capable of. By the definition of MSC-forms, it circumscribes the use of union elimination to a very specific setting, thus advancing in the quest for the characterization of inversion of union rules. More importantly, it shows that well-typing in such systems is equivalent to the problem of reconstructing annotations for MSC-forms, thus providing a formal yard-stick to compare different approaches. We exploited this new setting to define a sound reconstruction algorithm that we rendered in a proof-of-concept implementation. This implementation demonstrates the potential practical implications of our work, even though the gap with mature implementations such as those of Flow, Typed Racket, or TypeScript is still huge. Last but not least, we obtained a system that is arguably more robust to extensions such as the addition of side-effects and of polymorphic types, as we are eager to verify in the near future.

For space reasons several definitions and rules, the extensions with records and let-constructs, and all the proofs are moved to the appendix, available on line as supplemental material.

2 SOURCE LANGUAGE AND DECLARATIVE TYPE SYSTEM

2.1 Types

Types are exactly those of the semantic subtyping framework by Frisch et al. [2002, 2008].

Definition 2.1 (Types). *The set of types* **Types** *is formed by the terms t coinductively produced by the grammar:*

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

and that satisfy the following conditions

- *(regularity) every term has a finite number of different sub-terms;*
- *(contractivity) every infinite branch of a term contains an infinite number of occurrences of the arrow or product type constructors.*

We introduce the abbreviations $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2)$, and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$. b ranges over basic types (e.g., Int, Bool), $\mathbb{0}$ and $\mathbb{1}$ respectively denote the empty (that types no value) and top (that types all values) types. Coinduction accounts for recursive types and the condition on infinite branches bars out ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). Regularity is needed

for the decidability of the subtyping relation. We refer to b, $\times$, and $\to$ as *type constructors*, and to $\vee$, $\neg$, $\wedge$, and $\setminus$ 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 $((\neg s) \vee t) \to (u \wedge v)$.

The subtyping relation for these types, noted $\leq$, is the one defined by Frisch et al. [2008] to which the reader may refer for the formal definition (we recall it in Appendix A.1 for the reader's convenience). A detailed description of the algorithm to decide this relation can be found in [Castagna 2020]. For this presentation it suffices to consider that types are interpreted as sets of *values* (i.e., either constants, λ-abstractions, or pairs of values: see Section 2.2 right below) that have that type, and that subtyping is set containment (i.e., a type s is a subtype of a type t if and only if t contains all the values of type s). In particular, $s \to t$ contains all λ-abstractions that when applied to a value of type s, if their computation terminates, then they return a result of type t (e.g., $\mathbb{0} \to \mathbb{1}$ is the set of all functions and $\mathbb{1} \to \mathbb{0}$ is the set of functions that diverge on every argument). Type connectives (i.e., union, intersection, negation) are interpreted as the corresponding set-theoretic operators (e.g., $s \vee t$ is the union of the values of the two types). We use $\simeq$ to denote the symmetric closure of $\leq$: thus $s \simeq t$ (read, s is equivalent to t) means that s and t denote the same set of values and, as such, they are semantically the same type.

2.2 Terms

The expressions of our *source language*, that is the language the programmer uses, are defined as:

$$\begin{array}{lcll} \textbf{Test Types} & \tau & ::= & b \mid \mathbb{0} \to \mathbb{1} \mid \tau \times \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \\ \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi_i e \mid (e{\in}\tau)\,?\,e:e \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array} \tag{5}$$

Expressions are an untyped λ-calculus with constants c, pairs (e,e), pair projections $\pi_i e$, and type-cases. A typecase $(e_0{\in}\tau)\,?\,e_1 : e_2$ 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 τ or e_2 otherwise. Type-cases cannot test arbitrary types but just types of the form τ where the only arrow type that can occur in them is $\mathbb{0} \to \mathbb{1}$, the type of all functions. This means that type-cases can distinguish functions from other values but they cannot distinguish, say, functions that have type $\mathsf{Int}\to\mathsf{Int}$ from those that do not. In previous work on semantic subtyping, there is no such restriction, but this is possible only because λ-abstractions are explicitly annotated with their types. If in the presence of non-annotated λ-abstractions we allowed tests on function types, then in a practical implementation the semantics would depend on the implementation of the type checking or type inference algorithms. Thanks to this restriction, instead, the semantics does not depend on the type system: it can be implemented without keeping track of compile-time types at runtime. Moreover, the interest of the typecase construct in this work is mostly to encode a pattern matching construct. Standard pattern matching cannot check function types, so the restriction is not a problem for this. Typecases of this form also have the same expressiveness as the type-testing primitives of dynamic languages like JavaScript and Racket.

Since every test type τ is also a type, then in what follows we may sometimes use the metavariable t to denote test types when this is clear from the context.

2.3 Reduction Semantics

The reduction semantics is the one of call-by-value pure λ-calculus with products and with a type-case expression. The reduction is given by the following notions of reductions

$$\begin{array}{rcl} (\lambda x.e)v & \rightsquigarrow & e\{v/x\} \\ \pi_1(v_1,v_2) & \rightsquigarrow & v_1 \\ \pi_2(v_1,v_2) & \rightsquigarrow & v_2 \end{array} \qquad \begin{array}{rcll} (v{\in}\tau)\,?\,e_1:e_2 & \rightsquigarrow & e_1 & \text{if } v \in \tau \\ (v{\in}\tau)\,?\,e_1:e_2 & \rightsquigarrow & e_2 & \text{if } v \in \neg\tau \end{array}$$

together with the context rules that implement a leftmost outermost reduction strategy.

The definition uses the standard substitution operation $e\{e'/x\}$ that denotes the capture avoiding substitution of e' for x in e, and the relation $v \in \tau$ that determines whether a value is of a given type or not and holds true if and only if $\mathsf{typeof}(v) \leq \tau$, where $\mathsf{typeof}(\lambda x.e) = \mathbb{0}\to\mathbb{1}$, $\mathsf{typeof}(c) = \boldsymbol{b}_c$, $\mathsf{typeof}((v_1, v_2)) = \mathsf{typeof}(v_1)\times\mathsf{typeof}(v_2)$, and $\boldsymbol{b}_c$ is the unique basic type of the constant c (e.g., $\boldsymbol{b}_{42} = \mathsf{Int}$). Note that $\mathsf{typeof}()$ maps every λ-abstraction to $\mathbb{0} \to \mathbb{1}$, so it does not depend on static types. This approximation is allowed by the restriction on arrow types in typecases.

The language we just defined is the same as the functional core of CDuce defined by Frisch et al. [2002, 2008] bar two important differences. The first difference is that λ-abstractions in [Frisch et al. 2002, 2008] are explicitly annotated with their types while here no annotation is needed. The absence of annotations not only relieves the programmer of an important burden but also, in the case of curried functions, makes it possible to type some expressions that could not be typed with the annotations used in CDuce (see Section 4). The price to pay for this choice is twofold: the burden of finding the annotations for λ-abstractions is passed on the type system and, as explained before, we no longer allow type-cases to test for arbitrary types. The second difference is that the type-case expressions in [Frisch et al. 2002, 2008] introduce a binding since they are of the form $(x{:=}e_\circ{\in}\tau)\ ?\ e_1 : e_2$, that is, the expression binds the result of the tested expression $e_\circ$ to the variable x so that it is possible to specialize the type of x differently for typing e_1 and e_2 and thus implement a limited form of occurrence typing (if $e_\circ : t$, then we assume $x : t \wedge \tau$ when typing e_1 and $x : t \wedge \neg\tau$ when typing e_2). Here we do not ask the programmer to write such a binding: in our system such a binding is deduced by the type system (and this deduction is not limited to type-case expressions). The deduction of this binding is the core of our approach and constitutes the key idea of our generalization of occurrence typing.

2.4 Type System

While the terms, types, and operational semantics of the language are essentially the same as the language by Frisch et al. [2002, 2008], the type inference system is completely different, in particular in what concerns the typing of the type-cases. *Per se* the typing system we detail below is far from being new: it is composed exactly by the rules of the classic system of union and intersection types defined by Barbanera et al. [1995] to which we add standard introduction and elimination rules for products ([×I], [×E$_1$], [×E$_2$]) and the three rules for the type-cases ([$\mathbb{0}$], [$\in_1$], [$\in_2$]). The typing rules are given in Figure 3 and use the following definition of type environments.

Definition 2.2 (Type environment). *Type environments, ranged over by* Γ *are finite sets of mappings from pairwise distinct variables to types. We denote by* $dom(\Gamma)$ *the set of variables mapped by* Γ. *We write* $\Gamma, x : t$ *for the type environment* $\Gamma \cup \{x \mapsto t\}$, *when* Γ *is a type environment such that* $x \notin dom(\Gamma)$. *We write* $\varnothing$ *for the type environment formed by an empty set of mappings.*

If we remove from Figure 3 the three rules for type-cases ([$\mathbb{0}$], [$\in_1$], [$\in_2$]) the resulting system is the same as the one in Definition 3.5 of Barbanera et al. [1995, Definition 3.5] (extended with standard rules for products and constants). The rules for abstractions and applications are those of the simply typed λ-calculus while those for pairs and projections extend it to products. Union and intersection types are handled by the rule [$\wedge$] that introduces intersections, the rule [$\vee$] that eliminates unions, and the subsumption rule [$\leq$] that introduces unions and eliminate intersections.

Notice that, by the definition of type-environments, it is not possible to type two nested λ-abstractions abstracting the same variable and that in the rule [$\vee$] we have that $x \notin \mathrm{dom}(\Gamma)$. It is instead possible to have two distinct (not nested) λ-abstractions with the same abstracted variable, even though from a type-perspective point of view this is not relevant since, as it is

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

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

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

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

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

Fig. 3. Declarative type system

standard in such systems, the deduction is defined modulo α-conversion. In other terms, we suppose the existence of the implicit rule given below on the right (where $\equiv_\alpha$ denotes α-conversion). Working modulo α-conversion is crucial in systems with union types since rule [$\vee$] breaks the α-invariance property (see Hindley and Seldin [2008, Discussion 12.5]). For instance, the following judgement $y : \mathbb{1}\to\mathsf{Bool} \vdash (y(\lambda x.x), y(\lambda x.x)) : (\mathsf{True}\times\mathsf{True}) \vee (\mathsf{False}\times\mathsf{False})$ can be derived in the system above by using [$\vee$] together with the rules for application and pairs (where $\mathsf{Bool} = \mathsf{True}\vee\mathsf{False}$, with True being the singleton type containing the value true,[2] and likewise for False). However, to derive the same type for the α-equivalent term $(y(\lambda x.x), y(\lambda z.z))$ the rule [$\equiv_\alpha$] must be used. In what follows, we will use a variant of the system above where the rules [$\vee$] and [$\wedge$] are replaced by the following rules that produce more compact derivations and are closer to the syntax-directed system given in the next section:

$$[\equiv_\alpha]\ \frac{\Gamma \vdash e : t \qquad e \equiv_\alpha e'}{\Gamma \vdash e' : t}$$

$$[\wedge+]\ \frac{(\forall i \in I) \quad \Gamma \vdash e : t_i}{\Gamma \vdash e : \bigwedge_{i\in I} t_i}\ I \neq \varnothing \qquad [\vee+]\ \frac{\Gamma \vdash e' : \bigvee_{i\in I} t_i \qquad (\forall i \in I) \quad \Gamma, x : t_i \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}\ I \neq \varnothing$$

Although the rules in our system are textually the same as those in [Barbanera et al. 1995, Definition 3.5] there is an important difference, namely, that our types are an extension of those of Barbanera et al. [1995] since they include recursive types, negation types, and the empty type. As a consequence our subsumption rule uses a type-theory that is more general than the one of Barbanera et al. [1995], the theory of semantic subtyping $\mathfrak{C}$ rather than the type theory $\mathfrak{I}$ of Barbanera et al. [1995] of which semantic subtyping is a conservative extension (cf. Dezani-Ciancaglini et al. [2003]). The subsumption rule handles directly the addition of recursive types; negation and empty types are explicitly handled by the rules to type type-case expressions that we comment next.

