Revisiting Occurrence Typing

Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, Kim Nguyen

Revisiting Occurrence Typing

GIUSEPPE CASTAGNA, CNRS - UniversitÃĭ de Paris, France
VICTOR LANVIN, UniversitÃĭ de Paris, France
MICKAÃŃL LAURENT, ÃĽcole Normale SupÃĭrieure Paris-Saclay, France
KIM NGUYEN, UniversitÃĭ Paris Sud, France

We revisit occurrence typing, a technique to refine the type of variables occurring in type-cases and, thus, capture some programming patterns used in untyped languages. Although occurrence typing was tied from its inception to set-theoretic types—union types, in particular—it never fully exploited the capabilities of these types. Here we show how, by using set-theoretic types, it is possible to develop a general typing framemork that encompasses and generalizes several aspects of current occurrence typing proposals and that can be applied to tackle other problems such as the inference of intersection types for functions and the optimization of the compilation of gradually typed languages.

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

1 INTRODUCTION

Typescript and Flow 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) {
  (typeof(x) === "number")? x++ : 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++`), 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 string method of the ECMAScript 5 standard that trims whitespaces from the beginning and end of the string) and both the type-checker of TypeScript and the one of Flow rightly accept this function. This is possible because both type-checkers 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 test to specialize—precisely to *refine*—the type 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.

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

Authors' addresses: Giuseppe Castagna, Institut de Recherche en Informatique Fondamentale (IRIF), CNRS - UniversitÃĭ de Paris, France; Victor Lanvin, Institut de Recherche en Informatique Fondamentale (IRIF), UniversitÃĭ de Paris, France; MickaÃńl Laurent, ÃĽcole Normale SupÃĭrieure Paris-Saclay, France; Kim Nguyen, Laboratoire de Recherche en Informatique (LRI), UniversitÃĭ Paris Sud, France.