The combination of the union rule [$\vee$] with the three new rules [$\mathbb{0}$], [$\in_1$], [$\in_2$] is the key novelty of our type-system and, we claim, it captures the essence of occurrence typing. For one thing, thanks to this combination it is possible to type all the examples of [Tobin-Hochstadt and Felleisen 2010], all examples of [Castagna et al. 2021], and several more examples that are captured by neither of these systems. Of course, to paraphrase a famous quotation [see Wikipedia 2021], with great power comes great algorithmic complexity, and thus to determine when an expression is well typed is more challenging with [$\vee$] than in the cited systems. To see how this combination works it may

[2]Henceforth, for every constant of the language we suppose the existence of a singleton type containing that constant.

be useful to see how type-case expressions are typed in the system by Frisch et al. [2002, 2008]. These, we remind, require an explicit binding to perform occurrence typing, and are typed by the following rule:

$$\frac{\Gamma \vdash e' : s \qquad (s\wedge\tau \simeq \mathbb{0} \text{ or } \Gamma, x : s\wedge\tau \vdash e_1 : t) \qquad (s\wedge\neg\tau \simeq \mathbb{0} \text{ or } \Gamma, x : s\wedge\neg\tau \vdash e_2 : t)}{\Gamma \vdash (x{:=}e'{\in}\tau)\ ?\ e_1 : e_2\ :\ t}$$

In a nutshell, we know that e' can yield only a result in s and that this result will be bound to x; so we type e_1 only if e' can yield a result in τ (i.e., $s\wedge\tau \not\simeq \mathbb{0}$) and in that case we can assume that the obtained value (bound to x) is of type $s\wedge\tau$; likewise for e_2. We can type exactly the same type-case by using [$\vee$] combined with [$\mathbb{0}$], [$\in_1$], and/or [$\in_2$] and for that we do not need an explicit binding, since this is taken care of by [$\vee$]: our rules can directly type the expression in which we removed the binding, that is $e = (e'{\in}\tau)\ ?\ (e_1\{e'/x\}) : (e_2\{e'/x\})$, simply by noticing that this expression is equivalent to $((x{\in}\tau)\ ?\ e_1 : e_2)\{e'/x\}$, that $s \simeq (s\wedge\tau) \vee (s\wedge\neg\tau)$, and then applying [$\vee$]. For instance, when both $s\wedge\tau$ and $s\wedge\neg\tau$ are different from $\mathbb{0}$ we have:

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \dfrac{[\in_1]\ \dfrac{[\leq]\ \dfrac{[\text{Ax}]\ \dfrac{}{\Gamma, x{:}s\wedge\tau \vdash x : s\wedge\tau}}{\Gamma, x{:}s\wedge\tau \vdash x : \tau} \qquad \Gamma, x{:}s\wedge\tau \vdash e_1 : t}{\Gamma, x : s\wedge\tau \vdash (x{\in}\tau)\ ?\ e_1 : e_2\ :\ t} \qquad \dfrac{[\in_2]\ \dfrac{[\leq]\ \dfrac{[\text{Ax}]\ \dfrac{}{\Gamma, x{:}s\wedge\neg\tau \vdash x : s\wedge\neg\tau}}{\Gamma, x{:}s\wedge\neg\tau \vdash x : \neg\tau} \qquad \Gamma, x{:}s\wedge\neg\tau \vdash e_2 : t}{\Gamma, x : s\wedge\neg\tau \vdash (x{\in}\tau)\ ?\ e_1 : e_2\ :\ t}}{\Gamma \vdash ((x{\in}\tau)\ ?\ e_1 : e_2)\{e'/x\}\ :\ t}$$

and if either $s\wedge\tau$ or $s\wedge\neg\tau$ is empty, then we replace the corresponding [$\in_i$] rule by [$\mathbb{0}$]. In summary, the combination of [$\vee$] with the three type-cases rule [$\mathbb{0}$], [$\in_1$], and [$\in_2$] encodes and simplifies the system by Frisch et al. [2002, 2008] moving the burden of the binding from the programmer to the type-system. But it also generalizes the approaches for occurrence typing defined by Tobin-Hochstadt and Felleisen [2010] and Castagna et al. [2021] since the binding in [$\vee$] is unconstrained, that is, it is not limited to the occurrences of an expression e' that appear in some particular positions (e.g., in the tests of type-cases [Castagna et al. 2021] or at the root of path expressions [Tobin-Hochstadt and Felleisen 2010]).

2.5 Type Soundness

It is well-known that subject-reduction (i.e., type preservation) does not hold in systems that, like ours, include the rule [$\vee$]. For instance, consider the expression $(f3, f3)$ where $f : \mathbb{1}\to$Bool. Using the rule [$\vee$], it can be typed by (True × True) ∨ (False × False). However, after a step of reduction, we might get for instance the expression $(\text{true}, f3)$, which cannot be typed by (True × True) ∨ (False × False) anymore (the smallest type we can deduce for it is (True × Bool)). Nevertheless, our type system is sound in the sense of Wright and Felleisen [1994]:

Theorem 2.3 (Type Soundness). *If $\varnothing \vdash e : t$, then either e diverges or $e \rightsquigarrow^* v$ with $\varnothing \vdash v : t$.*

For instance, with the earlier example, even if $(\text{true}, f3)$ cannot be typed by (True × True) ∨ (False × False), it will finally reduce to (true, true) which is of type (True × True) ∨ (False × False).

In order to prove this theorem, we introduce an alternative semantics which performs parallel reductions and that satisfies both subject reduction and progress (a typable expression is either a value or can be reduced). These two properties yield soundness for the parallel semantics. Finally, we prove that the parallel semantics is equivalent to the semantics introduced in Section 2.3, in the sense that for any expression e, if e diverges with one semantics then it also diverges with the other, and if e reduces to a value v with one semantics then it also reduces to the value v with the other. From this we deduce the soundness of our system. All the details are given in the appendix.

3 INTERMEDIATE SYSTEM: SYNTAX-DIRECTED RULES AND CANONICAL FORMS

We have seen that the previous system is sound, that is, that every well-typed term can only diverge or yield a result of the same type. Now the problem is to decide whether a given term is well typed or not. For that the rules of Figure 3 are not very useful since they allow too many different possibilities to type a given term. 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*:[3] 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. We tackle each problem in the order.

In the rest of this section we deal with rules that are not syntax directed. These are the intersection rule [∧], the subsumption rule [≤], and the union rule [∨] insofar as the expression in their conclusion can have any form. We adopt different solutions for the rules [∧] and [≤] and for the rule [∨]. For [∧] and [≤] we simply eliminate them and embed the use of intersections and subtyping in the remaining rules. This essentially amounts to resorting to canonical derivations: we prove that it is possible to derive a type for a term if and only if there exists a derivation for that typing judgment in which intersection [∧] and subsumption [≤] rules are used only at determined specific places.[4] For the union rule [∨] we add to the language a *binding* expression whose typing rule will replace the current [∨] rule. Since the binding expressions will explicitly determine both the subterm e' and the variable x to be used in a [∨] instance, then the introduction of bindings also addresses part the non-analyticity of the union rule.

In Section 4 we tackle the non-analyticity problem, or what it remains of it, namely, how to determine the type(s) to assign to a function parameter in a [→I] instance and the split types t_1, t_2 in a [∨] instance. We solve this problem by adding explicit type annotations for the variables bound in bind-expressions and λ-abstractions: these annotations will provide the information that is missing when applying the [→I] and [∨] rules. The rest of this section proceeds as follows:

(1) In Section 3.1 we introduce an intermediate language obtained by adding bind-expressions to the source language of Section 2.2.
(2) In Section 3.2 we define a syntax-directed type system for this intermediate language (with a small caveat for the union rule). We prove that this system is sound and complete with respect to the source language type system, in the sense that every well-typed term of the intermediate language encodes a valid derivation in the source language (soundness) and that every term of the source language is well typed only if there exists a term of the intermediate language that encodes one of its type derivations (completeness).
(3) In Section 3.3 we show that soundness and completeness hold also for a strict sub-language of the intermediate language. We call the terms of this sub-language the *MSC-forms* (maximal sharing canonical forms). The advantage of this sub-language is that there is a one-to-one correspondence between MSC-forms and the expressions of the source language and such a correspondence is effective since it is easy to transform every source language expression into "its" MSC-form.

[3] We consider *non-analytic* (or *synthetic*) a rule in which the input (i.e., Γ and e) of the judgement at the conclusion is not sufficient to determine the inputs of the judgments at the premises (cf. [Martin-Löf 1994; Types 2019]).

[4] Intuitively, a deduction is canonical if (i) subsumption is only used on the premises of application, type-case, union, and projection rules 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]. See the deduction system in Appendix A.3 and Lemmas D.5 and D.6.

In this way, we reduced the problem of typing a source language expression to the one of typing "its" MSC-form, for which a syntax-directed type system exists. In Section 4 we show this to be equivalent to searching for a way to annotate this MSC-form to make it typable. The search for these annotations is performed by the algorithm described in Section 5.

3.1 Expressions with Bindings

Formally, we consider the following grammar of *intermediate expressions* (or *expressions with bindings*) ranged over by the meta-variable $\boldsymbol{e}$ so as to distinguish them from expressions of the source language (*declarative expressions*) for which we continue to use the non-bold symbol e.

$$\textbf{Intermediate exprs} \quad \boldsymbol{e} \ ::= \ c \mid x \mid \lambda x.\boldsymbol{e} \mid \boldsymbol{e}\boldsymbol{e} \mid (\boldsymbol{e},\boldsymbol{e}) \mid \pi_i \boldsymbol{e} \mid (\boldsymbol{e} \in \tau)\,?\,\boldsymbol{e} : \boldsymbol{e} \mid \mathtt{bind}\, x = \boldsymbol{e}\, \mathtt{in}\, \boldsymbol{e} \tag{6}$$

Intermediate expressions extend the syntax of declarative expressions by adding a new construction $\mathtt{bind}\, x = \boldsymbol{e}\, \mathtt{in}\, \boldsymbol{e}$. Given an expression $\mathtt{bind}\, x = \boldsymbol{e}'\, \mathtt{in}\, \boldsymbol{e}$ we call $\boldsymbol{e}'$ the *argument* of the expression and $\boldsymbol{e}$ the *body* of the expression. Such an expression is used to bind a variable to a definition. A bind-expression is different from a let-expression $\mathtt{let}\, x = e'\, \mathtt{in}\, e$ (cf. Appendix C.1) as it is not associated with a call-by-value semantics: $\mathtt{bind}\, x = \boldsymbol{e}'\, \mathtt{in}\, \boldsymbol{e}$ is just a way to indicate that a specific instance of the [∨] rule must be used to type the expression, but it does not force the evaluation of the argument of the bind expression (call-by-need would be appropriate: cf. Appendix A.6).

3.2 Intermediate Typing Rules

We want to define a syntax-directed type-system for the expressions above. The addition of a binding expression makes [∨] syntax-directed, but we still have to eliminate the intersection [∧] and subsumption [≤] rules. In order to define the typing of applications and projections in the absence of subsumption we need some operators on types. Consider the rule [→E] for applications of source language expressions (Figure 3). 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 that 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. 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 $\mathbb{0}\to\mathbb{1}$. 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 an arrow type that subsumes t and has domain s. We need similar operators for projections since 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 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 $\boldsymbol{\pi_2}(t) \stackrel{\text{def}}{=} \min\{u \mid t \leq \mathbb{1}\times u\}$. All these type operators can be effectively computed (cf. Appendix A.4). We have now all the notions we need to define the syntax-directed type system for the intermediate language whose rules are given in Figure 4. The rules for constants, variables, and pairs are omitted since they are the same as in the deduction system for the declarative expressions. Contrary to the previous system, there no longer are explicit rules for intersection and subtyping: we want to have canonical derivations for which the deductions performed by these rules are distributed over the rest of the system. In particular, the only rule that introduces intersections is now the rule [→I-Int] for λ-abstractions. The type subsumption rule [≤] is no longer needed since the

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

$$[\times\text{E}_1\text{-Int}]\ \frac{\Gamma \vdash_{\mathcal{I}} \boldsymbol{e} : t \leq (\mathbb{1} \times \mathbb{1})}{\Gamma \vdash_{\mathcal{I}} \pi_1\boldsymbol{e} : \pi_1(t)} \quad [\times\text{E}_2\text{-Int}]\ \frac{\Gamma \vdash_{\mathcal{I}} \boldsymbol{e} : t \leq (\mathbb{1} \times \mathbb{1})}{\Gamma \vdash_{\mathcal{I}} \pi_2\boldsymbol{e} : \pi_2(t)} \qquad [\mathbb{0}\text{-Int}]\ \frac{\Gamma \vdash_{\mathcal{I}} \boldsymbol{e} : \mathbb{0}}{\Gamma \vdash_{\mathcal{I}} (\boldsymbol{e}\in t)\,?\,\boldsymbol{e}_1 : \boldsymbol{e}_2 : \mathbb{0}}$$

$$[\in_1\text{-Int}]\ \frac{\Gamma \vdash \boldsymbol{e} : t_0 \leq t \qquad \Gamma \vdash \boldsymbol{e}_1 : t_1}{\Gamma \vdash (\boldsymbol{e}\in t)\,?\,\boldsymbol{e}_1 : \boldsymbol{e}_2 : t_1}\ t_0 \not\simeq \mathbb{0} \quad [\in_2\text{-Int}]\ \frac{\Gamma \vdash_{\mathcal{I}} \boldsymbol{e} : t_0 \leq \neg t \qquad \Gamma \vdash \boldsymbol{e}_2 : t_2}{\Gamma \vdash_{\mathcal{I}} (\boldsymbol{e}\in t)\,?\,\boldsymbol{e}_1 : \boldsymbol{e}_2 : t_2}\ t_0 \not\simeq \mathbb{0}$$

$$[\vee_1\text{-Int}]\ \frac{\Gamma \vdash_{\mathcal{I}} \boldsymbol{e}_2 : s}{\Gamma \vdash_{\mathcal{I}} \mathtt{bind}\, x = \boldsymbol{e}_1\, \mathtt{in}\, \boldsymbol{e}_2 : s}\ x \notin \mathrm{dom}(\Gamma) \quad [\vee_2\text{-Int}]\ \frac{\Gamma \vdash_{\mathcal{I}} \boldsymbol{e}_1 : \bigvee_{j\in J} t_j \quad {\scriptstyle(\forall j\in J)}\ \Gamma, x{:}t_j \vdash_{\mathcal{I}} \boldsymbol{e}_2 : s_j}{\Gamma \vdash_{\mathcal{I}} \mathtt{bind}\, x = \boldsymbol{e}_1\, \mathtt{in}\, \boldsymbol{e}_2 : \bigvee_{j\in J} s_j}\ J \neq \varnothing$$

Fig. 4. Intermediate typing rules

checks for the subtyping relation are performed in the elimination rules and in the two $[\in_i]$ rules. The [→E-Int] rule works as we explained above: (*i*) it checks that the type t_1 of the expression in the function position is functional (i.e., $t_1 \leq \mathbb{0}\to\mathbb{1}$); (*ii*) it checks that the type t_2 of the argument is contained the domain of the function (i.e, $t_2 \leq \mathrm{dom}(t_1)$), and (*iii*) it returns the type $t_1 \circ t_2$ of the application. The product elimination rules check whether the argument of the projection is a product (i.e., $t \leq \mathbb{1}\times\mathbb{1}$) and apply the corresponding type operator on this type.

The three rules for type-case expressions are essentially as before, bar two minor modifications. First, in the $[\in_i$-Int] rules the checked expression must be explicitly subsumed to either t or $\neg t$ since there no longer is a [≤] rule in our system to do that. Second, since we want our type-system to be syntax-directed, then we add the side condition $t_0 \not\simeq \mathbb{0}$ in the $[\in_i]$ rules so as to avoid any overlap with the [𝟘] rule and thus giving priority to the latter.

Finally, the [∨] rule (actually, [∨+]) is encoded by a binding expression. We split the [∨+] in two rules. One, [$\vee_1$-Int], is for the case when the variable x in $\boldsymbol{e}_2$ is not reachable (either because it is not free in $\boldsymbol{e}_2$ or because it is in a type-case branch that cannot be selected such as in $(42\in\mathsf{Int})\,?\,3 : x$). The other, [$\vee_2$-Int], is the normal case for [∨+] where the bind-expression determines the variable x and the expression $\boldsymbol{e}_1$ to be substituted for it, but does not specify how to split the type of $\boldsymbol{e}_1$ into a union of types. Notice that in [$\vee_1$-Int] we added the side condition $x \notin \mathrm{dom}(\Gamma)$: this condition is not necessary in [$\vee_2$-Int] since (as in [∨] and [∨+]) the environment Γ is extended and, by Definition 2.2, the extension $\Gamma, x : t_j$ is defined only if $x \notin \mathrm{dom}(\Gamma)$.

The system is syntax-directed: the form of the expression determines the rules to apply and the rules for a same form do not overlap (for bindings, [$\vee_1$-Int] must be used only if $\boldsymbol{e}_1$ is not typable: we will be more precise in Section 4).

Soundness and completeness. A well-typed expression of the intermediate language is typed by derivations in which every instance of the [∨] rule corresponds to a bind-expression. Any such derivation is also a canonical derivation for a particular expression of the source language. 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 the 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 (cf. Appendix A.5).

We can now prove that the problem of typing a declarative expression 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. This is formally stated by the theorems of soundness and completeness of the intermediate system:

Theorem 3.1 (Soundness). *If* $\Gamma \vdash_{\mathcal{I}} \boldsymbol{e} : t$ *then* $\Gamma \vdash \lceil \boldsymbol{e} \rceil : t$

Theorem 3.2 (Completeness). *If* $\Gamma \vdash e : t$ *then* $\exists \boldsymbol{e}', t'$ *such that* $\lceil \boldsymbol{e}' \rceil \equiv_\alpha e$, $t' \leq t$, *and* $\Gamma \vdash_{\mathcal{I}} \boldsymbol{e}' : t'$

3.3 Maximal Sharing Canonical Forms

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

$$\begin{array}{lrcl} \textbf{\textit{Atomic expressions}} & \boldsymbol{a} & ::= & c \mid \lambda x.\boldsymbol{\kappa} \mid (x,x) \mid xx \mid (x{\in}\tau)\,?\,x:x \mid \pi_i x \\ \textbf{\textit{MSC-forms}} & \boldsymbol{\kappa} & ::= & x \mid \mathtt{bind}\, x = \boldsymbol{a}\, \mathtt{in}\, \boldsymbol{\kappa} \end{array} \tag{7}$$

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 possibly 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. MSC-forms are defined modulo α-conversion.[5] Since MSC-forms are also intermediate expressions, then the typing rules and the definition of unwinding for intermediate terms of Section 3.2 apply to MSC-forms, too.

The syntactic form of MSC-forms guarantees that if a source language expression e is the unwinding of an MSC-form $\boldsymbol{\kappa}$, then every distinct sub-expression of e is bound by a variable in $\boldsymbol{\kappa}$, while the first property of Definition 3.3 guarantees that distinct variables bind distinct (i.e., non α-convertible) sub-expressions (i.e., this first property enforces the *maximal sharing* of common sub-expressions). The second property requires that bind-expressions must extrude λ-abstractions whenever possible. The third property guarantees that in a MSC-form there is no useless binding.

The first two properties of Definition 3.3 are important since they ensure that an expression of the source language is typable *if and only if* it is the unwinding of a typable MSC-form. For the first property, this is because reducing the bindings in an intermediate expression—while preserving unwinding—increases the typeability of a term: if we can type an intermediate term in which two distinct variables bind the same sub-expression, then the same term in which this sub-expression is bound by a single variable can also be typed by assigning to the unique variable the intersection of the types of the distinct variables, but the converse does not hold. For the second property this is because outer bindings may produce better types. For instance, consider the expression $\mathtt{bind}\, x = \boldsymbol{a}\, \mathtt{in}\, \lambda y.\, x$, where $\boldsymbol{a}$ is an expression that can be either an integer or a Boolean. This expression can be typed with $(\mathbb{1} \to \mathsf{Int}) \vee (\mathbb{1} \to \mathsf{Bool})$. However, for the expression $\lambda y.\ (\mathtt{bind}\, x = \boldsymbol{a}\, \mathtt{in}\, x)$ which has the same unwinding as the previous one, the most precise type one can deduce is $\mathbb{1} \to (\mathsf{Int} \vee \mathsf{Bool})$, which is strictly larger than $(\mathbb{1} \to \mathsf{Int}) \vee (\mathbb{1} \to \mathsf{Bool})$.

The last property of Definition 3.3 is important because it ensures that given a source language expression e there exists a unique (modulo α-conversion and the order of bindings) MSC-form whose unwinding is e (cf. Proposition 3.5): we denote this MSC-form by $\mathsf{MSC}(e)$.

Given a source language expression e it is easy to produce its unique MSC-form $\mathsf{MSC}(e)$. For space constraints we give the formal definition of the transformation and all details in the Appendixes A.7

[5] For instance, both $\lambda x.\mathtt{bind}\, z = xy\, \mathtt{in}\, zy$ and $\lambda x.\mathtt{bind}\, z = xy\, \mathtt{in}\, z$ are two distinct atoms that can occur in the same MSC-form, even though the atom xy appears in both: an α-renaming of x makes the first MSC-property hold.

and A.8, but in practice what the transformation needs to do is to visit e bottom up and generate a distinct binding for each distinct sub-expression, taking special care for free-variables, when extruding abstractions, and for α-convertible expressions (see Section 6 for more details).

We just got rid of the problem of determining where to put the bindings in a source language expression e: generate $\mathsf{MSC}(e)$ and try to type it in the syntax-directed system of Section 3.2.

Formally, we define the following congruence on MSC-forms:

Definition 3.4 (Canonical equivalence). *We denote by $\equiv_\kappa$ the smallest congruence on MSC-forms that is closed by α-conversion and such that*

$$\mathtt{bind}\, x_1 = \boldsymbol{a}_1 \,\mathtt{in}\, \mathtt{bind}\, x_2 = \boldsymbol{a}_2 \,\mathtt{in}\, \boldsymbol{\kappa} \quad \equiv_\kappa \quad \mathtt{bind}\, x_2 = \boldsymbol{a}_2 \,\mathtt{in}\, \mathtt{bind}\, x_1 = \boldsymbol{a}_1 \,\mathtt{in}\, \boldsymbol{\kappa} \qquad x_1 \notin fv(\boldsymbol{a}_2), x_2 \notin fv(\boldsymbol{a}_1)$$

Then we prove that all the MSC forms of a source language expression are equivalent:

Proposition 3.5. *If $\boldsymbol{\kappa}_1$ and $\boldsymbol{\kappa}_2$ are two MSC-forms and $\lceil \boldsymbol{\kappa}_1 \rceil \equiv_\alpha \lceil \boldsymbol{\kappa}_2 \rceil$, then $\boldsymbol{\kappa}_1 \equiv_\kappa \boldsymbol{\kappa}_2$.*

(Proof hint). Given two MSC-forms with the same unwinding, conditions (1) *and* (3) *of Definition 3.3 ensure that there is a one-to-one correspondence between their bindings, and condition* (2) *that each binding is located in the outermost possible λ. So the two MSC-forms differ by the relative order between independent bindings that are in the same λ-abstraction or at top-level and, thus, are equivalent.* □

It is easy to observe that the canonical equivalence preserves typeability (this is a direct consequence that type environments are mappings in which order does not matter). Thus, the corollary of this proposition is that an expression e is typable if and only if its unique (modulo $\equiv_\kappa$) MSC-form is typable, too. Formally, let $\mathsf{MSC}(e)$ be any element of the set $\{\boldsymbol{\kappa} \mid e \equiv_\alpha \lceil \boldsymbol{\kappa} \rceil\}$, then

Corollary 3.6 (Soundness and Completeness). *For every closed term e of the source language*

$$\vdash e : t \quad \Rightarrow \quad \vdash_{\mathcal{I}} \mathsf{MSC}(e) : t' \leq t \qquad \textit{(completeness)}$$
$$\vdash e : t \quad \Leftarrow \quad \vdash_{\mathcal{I}} \mathsf{MSC}(e) : t \qquad \textit{(soundness)}$$

Corollary 3.6 states that given a source language expression e it 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 a syntax-directed type system.

4 ALGORITHMIC SYSTEM: ADDING TYPE ANNOTATIONS

The intermediate type system of Figure 4 is not algorithmic since it still contains non-analytic rules: we neither know which decomposition in t_i's to use when applying the [$\vee_2$-Int] rule, nor which t_j's to choose when applying the [$\to$I-Int] rule. To solve this problem, we enrich our intermediate language with *annotations* that determine the types to use when typing a bind expression or a λ-abstraction. It is then straightforward to define an algorithmic system (i.e., a syntax-directed system composed only of analytic rules) for these enriched expressions and prove it to be sound and complete with respect to the system of the intermediate language.

Annotations. In a nutshell, 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 is assumed 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, $\mathtt{bind}\, x{:}\{t_1, \ldots, t_n\} = \boldsymbol{e}_1 \,\mathtt{in}\, \boldsymbol{e}_2$ stands for $\mathtt{bind}\, x{:}\{\varnothing \triangleright t_1, \ldots, \varnothing \triangleright t_n\} = \boldsymbol{e}_1 \,\mathtt{in}\, \boldsymbol{e}_2$.

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 $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int}) \wedge (\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$. 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 $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int}) \wedge (\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$. Clearly $A_1 = \{\mathsf{Int}, \mathsf{Bool}\}$. However, the typing of the parameter y depends on the typing of x: when $x{:}\mathsf{Int}$ then y must have type Int (the type of y must be larger than the one of x—the argument it will be bound to—, but also smaller than Int so as to deduce that $\lambda y.(x, y)$ returns a pair in $\mathsf{Int}\times\mathsf{Int}$). Likewise when $x{:}\mathsf{Bool}$, then y must be of type Bool, too. Therefore, we use as A_2 the annotation $\{x{:}\mathsf{Int}\triangleright\mathsf{Int}, x{:}\mathsf{Bool}\triangleright\mathsf{Bool}\}$, which precisely states that when $x{:}\mathsf{Int}$, then we must suppose that y (the variable annotated by A_2) is of type Int, and likewise for Bool. Using the typing rules we describe in the next section we are then able to deduce that $\lambda x{:}\{\mathsf{Int}, \mathsf{Bool}\}.(\lambda y{:}\{x{:}\mathsf{Int}\triangleright\mathsf{Int}, x{:}\mathsf{Bool}\triangleright\mathsf{Bool}\}.(x, y))x$ has type $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int}) \wedge (\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$. Because of this last form of annotations, these annotations are strictly more expressive than those of the functional core of CDuce presented in [Frisch et al. 2008]: even if CDuce is a calculus with explicit full annotations, it is not possible to decorate $\lambda x.((\lambda y.(x, y))x)$ with a (CDuce) annotation so that the resulting term has type $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int}) \wedge (\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$: CDuce annotations are exactly as expressive as our $\{\varnothing\triangleright t_1, \ldots, \varnothing\triangleright t_n\}$ annotations.

In the preceding paragraphs we explained how our annotations work by using as examples expressions of the intermediate language. However, recall that our ultimate goal is to type the expressions of the *source language* we introduced in Section 2.2 and, as we saw in the previous section, for that one does not need to consider the whole intermediate language: the MSC-forms suffice. This is why in the rest of this section we add annotations *only to MSC-forms*: the definitions and results of this section can be easily extended to all expressions of the intermediate language.

4.1 Algorithmic Expressions and Typing Rules

Formally, we consider the following grammar of *explicitly annotated MSC-forms* (or *algorithmic expressions*)—i.e., MSC-forms as per Definition 3.3 enriched with annotations—ranged over by the meta-variable κ (to distinguish them from the unannotated MSC-forms ranged over by $\boldsymbol{\kappa}$).

$$\begin{array}{lrcl} \textbf{Annotations} & A & ::= & \{\Gamma\triangleright t, \ldots, \Gamma\triangleright t\} \\ \textbf{Algorithmic Atoms} & a & ::= & c \mid \lambda x{:}A.\kappa \mid (x, x) \mid xx \mid (x{\in}\tau)\,?\,x : x \mid \pi_i x \\ \textbf{Algorithmic Expressions} & \kappa & ::= & x \mid \mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa \end{array} \tag{8}$$

These enrich the syntax of MSC-forms by inserting an annotation wherever a variable is introduced by a binder (either a "λ" or a "bind"). We use φ to range over either atoms a or expressions κ.

For the algorithmic typing rules we need to define the following pre-order on type environments:

Definition 4.1 (Envinronment subsumption). *Given two type environments* Γ *and* Γ'*, we say that* Γ' subsumes Γ*, written,* $\Gamma \le \Gamma'$ *if and only if* $\forall x \in dom(\Gamma')$ *we have* $\Gamma(x) \le \Gamma'(x)$.

The algorithmic type system is then given by the rules for abstractions and binding in Figure 5 plus all the other rules of the intermediate type system (specialized for MSC-forms, i.e., where every subexpression is a variable: cf. Figure 4 and Appendix A.9). The introduction of intersections is driven by the annotation $\{\Gamma_i\triangleright t_i\}_{i\in I}$ of the λ-abstraction: for every t_j whose hypotheses Γ_j subsume the current environment Γ, the system checks whether the function has type $t_j \to s_j$, that is, under the hypothesis that $x : t_j$ it tries to infer a type s_j for the body κ of the function; the condition $J \neq \varnothing$ ensures that at least one Γ_i subsumes Γ. Bind expressions are still handled by the two rules [$\vee_1$-Int] and [$\vee_2$-Int]. The first one, [$\vee_1$-Int], is for the case when the variable x is not reachable in κ. It is used when the current environment Γ is not subsumed by any of the environments in

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

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

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

Fig. 5. Algorithmic typing rules

the annotation of x. The other, [$\vee_2$-Alg], is the normal case for [$\vee$+] where the bind-expression determines both the variable x and the atom a to be substituted for it, and where its annotation determines how to split the type of a into a union of types.

The first important property satisfied by the algorithmic type system is that it is syntax directed and composed only by analytic rules (a simple visual check of the rules in Appendix A.9 suffices to verify it). As such it describes a deterministic algorithm to infer the type of an algorithmic expression. Furthermore, it is also decidable (this can be proved by a simple induction on the structure of the term, by observing that the subtyping relation is decidable, and that the operators used in the rules can be effectively computed: cf. Frisch et al. [2008, Section 6] and Appendix A.4).

The main interest of the algorithmic system is that a well-typed algorithmic term univocally encodes a type derivation for a MSC-form and, in virtue of Corollary 3.6, it also encodes a particular canonical derivation for a source language expression. The source language expression at issue is the one obtained by erasing the annotations from our algorithmic term and then applying the unwinding operation defined in Section 3.2. The annotation erasing operation, noted $\langle . \rangle$, is defined as $\langle \mathtt{bind}\, x{:}A = a\ \mathtt{in}\ \kappa \rangle \stackrel{\text{def}}{=} \mathtt{bind}\, x = \langle a \rangle\ \mathtt{in}\ \langle \kappa \rangle$, $\langle \lambda x{:}A.\kappa \rangle \stackrel{\text{def}}{=} \lambda x.\langle \kappa \rangle$, and as the identity otherwise. We can now prove that the problem of typing an MSC-form is equivalent to the problem of decorating it with some annotations that make it typable with the algorithmic type system. This is formally stated by the theorems of soundness and completeness of the system for algorithmic expressions (ranged over by κ) *with respect to the unannotated MSC-forms* (ranged over by $\boldsymbol{\kappa}$):

Theorem 4.2 (Soundness). *If* $\Gamma \vdash_{\mathcal{A}} \kappa : t$ *then* $\Gamma \vdash_{\mathcal{I}} \langle \kappa \rangle : t$

Theorem 4.3 (Completeness). *If* $\Gamma \vdash_{\mathcal{I}} \boldsymbol{\kappa} : t$, *then* $\exists \kappa$ *such that* $\langle \kappa \rangle = \boldsymbol{\kappa}$ *and* $\Gamma \vdash_{\mathcal{A}} \kappa : t' \le t$

Soundness states that if an algorithmic expression is well-typed, then removing its annotations gives a well-typed MSC-form. Completeness states that every well-typed MSC-form can be decorated with annotations so that it becomes a well-typed algorithmic expression.

By combining these theorems with the previous results on the intermediate language, we obtain the soundness and completeness of the algorithmic system *with respect to the source language*.

Corollary 4.4 (Algorithmic soundness and completeness).

$$\vdash_{\mathcal{A}} \kappa : t \quad \Rightarrow \quad \vdash \lceil \langle \kappa \rangle \rceil : t \qquad \textit{(soundness)}$$

$$\vdash e : t \quad \Rightarrow \quad \exists \kappa\ .\ \vdash_{\mathcal{A}} \kappa : t' \le t \text{ and } \lceil \langle \kappa \rangle \rceil \equiv_\alpha e \qquad \textit{(completeness)}$$

Notice that in the statement of completeness $\lceil \langle \kappa \rangle \rceil \equiv_\alpha e$ is equivalent to writing $\langle \kappa \rangle = \mathsf{MSC}(e)$. Combining this with the soundness result yields that for every closed declarative expression e, if $\vdash_{\mathcal{A}} \kappa : t$ and $\langle \kappa \rangle = \mathsf{MSC}(e)$, then $\vdash e : t$. All this gives us a procedure to check whether an expression e of the source language is well typed or not: produce $\mathsf{MSC}(e)$ and look for a way to annotate it so that it becomes a well-typed algorithmic expression κ. If we find such annotations, then the soundness property tells us that e is well typed. If such annotations do not exist, then the completeness property tells us that e is not well-typed.

In the next section we describe a sound algorithm to search for such annotations.

5 ALGORITHM FOR RECONSTRUCTING ANNOTATIONS

We define an algorithm to reconstruct annotations for an MSC-form to make it become a well-typed algorithmic expression. It starts by annotating all the bindings of the MSC-form by $\mathbb{1}$ (the weakest possible annotation) and performs several passes to refine these annotations until it either obtains a well-typed algorithmic expression or it fails.

At each pass the algorithm takes as input a type environment Γ, an algorithmic expression κ, and a type t and checks whether κ can be given the type t under the hypothesis Γ. The check yields one of three possible results: either (i) success with an expression κ' obtained from κ by refining some annotations—meaning that it is possible to deduce from Γ the type t for κ'—, or (ii) a failure—meaning that the algorithm cannot propose any better refinement of the annotations to type the expression—, or (iii) a refined expression κ' and a set of type environments that refine Γ—meaning that it is not yet possible to deduce t for κ' under Γ and that the algorithm must try further passes using the new type environments returned by this pass.

Formally, passes are defined by a deduction system whose judgments are of the form $\Gamma \vdash_{\mathcal{R}} \varphi : t \Rightarrow (\varphi', \mathbb{\Gamma})$, where $\mathbb{\Gamma}$ denotes a possibly empty set of type environments, t is a type, and φ, φ' are either algorithmic atoms or algorithmic expressions as defined in (8). Γ, φ, and t form the input of the pass while φ' and $\mathbb{\Gamma}$ are the output and they refine φ and Γ, respectively. The reason why the output is a pair is because we want to refine the type of *all* the variables in the input expression φ: the types of the variables that are free in φ are refined by providing new environments that refine the input environment Γ, yielding $\mathbb{\Gamma}$; the types of the variables that are bound in φ are refined by refining their annotations in φ, yielding φ'.

Given a judgment $\Gamma \vdash_{\mathcal{R}} \varphi : t \Rightarrow (\varphi', \mathbb{\Gamma})$, an empty $\mathbb{\Gamma}$ means failure while if $\mathbb{\Gamma}$ is the singleton $\{\Gamma\}$, this means success. For instance, the rules for constants in the deduction system are:

$$[\textsc{Const}]\ \frac{\boldsymbol{b}_c \leq t}{\Gamma \vdash_{\mathcal{R}} c : t \Rightarrow (c, \{\Gamma\})} \qquad [\textsc{ConstUntypable}]\ \frac{\boldsymbol{b}_c \not\leq t}{\Gamma \vdash_{\mathcal{R}} c : t \Rightarrow (c, \{\})}$$

The system succeeds in checking that a constant c has a supertype of $\boldsymbol{b}_c$ and fails otherwise. Notice that the atom in the result is the same as in the input, since in a constant there is no annotation to refine (this is true for all the rules for atoms excluding λ-abstractions).

If a pass neither fails nor succeeds, then it proposes a refinement of the types of the variables in the expression, refinement that is to submit to a further pass. This corresponds to a judgment $\Gamma \vdash_{\mathcal{R}} \varphi : t \Rightarrow (\varphi', \{\Gamma_1, ..., \Gamma_n\})$: for the variables that are bound in φ it proposes a refinement by refining the annotations in φ yielding the new expression φ'; for the variables that are free in φ, it proposes a refinement of the environment Γ by proposing the refinements $\Gamma_1 \dots \Gamma_n$. The type of a variable is refined in two ways: either it can be restricted to match the usage of the variable or it can be split when it is the union of simpler types (to determine a unique split of unions, the rules use the disjunctive normal forms of [Frisch et al. 2008]: cf. Appendix B.1). For instance, if we try to type the atom $\pi_1 x$ and the current annotation/environment for x is a type t that does not contain only pairs, then the algorithm proposes to refine the type of x into $t \wedge (\mathbb{1}\times\mathbb{1})$; if the type t is a union of products such as $(s_1\times s_2) \vee (t_1\times t_2)$, then the algorithm suggests to split the type of x into two separate types $(s_1\times s_2)$ and $(t_1\times t_2)$. Both these refinements are done by the rules for projections:

$$[\textsc{Proj}_1]\ \frac{\Gamma(x) \wedge (t \times \mathbb{1}) \simeq \bigvee_{i\in I} t_i \times s_i}{\Gamma \vdash_{\mathcal{R}} \pi_1 x : t \Rightarrow (\pi_1 x, \{\Gamma[x :\hat{=} t_i\times s_i]\}_{i\in I})} \qquad [\textsc{Proj}_2]\ \frac{\Gamma(x) \wedge (\mathbb{1} \times t) \simeq \bigvee_{i\in I} t_i \times s_i}{\Gamma \vdash_{\mathcal{R}} \pi_2 x : t \Rightarrow (\pi_2 x, \{\Gamma[x :\hat{=} t_i\times s_i]\}_{i\in I})}$$

where $\Gamma[x :\hat{=} t]$ is the environment obtained by *refining* the binding of x in Γ by intersecting it with t, that is, $\Gamma[x :\hat{=} t] \stackrel{\text{def}}{=} (\Gamma \setminus \{x \mapsto \Gamma(x)\}) \cup \{x \mapsto \Gamma(x) \wedge t\}$ for $x \in \text{dom}(\Gamma)$. The rules force the projections to have the checked type t by intersecting the type of x with $(t\times\mathbb{1})$ or $(\mathbb{1}\times t)$, and split

the resulting type into the different summands by proposing different refinements of the current environment Γ. Again, the returned atom is the same as in the input, since there are no annotations to refine in it. The inference for occurrence typing is performed by the rule for type-cases:

$$[\textsc{Case}]\ \frac{}{\Gamma \vdash_{\mathcal{R}} (x{\in}s)\,?\,x_1 : x_2 : t \Rightarrow ((x{\in}s)\,?\,x_1 : x_2, \{\Gamma[x :\hat{=} s][x_1 :\hat{=} t], \Gamma[x :\hat{=} \neg s][x_2 :\hat{=} t]\})}$$

In order to analyze the test that x has type s, the rule [Case] splits the current type of x by intersecting it with s and $\neg s$, a split that it proposes in the two refinements $\Gamma[x :\hat{=} s]$ and $\Gamma[x :\hat{=} \neg s]$ given in the result of the conclusion. The first refinement corresponds to the selection of the "then" branch, that is of x_1. Since the rule requires the whole expression to be of type t, then the first type environment also refines the type of x_1 with t. Likewise for the second environment and x_2. An important though pretty hidden detail is that the notation for the refinement of type environments handles the cases in which two or more variables coincide: for instance if $x = x_1$, then the rule [Case] will refine $\Gamma(x)$ in the first refinement by intersecting it both with s and with t. In all the rules presented in this section we suppose that the intersections occurring in them are not empty: the cases for empty types are handled by other rules, omitted here (see Appendix B.3).

The reason why we may split the type of a variable x when its type is a union or when we test it dynamically can be understood by considering the typing rule [$\vee_2$-Alg] in Figure 5: we are trying to reconstruct the annotation for x in the conclusion of [$\vee_2$-Alg] and thus determine the environments compatible with the current one that should be used in this annotation. However, [$\vee_2$-Alg] is not the only rule that splits the derivation in sub-derivations with different environments: also [$\to$I-Alg] does it, with the difference that it does not split a union or a tested case, but it splits an intersection of arrows. We encounter this split of intersections in the rules for applications, in particular in [AppR]. These rules are defined as follows (we present a simplified version):

$$[\textsc{AppR}]\ \frac{\Gamma(x_1) \simeq \bigwedge_{i\in I}(s_i \to t_i)}{\Gamma \vdash_{\mathcal{R}} x_1x_2 : t \Rightarrow (x_1x_2, \{\Gamma[x_1 :\hat{=} ((s_i \wedge \Gamma(x_2)) \to t)][x_2 :\hat{=} s_i]\}_{i\in I})}$$

$$[\textsc{AppL}]\ \frac{\Gamma(x_1) \wedge (\mathbb{0} \to \mathbb{1}) \simeq \bigvee_{i\in I} s_i}{\Gamma \vdash_{\mathcal{R}} x_1x_2 : t \Rightarrow (x_1x_2, \{\Gamma[x_1 :\hat{=} s_i]\}_{i\in I})}$$

The type of a functional expression is in general a union of intersections of arrows (cf. Frisch et al. [2008]). When it is a union then, as for any other algorithmic atom, the system splits this union into distinct type environments, here in the rule [AppL]. Then each summand (which is an intersection of arrows) is separately checked by the rule [AppR] which deserves a detailed explanation. The hypothesis about x_1 is that it is bound to a function whose type is an intersection of arrows. For each arrow $s_i \to t_i$ in this intersection, the rule proposes a refined environment Γ_i in which both the type of x_1 and the type of x_2 are refined: the first by intersecting it with $(s_i \wedge \Gamma(x_2)) \to t$, to ensure that the application uses the right domain and will yield a result not only in t_i (since $\Gamma(x_1) \leq s_i \to t_i$), but also in t (and, thus, in $t_i \wedge t$); the second by intersecting it with s_i since the argument must be in the domain of x_1. The intuition is that these different Γ_i correspond to the different typing checks performed by the rule [$\to$I-Alg] to type the body of the function bound to x_1. It is important to stress that, as in previous cases, an environment Γ_i is added to the result only if $\Gamma(x_2) \wedge s_i$ is not empty since we want to discard from the type of x_1 the arrows whose domain is not compatible with the current typing of the argument. Likewise, if the intersection in [AppL] is empty, then this produces an empty set of refinements, meaning failure since we are applying to some argument an expression that is not a function.

All the rules we have seen so far are actually axioms (we only considered atoms in which the only subexpressions are variables) in which the returned expression is the same as in the input (there are no annotations to refine). The bulk of the inference work is done in the rules for binding expressions

$$[\textsc{BindArgSkip}]\ \frac{(\Gamma \triangleright A) = \{\} \qquad \Gamma \vdash_{\mathcal{R}} \kappa : t \Rightarrow (\kappa', \mathbb{\Gamma})}{\Gamma \vdash_{\mathcal{R}} \mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa\ : t \Rightarrow (\mathtt{bind}\, x{:}\{\} = a\, \mathtt{in}\, \kappa', \mathbb{\Gamma})}$$

$$[\textsc{BindArgUntyp}]\ \frac{\Gamma \vdash_{\mathcal{R}} a : \bigvee(\Gamma \triangleright A) \Rightarrow (a', \{\}) \qquad \Gamma \vdash_{\mathcal{R}} \kappa : t \Rightarrow (\kappa', \mathbb{\Gamma})}{\Gamma \vdash_{\mathcal{R}} \mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa\ : t \Rightarrow (\mathtt{bind}\, x{:}\{\} = a'\, \mathtt{in}\, \kappa', \mathbb{\Gamma})}$$

$$[\textsc{BindArgRefEnv}]\ \frac{\Gamma \vdash_{\mathcal{R}} a : \bigvee(\Gamma \triangleright A) \Rightarrow (a', \mathbb{\Gamma})}{\Gamma \vdash_{\mathcal{R}} \mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa\ : t \Rightarrow (\mathtt{bind}\, x{:}A = a'\, \mathtt{in}\, \kappa, \mathbb{\Gamma} \cup \{\Gamma\})}\ (\mathbb{\Gamma} \neq \{\Gamma\})$$

$$[\textsc{BindArgRefAnns}]\ \frac{\Gamma \vdash_{\mathcal{R}} a : \bigvee(\Gamma \triangleright A) \Rightarrow (a', \{\Gamma\}) \quad \Gamma \vdash_{\mathcal{R}} \mathtt{bind}\, x{:}A = a'\, \mathtt{in}\, \kappa\ : t \Rightarrow (\kappa', \mathbb{\Gamma})}{\Gamma \vdash_{\mathcal{R}} \mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa\ : t \Rightarrow (\kappa', \mathbb{\Gamma})}\ (a' \neq a)$$

$$[\textsc{Bind}]\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{R}} a : \bigvee(\Gamma \triangleright A) \Rightarrow (a, \{\Gamma\}) \qquad \Gamma \vdash_{\mathcal{A}} a : s \qquad \{s_i\}_{i \in I} = \mathtt{partition}(\{s \wedge u \mid u \in (\Gamma \triangleright A)\}) \\ (\forall i \in I)\ \ \Gamma, (x : s_i) \vdash_{\mathcal{R}} \kappa : t \Rightarrow (\kappa_i, \mathbb{\Gamma}_i) \quad \mathbb{\Gamma}'_i = \mathtt{propagate}_{x,a,s_i}(\mathbb{\Gamma}_i) \quad (A_i, \mathbb{\Gamma}''_i) = \mathtt{extract}_x(\mathbb{\Gamma}'_i)\end{array}}{\Gamma \vdash_{\mathcal{R}} \mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa\ : t \Rightarrow (\mathtt{bind}\, x{:}\bigcup_{i \in I} A_i = a'\, \mathtt{in}\, \mathtt{merge}(\{\kappa_i\}_{i \in I}), \bigcup_{i \in I} \mathbb{\Gamma}''_i)}$$

Fig. 6. Reconstruction rules for bind-expressions

(and in those for λ-abstractions that are conceptually similar to those for bindings). The complete set of rules for bind-expressions are presented in Figure 6 in their priority order: a rule can be applied only if the previous rules cannot (we just did two slight simplifications in the first and third rule: cf. Appendix B.3). These rules use some auxiliary definitions. We note by $(\Gamma \triangleright A)$ the set of the types of the annotation A that are compatible with Γ, that is, $(\Gamma \triangleright A) \stackrel{\text{def}}{=} \{t \mid \Gamma' \triangleright t \in A \text{ and } \Gamma \leq \Gamma'\}$, and by $\bigvee(\Gamma \triangleright A)$ the union of these types, that is, $\bigvee(\Gamma \triangleright A) \stackrel{\text{def}}{=} \bigvee_{t \in (\Gamma \triangleright A)} t$.

To check that the bind-expression $\mathtt{bind}\, x{:}A = a\, \mathtt{in}\, \kappa$ has type t under the hypotheses Γ, the system first focuses on the argument a of the bind-expression using the first four rules in Figure 6. If no type in A is compatible with the current environment Γ (i.e., $(\Gamma \triangleright A)$ is empty), then the bind is skipped and the system tries to type the body κ of the expression without using x: no type assumption for x is added to Γ and the annotation is emptied (rule [BindArgSkip]). If instead $(\Gamma \triangleright A)$ is not empty, then the argument a must have a type smaller than the union of all types in $(\Gamma \triangleright A)$. Thus the system tries to check under the current hypothesis Γ whether a can be given the type $\bigvee(\Gamma \triangleright A)$. The result of this check is a pair $(a', \mathbb{\Gamma})$ according to which we can distinguish four different outcomes, corresponding to the last four rules. (1) the check failed, that is, it returned an empty $\mathbb{\Gamma}$ (rule [BindArgUntyp]): then this binding must be skipped and we proceed as for rule [BindArgSkip]. (2) the check did not succeed but it proposed a set of refinements for Γ (and, possibly, for a), that is, $\mathbb{\Gamma} \neq \{\Gamma\}$ ([BindArgRefEnv]): then before attacking the body κ of the bind expression, the system proposes these refinements for the whole bind-expression updated with the refined argument a'; furthermore, since the first premise of the rules does not guarantee a to be well-typed, then the current environment Γ is also returned for the cases in which this should fail. (3) the test succeeded (i.e., $\mathbb{\Gamma} = \{\Gamma\}$) and proposed a refinement a' of a different from it (i.e., $a \neq a'$, rule [BindArgRefAnns]): since we do not know whether a' is the best possible refinement for the argument, yet, then before attacking the body κ the system retries to check the expression using the new refinement a' for argument. Finally, (4) the check for the argument succeeded (i.e., $\mathbb{\Gamma} = \{\Gamma\}$) and it proposed its best refinement for the argument (i.e., $a = a'$): then the system can attack the body κ of the bind-expression, which is done in the rule [Bind].

The [Bind] rule is, by far, the most complex rule of our system and needs several auxiliary definitions. First, [Bind] uses the algorithmic system to deduce the best type s for the argument a, since this can be a strict subtype of $\bigvee(\Gamma \triangleright A)$. Then it uses this type s to refine the types in the

annotation A that are compatible with the current Γ, yielding the set $\{s \wedge u \mid u \in (\Gamma \triangleright A)\}$. The function partition is applied to this set: this splits the types in the set so that they are pairwise disjoint (actually, two types with a non-empty intersection are split in three types: their intersection and their two differences).[6] This yields a set of types $\{s_i\}_{i\in I}$ which are the summands into which we want to split the type of the argument for the union rule: indeed we have $\Gamma \vdash_{\mathcal{A}} a : s = \bigvee_{i\in I} s_i$. Therefore the next step it to check that for each $i \in I$ the body κ of the bind-expression has type t under the hypothesis that x has type s_i. This gives a set of result pairs $\{(\kappa', \mathbb{\Gamma}_i)\}_{i\in I}$. We must extract from this set the appropriate information to elaborate the result for the whole bind-expression.

Using the various κ_i's is easy: all these are copies of κ with refined annotations, and we merge all of them simply by unioning the corresponding annotations: this is what the merge function occurring in the conclusion of the rule does. Using the various $\mathbb{\Gamma}_i$'s requires more work, since the type environments in $\mathbb{\Gamma}_i$ contain hypotheses about the variable x defined in the examined bind-expression. In particular, if the type for x has been refined, then we have to reflect this refinement to the free variable of a, since a is bound to x. Consider a Γ' in $\mathbb{\Gamma}_i$ for some i. If the type of x has been refined in Γ', that is, if $\Gamma'(x) \neq s_i$, then we have to refine in Γ' also the free variables of a. For instance imagine that a is (x_1, x_2), $s_i = \mathbb{1}\times\mathbb{1}$, and $\Gamma'(x) = (\mathsf{Int}\times\mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{String})$. Since $\Gamma'(x)$ is strictly smaller than s_i, then we have to refine the types of the variables in a by proposing two refinements for Γ', namely, $\Gamma'[x_1 :\hat{=} \mathsf{Int}][x_2 :\hat{=} \mathsf{Int}]$ and $\Gamma'[x_1 :\hat{=} \mathsf{Bool}][x_2 :\hat{=} \mathsf{String}]$. This is what $\mathtt{propagate}_{x,a,s_i}$ does: it propagates to the types of the free variables of a any refinement of s_i specified in the typing of x. This yields a new set $\mathbb{\Gamma}'_i$ whose environments refine those in $\mathbb{\Gamma}_i$. Once we obtained this new $\mathbb{\Gamma}'_i$ we can now extract the hypotheses about x to create a new annotation A_i for the binding and pass the rest of the environment as a refinement for the whole bind expression. This is done by the function $\mathtt{extract}_x$ defined as follows: $\mathtt{extract}_x(\mathbb{\Gamma}) \stackrel{\text{def}}{=} (\{(\Gamma\setminus x) \triangleright \Gamma(x) \mid \Gamma \in \mathbb{\Gamma}\}, \{\Gamma\setminus x \mid \Gamma \in \mathbb{\Gamma}\})$ where $\Gamma\setminus x \stackrel{\text{def}}{=} \Gamma \setminus \{x \mapsto \Gamma(x)\}$ for $x \in \mathsf{dom}(\Gamma)$. Finally, the result of the rule is formed by a pair obtained by unioning all the annotations A_i and all the refinements $\mathbb{\Gamma}''_i$ obtained for each $i \in I$.

We conclude the presentation of our system by explaining a simplified version of the main rule for checking λ-abstractions which is applied when the type t to check is an intersection of arrows:

$$[\textsc{Abs}]\ \frac{\begin{array}{c}\{s_i\}_{i\in I} = \mathtt{partition}((\Gamma\triangleright A) \cup \{s_j \mid j \in J\}) \\ (\forall i \in I) \quad \Gamma, (x : s_i) \vdash_{\mathcal{R}} \kappa : t \circ s_i \Rightarrow (\kappa_i, \mathbb{\Gamma}_i) \quad (A_i, \mathbb{\Gamma}'_i) = \mathtt{extract}_x(\mathbb{\Gamma}_i)\end{array}}{\Gamma \vdash_{\mathcal{R}} \lambda x{:}A.\kappa : t \Rightarrow (\lambda x{:}\bigcup_{i\in I} A_i.\, \mathtt{merge}(\{\kappa_i\}_{i\in I}), \bigcup_{i\in I} \mathbb{\Gamma}'_i)}\ t \simeq \bigwedge_{j\in J}(s_j \to t_j)$$

According to [→I-Alg], we must find how to split the domain of the function into some domain types, that we assign to the parameter of the function to check its body. This will yield the intersection type of the function. To determine this set of domains, we take those of the type t we are checking (i.e., $\{s_j \mid j \in J\}$, since $t \simeq \bigwedge_{j\in J}(s_j \to t_j)$) and we add them to those we already know, which are specified in the annotation A of the parameter (i.e., $(\Gamma\triangleright A)$). Then, as for [Bind], we partition this set yielding the set of domains $\{s_i\}_{i\in I}$. As customary for each i we check under the hypothesis $x : s_i$ that the body has the expected type, that is, the type of the function t applied to the type of the parameter s_i, namely, $t \circ s_i$. This yields a set of result pairs $\{(\kappa_i, \mathbb{\Gamma}_i)\}_{i\in I}$ that we use in the same way as we did in [Bind] to form the final result. The only difference is that we do not need any further refinements for the $\mathbb{\Gamma}_i$'s, since, contrary to [Bind], there is no argument whose variables need to be refined.

All the remaining rules (pairs, variables, and the rules for the special cases of unions and empty types) are mostly straightforward and can be found in Appendix B with a detailed explanation and the formal definition of all the auxiliary functions used therein. All that remains to do is to define

[6] Formally, $\mathtt{partition}(\{t_i\}_{i\in I})$ is the smallest (in term of cardinality) non-empty set of types $\{s_j\}_{j\in J}$ such that (i) $\bigvee_{j\in J} s_j \simeq \bigvee_{i\in I} t_i$, (ii) $\forall j \in J.\ \forall j' \in J.\ j \neq j' \Rightarrow s_j \wedge s'_j \simeq \mathbb{0}$, and (iii) $\forall j \in J.\ \forall i \in I.\ s_j \leq t_i$ or $s_j \wedge t_i \simeq \mathbb{0}$

$a : ((\mathsf{Int}\to(\mathsf{Int}\vee\mathsf{Bool}))\vee(\mathsf{Int}\times(\mathsf{Int}\vee\mathsf{Bool})))$
$n : \mathsf{Int}$

Fig. 7. Types of the atoms a and n.

```
1  (a ∈ (Int×Int))  ?  π1a == π2a
2    :(a ∈ (𝟙×𝟙))   ?  π2a
3    :((a  n) ∈ Int) ? (a  n) < 42
4    :(a  n)
```

Fig. 8. Expression $e_\circ$ of the source language.

```
0  bind x0 : A0 = a in
1  bind x1 : A1 = n in
2  bind x2 : A2 = π1x0 in
3  bind x3 : A3 = π2x0 in
4  bind x4 : A4 = x2 == x3 in
5  bind x5 : A5 = x0  x1 in
6  bind x6 : A6 = x5 < 42 in
7  bind x7 : A7 = (x5 ∈ Int) ? x6 : x5 in
8  bind x8 : A8 = (x0 ∈ 𝟙×𝟙) ? x3 : x7 in
9  bind x9 : A9 = (x0 ∈ Int×Int) ? x4 : x8 in x9
```

Fig. 9. MSC-form of the expression $e_\circ$.

the result of the inference algorithm as the fixpoint of the transformation defined by that system. Formally, let κ be a closed MSC-form, we define the annotation reconstruction algorithm $\mathcal{R}$ as:

$$\mathcal{R}(\kappa) = \begin{cases} \kappa & \text{if } \varnothing \vdash_{\mathcal{R}} \kappa : \mathbb{1} \Rightarrow (\kappa, \{\varnothing\}) \\ \mathcal{R}(\kappa') & \text{if } \varnothing \vdash_{\mathcal{R}} \kappa : \mathbb{1} \Rightarrow (\kappa', \{\varnothing\}) \text{ and } \kappa \neq \kappa' \\ \mathsf{Fail} & \text{if } \varnothing \vdash_{\mathcal{R}} \kappa : \mathbb{1} \Rightarrow (\kappa', \{\}) \end{cases}$$

The reconstruction algorithm is sound:

Theorem 5.1 (Soundness). *If κ is a closed MSC-form and $\mathcal{R}(\kappa) = \kappa'$, then $\varnothing \vdash_{\mathcal{A}} \kappa' : t$ for some t.*

Notice that if the algorithm does not fail, then $\lceil\langle\mathcal{R}(\kappa)\rangle\rceil = \lceil\langle\kappa\rangle\rceil$. This, together with Corollary 4.4, yields a sound procedure to type an expression e of the source language. Let κ be the algorithmic expression obtained by adding the annotation $\{\varnothing\triangleright\mathbb{1}\}$ everywhere in $\mathsf{MSC}(e)$. If $\mathcal{R}(\kappa)$ does not fail, then $\varnothing \vdash_{\mathcal{A}} \mathcal{R}(\kappa) : t$. Since $\lceil\langle\mathcal{R}(\kappa)\rangle\rceil = \lceil\langle\kappa\rangle\rceil = \lceil\mathsf{MSC}(e)\rceil \equiv_\alpha e$, then by the soundness part of Corollary 4.4 we can conclude $\varnothing \vdash e : t$. Notice also that the algorithm works for initial annotations different from $\mathbb{1}$, too. In particular, soundness holds also for intermediate terms whose λ-abstractions are explicitly annotated as in $\lambda x{:}A.\boldsymbol{e}$: if it succeeds, the algorithm will refine the term so that its domain is a subtype of $\bigvee(\Gamma\triangleright A)$. This means that we have for free a typing algorithm for the source language (5) of Section 2.2 extended with explicitly annotated functions of the form $\lambda x{:}A.e$. This is why in our prototype, presented in next section, function parameters may be optionally annotated.[7] Finally, we conjecture that the algorithm terminates, that is, that $\mathcal{R}(\kappa)$ is defined for every closed MSC-form κ, but this result is difficult to prove because the rule [AppR] creates new arrow types.

Example. We illustrate on a complete example the behaviour of our inference algorithm. We type the source language expression $e_\circ$ given in Figure 8 in which n and a are atomic expressions (whose definitions we omit) whose types are given in Figure 7. The MSC-form of $e_\circ$ is given in Figure 9. Notice that the various occurrences of $\pi_1 a$ and $a\,n$ are shared, using x_2 and x_5 respectively. We describe the iterations of Algorithm $\mathcal{R}$ which deduces for $e_\circ$ the type Bool. In what follows, we call t_a the original type of a given in Figure 8. We start with $A_0 = t_a$, $A_1 = \mathsf{Int}$, and $A_i = \mathbb{1}$ for $i = 2..9$.

First iteration.

→ `bind` $x_0 \ldots$, $\Gamma = \varnothing$, $A_0 = \{t_a\}$: Rule [Bind] on the only type in the annotation A_0 (we assume nothing is learned while typing a);

→ `bind` $x_1 \ldots$, $\Gamma = \{x_0 : t_a\}$, $A_1 = \{\mathsf{Int}\}$: Rule [Bind] on the only type in the annotation A_1;

→ `bind` $x_2 \ldots$, $\Gamma = \{x_0 : t_a\,,\ x_1 : \mathsf{Int}\}$, $A_2 = \{\mathbb{1}\}$: Rule [BindArgRefEnv] triggers recursively rule [Proj1] to type $\pi_1\, x_0$. It returns the singleton set of environments:

$$\mathbb{\Gamma} = \{\ \{x_0 : (\mathsf{Int}\times(\mathsf{Int}\vee\mathsf{Bool}))\,,\ x_1 : \mathsf{Int}\}\ \}.$$

[7] In the implementation we forbid the annotations written by the user to be refined, so that the domain of $\lambda x{:}A.e$ will be exactly $\bigvee(\Gamma\triangleright A)$. In this way the system deduces for $\lambda x.(x+1)$ the type $\mathsf{Int}\to\mathsf{Int}$ but rejects $\lambda x{:}\mathbb{1}.(x+1)$ as ill-typed.

← bind $x_2 \ldots$: Rule [BindArgRefEnv] returns the following set containing two environments
$$\mathbb{\Gamma}_0 = \{\ \{x_0 : (\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool})), x_1 : \mathsf{Int}\}\ ,\ \{x_0 : t_a, x_1 : \mathsf{Int}\}\ \}.$$
Notice that the rest of the program is ignored and $\mathbb{\Gamma}_0$ is propagated upward.

← bind $x_1 \ldots$: Rule [Bind] returns. Here since nothing new is learned about the type of x_1 (functions `propagate` and `extract` behave as identities), $\mathbb{\Gamma}_0$ is propagated upward.

← bind $x_0 \ldots$: Rule [Bind] returns. Here functions `propagate` and `extract` create the new annotation for x_0 that is $A_0' = \{t_a, (\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool}))\}$

At this point a new annotation has been inferred (i.e., A_0'), so Algorithm $\mathcal{R}$ restarts.

Second iteration.

→ bind $x_0 \ldots$, $\Gamma = \varnothing$, $A_0 = \{(\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool}))\ ,\ t_a\}$: Rule [Bind] on the two types in A_0. Here `partition` splits the annotation into $\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool})$ and $t_a \setminus (\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool})) \simeq \mathsf{Int} \to (\mathsf{Int} \vee \mathsf{Bool})$. The rule tries both types for x_0 in turn. We focus first on $\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool})$.

→ bind $x_1 \ldots$, $\Gamma = \{x_0 : \mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool})\}, A_1 = \{\mathsf{Int}\}$: Rule [Bind] on the only type of the annotation A_1; (the following cases from bind x_2 to bind x_4 are similar and omitted);

→ bind x_5, $\Gamma = \{x_4 : \mathsf{Bool}, \ldots\}$: Rule [BindArgUntyp] since x_0 does not have a function type;

→ bind x_6 and bind x_7, $\Gamma = \{x_4 : \mathsf{Bool}, \ldots\}$: Rule [BindArgUntyp] since $x_5 \notin \mathrm{dom}(\Gamma)$

→ bind $x_8 \ldots$, $\Gamma = \{x_4 : \mathsf{Bool}, \ldots\}$: Rule [Bind] recursively calls [Case]. Here since, the type of x_0 is completely contained in $\mathbb{1} \times \mathbb{1}$, the case rule only returns the original Γ (cf. the definition of $[\_ :\hat{=} \_]$): no new information is learned; the rest of the MSC-form is examined.

→ bind $x_9 \ldots$, $\Gamma = \{x_8 : (\mathsf{Int} \vee \mathsf{Bool}), \ldots\}$: Rule [BindArgRefEnv] recursively calls [Case] rule. Here, however, the type of x_0 is not a subtype of the tested type, this rule therefore returns a singleton with a refined environment $\mathbb{\Gamma} = \{\Gamma\{[x_0 :\hat{=} \mathsf{Int} \times \mathsf{Int}]\}$

← bind $x_9 \ldots$: Rule [BindArgRefEnv] returns the original environment and a refined one for x_0. The typing does not continue further and returns upward.

← bind $x_0 \ldots$ like in the previous iteration the annotation for x_0 comes back to its binder, yielding a new annotation : $A_0'' = \{t_a, (\mathsf{Int} \times (\mathsf{Int} \vee \mathsf{Bool}))), \mathsf{Int} \times \mathsf{Int}\}$

At the third iteration, `partition()` used in [Bind] on the "bind x_0" expression splits A_0'' into three types : Int×Int, Int×Bool, and Int→(Int∨Bool). For both product types, the typing succeeds till the end (each time skipping the lines 5–7 where x_0 is used as a function). As for the functional annotation, it is almost straightforward. One caveat is in the typing of x_6: since $< : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool}$, the application $x_5\ 42$ introduces a new refinement for x_5 which, at this point, has type Int∨Bool (the return type of x_0). The algorithm propagates both types to A_5 and the rest of the program is typed twice, under both hypotheses. Notice that both succeed since, in the case where x_5 has type Bool, x_6 only occurs in an unreachable branch of a type case. The final MSC-form has annotations A_0={Int×Int, Int×Bool, Int→(Int∨Bool)}, A_1=A_2=Int, A_3={x_0:Int×Int▷Int, x_0:Int×Bool▷Bool}, A_5={x_0:Int→(Int∨Bool)▷Int, x_0:Int→(Int∨Bool)▷Bool}, Bool guarded by the active environments for the others. For this term $\vdash_{\mathcal{R}}$ infers the type Bool.

6 IMPLEMENTATION

We have implemented Algorithm $\mathcal{R}$ in OCaml, using CDuce [CDuce] as a library to provide set-theoretic types and semantic subtyping. The prototype amounts to 3500 lines of OCaml code and features several extensions such as let bindings and records (both formalised in Appendix C) and annotations of function parameters (see Footnote 7). The transformation of terms in their MSC-form is similar to the *locally nameless* approach ([Charguéraud 2012]). Expressions from the source language are transformed so that bound variables are represented using De Bruijn indices, while free variables are represented with symbolic names. While performing this transformation, hash-consing is used to identify structurally equal subterms. We give in Table 1 the code of

Table 1. Types inferred by the implementation (times are in ms)

	Code	Inferred type	MSC	Inf.
1	`let is_int = fun x -> if x is Int then true else false`	(Int → True) ∧ (¬Int → False)	0.02	0.22
	`let is_bool = fun x -> if x is Bool then true else false`	(Bool → True) ∧ (¬Bool → False)	0.02	0.21
2	`type Falsy = False | "" | 0` `type Truthy = ~Falsy`			
	`let not_ = fun x -> if x is Truthy then false else true`	(Truthy → False) ∧ (Falsy → True)	0.03	0.58
	`let to_Bool = fun x -> not_ (not_ x)`	(Truthy → True) ∧ (Falsy → False)	0.02	0.93
	`let and_ = fun x -> fun y -> if x is Truthy then to_Bool y else false`	(Falsy→𝟙→False) ∧ (Truthy→Truthy→True)∧ (Truthy→Falsy→False)	0.03	2.24
	`let or_ = fun x -> fun y -> not_ (and_ (not_ x) (not_ y))`	(Truthy→𝟙→True) ∧ (Falsy→Truthy→True)∧ (Falsy → Falsy → False)	0.03	3.34

3	`strlen : String -> Int`
	`let example14 =`
	`fun input -> fun extra ->`
	`if and_ (is_int input) (is_int(fst extra)) is True`
	`then input + (fst extra)`
	`else if is_int (fst extra) is True then (strlen input) + (fst extra)`
	`else 0`
		((Int ∨ String) → (¬Int×𝟙) → 0)∧ ((Int ∨ String) → (Int×𝟙) → Int)∧ (¬(Int ∨ String) → (¬Int×𝟙) → 0)	0.05	3.61

4	`let example6_wrong =`
	`fun (x : Int|String) -> fun (y : Any) ->`
	`if and_ (is_int x)(is_string y) is True then add x (strlen y) else strlen x`
		Ill typed	0.07	1.52

	`let example6_ok =`
	`fun x -> fun y ->`
	`if and_ (is_int x)(is_string y) is True then add x (strlen y) else strlen x`
		(String → 𝟙 → Int)∧ (Int → String → Int)	0.07	3.51

5	`let detailed_ex =`
	`fun (a : (Int -> (Int|Bool)) |(Int, (Int|Bool))) ->`
	`fun (n : Int) ->`
	`if a is (Int,Int) then (fst a)=(snd a)`
	`else if a is (Any,Any) then snd a`
	`else if (a n) is Int then (a n) < 42`
	`else a n`
		(Int → (Int ∨ Bool)) ∨ (Int×(Int ∨ Bool)) → Int → Bool	0.08	1.77

several functions, using a syntax similar to OCaml, where uppercase identifiers (e.g., True, String) denote types and lowercase identifiers denote variables or constants. For each function we report its inferred type, the time taken to put its body in MSC form, and the time taken to infer its type or typecheck it (for annotated functions). All runtimes are given in milliseconds, averaged over ten runs. The experiments were done on an Intel Core i7-8565U 1.8GHz CPU (with 16GB of RAM). The code was compiled natively using OCaml 4.12.0. All these examples (and more) can be tested online with the interactive prototype hosted at https://typecaseunion.github.io [Castagna et al. 2022].

Code 1 and 2 show that exact overloaded types can be inferred even in the absence of annotations. In Code 1 we encode type predicates as they can be found in Typed Racket. The inferred overloaded types exactly specify the semantics of these functions using the singleton types of the values `true` and `false`, while in Typed Racket this requires these predicates to be primitives of the language and are typed with specific rules. Code 2 implements Boolean operators by considering values as in JavaScript where eight specific "falsy" values (`false`, `""`, `0`, `-0`, `0n`, `undefined`, `null`, and `NaN`) are considered to be equivalent to false, and all the others—called "truthy" values—to be equivalent to true. We first define the type `Falsy` as the union of the singleton types of `false`, `""`, and `0` (the other values are absent in our prototype) and the type `Truthy` as the negation of `Falsy`. We said that our system decides how to split the types of variables in bindings by using the type-cases and the applications of overloaded functions that occur in the program. The function `not_` is an

example in which the decision is based on the type-cases: the exact inferred intersection type is obtained by splitting the type of `x` because `x` is tested in a type-case. The function `to_Bool` is an example in which the decision is based on an overloaded application: the type is inferred by splitting the type of `x` since `x` is the argument of the function `not_`. In `to_Bool` by a double application of `not_` we map truthy values into `true` and falsy ones into `false`, as the type inferred for `to_Bool` exactly specifies. The function `and_` mixes the two kinds of decision since it tests the type of the first argument and applies an overloaded function to the second argument. We could have defined `and_` as two nested tests checking whether both arguments are truthy and returning false otherwise: the system would have inferred the same type which, once more, exactly specifies the semantics of the function. If we wanted to implement for the "and" operator the same semantics as the one defined for the logical AND (`&&`) of JavaScript, then we should instead have used the following definition `fun x -> fun y -> if x is Falsy then x else y` whose inferred type `(Falsy→𝟙→Falsy)∧(Truthy→𝟙→𝟙)` does not specify the function's exact semantics because our system lacks polymorphism (with polymorphic types we would expect the inferred type to be $\forall\alpha.(\alpha\wedge\texttt{Falsy}\to\mathbb{1}\to\alpha\wedge\texttt{Falsy})\wedge(\texttt{Truthy}\to\alpha\to\alpha)$—where α is a type variable—which states that if the first argument is of type (subtype of) `Falsy`, then the result will be of the same type as the type of first argument—independently from the second one—, while if the first argument is of type `Truthy`, then the result will be of the same type as the type of second argument). Finally, the last example in Code 2 defines `or_` by combining the two previous functions according to De Morgan's laws: again the inferred type is exact. The degree of precision achieved by the type inference for the examples in Code 2 is out of reach of all existing approaches to occurrence typing.

Our implementation can type all the 14 paradigmatic examples listed by Tobin-Hochstadt and Felleisen [2010] (THF) whose results we improve in several ways: first, we infer types that are more precise than those inferred in THF; second, our system types all examples without needing any annotation, whereas THF must specify some annotations in 5 of the 14 examples; third, our analysis works also when in these example we employ user-defined connectives and type predicates, whereas in THF these must be hard-coded to be used inside a test. For space reasons, we did not detail all these examples here (but they can be tested in our online prototype by selecting the appropriate menu entry) and chose instead to show only two of them in Code 3 and 4.

Code 3 is Example 14 of THF, which is last and most complete of the 14 examples of THF and summarizes the features of all the others. This definition shows all the improvements brought by our system and listed above: first, we infer a more precise type which discriminates the cases in which the function returns a generic integer or `0` (i.e., `0` is returned whenever the second argument is of type $(\neg\mathsf{Int}\times\mathbb{1})$, independently from the first argument's type); second, our inference does not need any annotation, while in THF both parameters of the function must be explicitly annotated; third, the tested expression is a Boolean expression obtained by applying custom user-defined connectives (`and_`) and type predicates (`is_int`) whose use would make the analysis of THF fail.

Code 4 is Example 6 of THF which shows error detection: if `x` is assumed of type Int∨String (as in `example6_wrong`), then the function is ill-typed, and rightly so since if both arguments are not strings, then `strlen x` is selected and its execution fails. However, if as in `example6_ok`, we let our system determine the types of the parameters, then it rightly determines that the first argument must be either a string or an integer (any other type would select `strlen x` and then fail), but also that when the first is an integer, then the second must be a string, or the function would fail. Such a deduction is out of reach of the approach defined by THF.

Code 5 is the detailed example of the previous section and shows that the type of function parameters can easily be constrained if one wishes. Interestingly, if we remove the `Int` annotation from the second parameter `n`, the system computes a more precise type $((\mathsf{Int}\to(\mathsf{Bool}\vee\mathsf{Int}))\to$

$\mathsf{Int} \to \mathsf{Bool}) \wedge ((\mathsf{Int} \times (\mathsf{Bool} \vee \mathsf{Int})) \to \mathbb{1} \to \mathsf{Bool})$ which clearly states that the second argument of the function needs to be an integer only when the first one is a function.

7 RELATED WORK

As previously said, the present paper extends the system of [Barbanera et al. 1995] with three rules for type-cases and with the use of a particular subtyping relation. We then define, in several steps, a type inference algorithm for this calculus. The resulting calculus and type-inference appears to be particularly well suited for typing programs written in dynamic languages such as JavaScript.

While taking a radically different approach, we achieve a goal similar to occurrence typing, introduced in [Tobin-Hochstadt and Felleisen 2008] and further advanced in [Tobin-Hochstadt and Felleisen 2010], in the context of the Typed Racket language. In this and subsequent work, types are annotated by two logical propositions that record the type of the input depending on the (Boolean) value of the output. For instance, the type of the `number?` function states that when the output is `true`, then the argument has type `Number`, and when the output is `false`, the argument does not. These propositions are propagated and used in particular in type-cases to refine the type of variables and, more generally, expressions in the "then" and "else" branches of a conditional. Furthermore, this analysis focuses on a particular set of pure operations, so that the approach works also in the presence of side-effects. Contrary to these works, we try not to depend on an external logic but, rather, to express as much as possible these conditions with set-theoretic types. For instance, we track the dependency between input and output types of functions using intersection types (cf. Code 1 in Table 1), while type-case expressions are typed using intersection and negation types to refine the typing environments of the branches. Our approach is more global since, not only our analysis strives to infer type information by analyzing all types of results (and not just `true` or `false`), but also tries to perform this analysis for all possible expressions (and not just for a restricted set of expressions). This allows our system to type all the examples given in [Tobin-Hochstadt and Felleisen 2010] (and contrary to the cited work, without needing any annotations) and many more but, as we explain at the end of this section, at the expense of an immediate compatibility with the presence of side-effects.

In a previous work [Castagna et al. 2021] we already used characteristics of semantic subtyping to improve occurrence typing but the approach we used there was completely different from the one presented here. Instead of relying on bindings to track the different occurrences of a same expression, we enriched type environments so that they mapped occurrences of expressions (expressed in terms of paths) to types. This yielded a type-theoretic approach with non standard features (the type environments) that, contrary to the present one, could not capture the flow of information between variables and thus failed to type Code 3 of Table 1. Furthermore, the connection with the union elimination rule was completely missing.

Set-theoretic types have also been used by [Kent 2019, Chapter 5] to extend the logical techniques developed for Typed Racket to track under which hypotheses an expression returns false or not. Kent uses set-theoretic types to express type predicates (a predicate that holds only for a type t has type $p : (t \to \mathtt{True}) \wedge (\neg t \to \mathtt{False})$) as well as to express in a more compact (and, sometimes, more precise) way the types of several built-in Typed Racket functions. It also uses the properties of set-theoretic types to deduce the logical types (i.e., the propositions that hold when an expressions produces `false` or not) of arguments of function applications. The main difference of Kent's approach with respect to ours is that, since it builds on the logical propositions approach, then it focuses the use of set-theoretic types and of the analysis of arguments of applications of a selected set of pure expressions (while we use all expressions) to determine when an expression yields a result of type `False` or $\neg$`False` (while we use all types of results). The consequence is that not only Kent's approach covers fewer cases than ours and cannot infer intersection types, but also the very

fact of focusing on truthy vs. false results may make Kent's analysis fail even for pure Boolean tests where it would be naively expected to work. The approach has however the advantage of building on Typed Racked which provides a mature and high-performing implementation. The reader will find in [Castagna et al. 2021, see section on related work] an extensive and detailed comparison between current approaches of occurrence typing and those based on set-theoretic types.

Our approach is based on program transformation: we transform expressions into MSC-forms. A similar approach is used by Rondon et al. [2008] who transform expressions into *A-normal forms* (ANF) [Sabry and Felleisen 1992] and track precise type information for every sub-expression by keeping this information for the variables. While this solution is close to ours it does not achieve the same degree of precision for the simple reason that, contrary to MSC-forms, ANFs were not designed to target occurrence typing. MSC-forms' rationale is to give a unique name to every α-equivalent sub-expression of the initial term, ANFs instead ensure that arguments of applications are immediate values. While the result looks similar, there are key differences: since the sharing of α-equivalent subterms is used only for typing, it does not need to preserve the semantics of the original term. For example, sub-expressions in the branches of a conditional are hoisted outside the conditional (crucial for occurrence-typing), which must not be done for ANFs. Conversely, all proper subterms of an application must be variables in MSC-forms but not in ANFs.

The typing algorithm we present in Section 5 works as a bi-directional typing algorithm: the definition of a variable gives a forward constraint on its type while the use of a variable (e.g., in a type-case or as part of an application) gives a backward constraint that is added to its definition. From the extensive survey by Dunfield and Krishnaswami [2019a] on bi-directional typing, we see that this technique is well-suited for the type-checking or type-inference of complex features such as, for instance, in [Pierce and Turner 2000] (local type inference in the presence of subtyping), [Pottier and Régis-Gianas 2006] (bi-directional type propagation for typing generalized algebraic data-types), or [Dunfield and Krishnaswami 2019b] (bi-directional type checking for higher rank polymorphism). The two latter works, in particular, seem to indicate that our approach remains viable when extending the present work with parametric polymorphism.

A feature we completely omitted in our study is gradual typing. Works such as [Chaudhuri et al. 2017] (for Flow) or [Rastogi et al. 2015] (for TypeScript), account for the presence of unsafe, already written code, by using a form of gradual typing. Castagna et al. [2021] outline how gradual typing can be integrated in a system with semantic subtyping and occurrence typing using work by [Castagna et al. 2017; 2019]: we think that those ideas can be adapted to the work presented here.

We end this presentation of related work with a discussion on side effects, a crucial feature for dynamic languages. Although in our system we did not take into account side-effects—and actually our system works because all the expressions of our language are pure—it is interesting to see how the different approaches of occurrence typing position themselves with respect to the problem of handling side effects, since this helps to better place our work in the taxonomy of the current literature. As Sam Tobin-Hochstadt insightfully noticed, one can distinguish the approaches that use types to reason about the dynamic behavior of programs according to the set of expressions that are taken into account by the analysis. In the case of occurrence typing, this set is often determined by the way impure expressions are handled. On the one end of the spectrum lies our approach (both this work and the one in Castagna et al. [2021]): our analysis takes into account *all* expressions but, in its current formulation, it works only for pure languages. On the other end of the spectrum we find the approach of Typed Racket whose analysis reasons about a limited and predetermined set of *pure* operations: all data structure accessors. Somewhere in-between lies the approach of the Flow language—whose core features were formalized by Chaudhuri et al. [2017]—which implements a complex effect systems to determine pure expressions. While the system presented here does not work for impure languages, we argue that its foundational nature

predisposes it to be adapted to handle impure expressions as well, by adopting existing solutions or proposing new ones. For instance, it is not hard to modify our system so that it takes into account only a set of predetermined pure expressions, as done by Typed Racket: it suffices to instruct the transformation in MSC-form to use distinct bind variables for distinct occurrences of expressions that are not in this set. However, such a solution would be marginally interesting since by excluding from the analysis all applications, we would lose most of the advantages of our approach with respect to the one with logical propositions. Thus a more interesting solution would be to use some external static analysis tools—e.g., to graft the effect system of Chaudhuri et al. [2017] on ours—to detect impure expressions. The idea would be to mark different occurrences of a same impure expression using different marks and, again, instruct the transformation in MSC-form to use distinct bind variables for expressions with distinct marks. For instance, consider the test $(f\,x \in \mathsf{Int})\,?\,\ldots : \ldots$: if fx were flagged as impure then an occurrence of fx in the "then" branch would not be supposed to be of type Int since the MSC-form of this expressions would use two distinct variables to bind fx occurring in the test and the one in the branch. Although this would certainly improve our analysis, is would still significantly limit the sharing. Which is why we believe that, ultimately, our system should not resort to external static analysis tools to detect impure expressions but, rather, it has to integrate this analysis with the typing one, so as to mark *only* those impure expressions whose side-effects may affect the semantics of some type-cases. For instance, consider a JavaScript object `obj` that we modify as follows: `obj["key"] = 3`. If the field `"key"` is already present in `obj` with type Int and we do not test it more than about this type, then it is not necessary to mark different occurrences of `obj` with different marks, since the result of the type-case will not be changed by the assignment; the same holds true if the field is absent but type-cases do not discriminate on its presence. Otherwise, some occurrences of `obj` must use different marks: the analysis will determine which ones. We leave this study for future work.

8 CONCLUSION

Although the technical development of our work may appear complex, the unfolding of the logical sequence of its steps can be easily summarized. In Section 1 we argued that the essence of occurrence typing can be captured by adding three typing rules, $[\vee]$, $[\in_1]$, and $[\in_2]$: the union elimination rule can split the type of any expressions into a union of two types that can be tested separately and the rules for type-cases distribute these tests differently on the two branches of a type-case. The addition of these three rules yields the system of Section 2 which captures the spirit of occurrence typing, covers the examples proposed by existing approaches, but is not algorithmic. To obtain an algorithmic system, four technical problems are to be solved: (*i*) how to choose on which expressions the rule $[\vee]$ must be applied; (*ii*) given an expression chosen for applying $[\vee]$, how to choose which sub-expression of this expression and which occurrences of this sub-expression should the system use to apply $[\vee]$; (*iii*) how to determine the union of types into which the system should split the type of a sub-expression chosen for $[\vee]$; (*iv*) how to determine the arrows that form the intersection type of a λ-abstraction that is not annotated. Section 3 solves the first two problems—which made the system *non syntax-directed*—by the definition of MSC-forms: the fact that MSC-forms bind atoms whose all proper sub-expressions are variables means that the system chooses to apply $[\vee]$ on *all* sub-expressions, while the maximal sharing property of MSC-forms means that the system chooses *all* occurrences of each sub-expression since it replaces all of them by the same variable. Section 4 solves the last two problems—which made the system *non analytic*—by the definition of annotations: annotations state 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 type of the argument of the `bind`-expression into a union). By this sequence of steps

we reduced the problem of typing expressions with occurrence typing to the problem of choosing annotations for MSC-forms and shown that this choice corresponds to determining how to split types into unions for `bind`-expressions and into intersection for λ-abstractions. Section 5 suggests that the choice of how to split these types can be based on a program analysis that focuses on the types tested in type-cases or involved in applications of overloaded functions. Section 6 implements these choices demonstrating the practical implications of our work.

In summary, the logical sequence described above highlights the connection between union elimination and occurrence typing techniques via the addition of negation types and their use in the typing of type-case expressions. It also provides an effective way to reduce this typing problem to the inference of some specific annotations.

From a theoretical viewpoint, our work is a step forward in the quest of an inversion lemma for the union elimination rule. Although we are still far from an inversion lemma, the results of Sections 3 and 4 show that for a well-typed term e of the source language (or of Barbanera et al. [1995]) there exists a canonical way to use the union rule to derive its type, which corresponds to the derivation encoded by $\mathsf{MSC}(e)$. Thus the problem now is no longer when to use the rule, but how to determine the split of the type of the argument of the union rule into the union of two types, which coincides with inferring the annotations for the corresponding binding-expression.

From a practical viewpoint, our work reframes the problem of occurrence typing into a classical setting that has been actively studied for thirty years and for which a wealth of results and techniques already exists. We want to transpose some of them to our specific setting, in particular the extension to polymorphism and the generation and resolution of systems of constraints, to infer the polymorphic types we hinted at in Section 6. For this we count reusing the theory of polymorphic types with semantic subtyping [Castagna and Xu 2011] together with the typing techniques and algorithms developed for CDuce, both for its explicitly typed version [Castagna et al. 2015, 2014] and the implicitly typed one [Castagna et al. 2016; Petrucciani 2019]. Eminently of practical interest is also the fact that we effectively reduced type inference to a very specific problem, namely, the reconstruction of some specific annotations. The algorithm we described in Section 5 is just one possible solution to this problem, but our formal setting opens the way to the definition of other different techniques. In particular, we are studying the feasibility of switching from the current system that at each pass generates type refinements for the variables of the term, to one that generates, instead, sets of type constraints (such as those defined by Petrucciani [2019, Chapter 4]) whose resolution would yield these (and hopefully better) refinements. This seems an obvious choice if we want to handle polymorphism and it would also improve the precision of our algorithm which currently works poorly when higher-order function parameters are not explicitly annotated. This shortcoming is expected—and shared among the approaches that lack polymorphism—since our algorithm initializes higher-order parameters with the type $\mathbb{0}\to\mathbb{1}$ while, if type variables were available, the algorithm could instead initialize them with $\alpha\to\beta$ where α and β are fresh. This would allow the system to better track and refine the types of these parameters.

ACKNOWLEDGMENTS

This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045- DIGICOSME) operated by ANR as part of the program «Investissement d'Avenir» Idex Paris-Saclay (ANR-11-IDEX-0003-02), by the « Chaire Langages Dynamiques pour les Données » of the Paris-Saclay foundation, and by a Google PhD fellowship. The authors would like to thank Delia Kesner for her help with the rewriting systems.

REFERENCES

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types. *Inf. Comput.* 119, 2 (June 1995), 202–230. https://doi.org/10.1006/inco.1995.1086

Giuseppe Castagna. 2020. 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 (2020), 15:1–15:58. https://doi.org/10.23638/LMCS-16(1:15)2020

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110285

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyễn. 2021. Revisiting Occurrence Typing. (oct 2021). arXiv:1907.05590 To appear in *Science of Computer Programming*, Elsevier.

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: a New Perspective. *Proc. ACM Program. Lang.* 3, POPL '19 46th ACM Symposium on Principles of Programming Languages, Article 16 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290329

Giuseppe Castagna, Mickaël Laurent, Kim Nguyễn, and Matthew Lutze. 2022. *Prototype for Article: On Type-Cases, Union Elimination, and Occurrence Typing*. https://doi.org/10.1145/3462306 Online interactive version available at https://typecaseunion.github.io.

Giuseppe Castagna, Kim Nguyễn, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15)*. 289–302. https://doi.org/10.1145/2676726.2676991

Giuseppe Castagna, Kim Nguyễn, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. 2014. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '14)*. 5–17. https://doi.org/10.1145/2676726.2676991

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. 2016. Set-Theoretic Types for Polymorphic Variants. In *ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming*. 378–391. https://doi.org/10.1145/2951913.2951928

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*. 94–106. https://doi.org/10.1145/2034773.2034788

CDuce. *The CDuce Compiler*. CDuce https://www.cduce.org

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. https://doi.org/10.1007/s10817-011-9225-2

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 48 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133872

Mariangiola Dezani-Ciancaglini. 2020. Personal communication.

Mariangiola Dezani-Ciancaglini, Alain Frisch, Elio Giovannetti, and Yoko Motohama. 2003. The Relevance of Semantic Subtyping. *Electronic Notes in Theoretical Computer Science* 70, 1 (2003), 88 – 105. https://doi.org/10.1016/S1571-0661(04)80492-4 ITRS '02, Intersection Types and Related Systems (FLoC Satellite Event).

Jana Dunfield and Neel Krishnaswami. 2019a. Bidirectional Typing. *CoRR* abs/1908.05839 (2019). arXiv:1908.05839

Jana Dunfield and Neelakantan R. Krishnaswami. 2019b. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290322

Facebook. *Flow*. Facebook https://flow.org/

Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. Ph. D. Dissertation. Université Paris Diderot. http://www.cduce.org/papers/frisch_phd.pdf

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 137–146. https://doi.org/10.1109/LICS.2002.1029823

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM* 55, 4 (Sept. 2008), 19:1–19:64. http://doi.acm.org/10.1145/1391289.1391293

J. Roger Hindley and Jonathan P. Seldin. 2008. *Lambda-Calculus and Combinators An Introduction*. Cambridge University Press.

Andrew M. Kent. 2019. *Advanced Logical Type Systems for Untyped Languages*. Ph. D. Dissertation. Indiana University. https://pnwamk.github.io/docs/dissertation.pdf

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1986. An ideal model for recursive polymorphic types. *Information and Control* 71, 1 (1986), 95–130. https://doi.org/10.1016/S0019-9958(86)80019-5

Per Martin-Löf. 1994. *Analytic and Synthetic Judgements in Type Theory*. Springer Netherlands, Dordrecht, 87–99. https://doi.org/10.1007/978-94-011-0834-8_5

Microsoft. *TypeScript*. Microsoft https://www.typescriptlang.org/

Tommaso Petrucciani. 2019. *Polymorphic Set-Theoretic Types for Functional Languages*. Ph. D. Dissertation. Joint Ph.D. Thesis, Università di Genova and Université Paris Diderot. https://tel.archives-ouvertes.fr/tel-02119930

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

François Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 232–244. https://doi.org/10.1145/1111037.1111058

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe and Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style.. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, 288–298. https://doi.org/10.1145/141471.141563

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1863543.1863561

Types 2019. What exactly should we call syntax-directed inference rules? Discussion on the Types mailing list. http://lists.seas.upenn.edu/pipermail/types-list/2019/002138.html.

Wikipedia. 2021. Peter Parker principle. https://en.wikipedia.org/wiki/With_great_power_comes_great_responsibility [Online; accessed 22-October-2021].

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093