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. From its inception, occurrence typing was intimately tied to type systems with set-theoretic types: unions, intersections, and negation of types. Union was the first type connective to appear, since it was already used by Tobin-Hochstadt and Felleisen [2008] where its presence was needed 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, more precise types to functions such as `foo`. For instance, since `++` returns an integer and `trim()` a string, then our function `foo` has type `(number|string)→(number|string)`.[2] 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 `number|string`. This is exactly what the intersection type `(number→number) & (string→string)` states (intuitively, an expression has an intersection of type, 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 => {
  (typeof(x) === "number")? x++ : x.trim()                    (2)
}
```

For what concerns negation types, they are pervasive in the occurrence typing approach, even though they are used only at meta-theoretic level,[3] 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`.

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. In this work we aim at removing this limitation on the contexts and develop a general theory to refine the type of variables that occur in tested expressions under generic contexts, such as variables occurring in the left or the right expressions of an application. In other words, we aim at establishing a formal framework to extract as much static information as possible from a type test. We leverage our analysis on the presence of full-fledged set-theoretic types connectives provided by the theory of semantic subtyping. Our analysis will also yield three byproducts. First, to refine the type of the variables we have to refine the type of the expressions they occur in and we can use this information to improve our analysis. Therefore our occurrence typing approach will refine not only the types of variables but also the types of generic expressions (bypassing usual type inference). Second, the result of our analysis can be used to infer intersection types for functions, even in the absence of precise type annotations such as the one in the definition of `foo` in (2). Third, we show how to combine occurrence typing with gradual typing, and in particular how the former can be used to optimize the compilation of the latter.

We focus our study on conditionals that test types and consider the following syntax: $(e \in t)?e:e$ (e.g., in this syntax the body of `foo` is rendered as $(x \in \mathsf{Int})?x+1:(\mathsf{trim}\ x)$). In particular,

[2] Actually, both Flow and TypeScript deduce as return type `number|string|void`, since they track when operations may yield `void` results. Considering this would be easy but also clutter our presentation, which is why we omit such details.
[3] At the moment of writing there is a pending pull request to add negation types to the syntax of TypeScript, but that is all.

in this introduction we concentrate on applications, since they constitute the most difficult case and many other cases can be reduced to them. A typical example is the expression

$$(x_1x_2 \in t)\,?e_1 : e_2 \tag{3}$$

where x_i's denote variables, t is some type, and e_i's are generic expressions. Depending on the actual t and on the static types of x_1 and x_2, we can make type assumptions for x_1, for x_2, *and* for the application x_1x_2 when typing e_1 that are different from those we can make when typing e_2. For instance, suppose x_1 is bound to the function `foo` defined in (2). Thus x_1 has type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{String} \to \mathsf{String})$ (we used the syntax of the types of Section 2 where unions and intersections are denoted by $\vee$ and $\wedge$ and have priority over $\to$ and $\times$, but not over $\neg$). Then it is not hard to see that the expression[4]

$$\texttt{let } x_1 = \texttt{foo in } (x_1x_2 \in \mathsf{Int})\,?((x_1x_2) + x_2) : 42 \tag{4}$$

is well typed with type Int: when typing the branch "then" we know that the test $x_1x_2 \in \mathsf{Int}$ suceeded and that, therefore, not only x_1x_2 is of type Int, but also that x_2 is of type Int: the other possibility, $x_2 : \mathsf{String}$, would have made the test fail. For (4) we reasoned only on the type of the variables in the "then" branch but we can do the same on the "else" branch as shown by the following expression, where `@` denotes string concatenation

$$(x_1x_2 \in \mathsf{Int})\,?((x_1x_2) + x_2) : ((x_1x_2)\,@\,x_2) \tag{5}$$

If the static type of x_1 is $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{String} \to \mathsf{String})$ then x_1x_2 is well typed only if the static type of x_2 is (a subtype of) $\mathsf{Int} \vee \mathsf{String}$ and from that it is not hard to deduce that (5) has type $\mathsf{Int} \vee \mathsf{String}$. Let us see this in detail. The expression in (5) is typed in the following type environment: $x_1 : (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{String} \to \mathsf{String})$, $x_2 : \mathsf{Int} \vee \mathsf{String}$. All we can deduce is then that the application x_1x_2 has type $\mathsf{Int} \vee \mathsf{String}$ which is not enough to type either the "then" branch or the "else" branch. In order to type the "then" branch $(x_1x_2) + x_2$ we must be able to deduce that both x_1x_2 and x_2 are of type Int. Since we are in the "then" branch, then we know that the type test succeeded and that, therefore, x_1x_2 has type Int. Thus we can assume in typing this branch that x_1x_2 has both its static type and type Int and, thus, their intersection: $(\mathsf{Int} \vee \mathsf{String}) \wedge \mathsf{Int}$, that is Int. For what concerns x_2 we use the static type of x_1, that is $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{String} \to \mathsf{String})$, and notice that this function returns an Int only if its argument is of type Int. Reasoning as above we thus deduce that in the "then" branch the type of x_2 is the intersection of its static type with Int: $(\mathsf{Int} \vee \mathsf{String}) \wedge \mathsf{Int}$ that is Int. To type the "else" branch we reason exactly in the same way, with the only difference that, since the type test has failed, then we know that the type of the tested expression is *not* Int. That is, the expression x_1x_2 can produce any possible value barring an Int. If we denote by $\mathbb{1}$ the type of all values and by $\setminus$ the set difference, then this means that in the else branch we know that x_1x_2 has type $\mathbb{1}\setminus\mathsf{Int}$—written $\neg\mathsf{Int}$—, that is, it can return values of any type barred Int. Reasoning as for the "then" branch we then assume that x_1x_2 has type $(\mathsf{Int} \vee \mathsf{String}) \wedge \neg\mathsf{Int}$ (i.e., $(\mathsf{Int} \vee \mathsf{String}) \setminus \mathsf{Int}$, that is, String), that x_2 must be of type String for the application to have type $\neg\mathsf{Int}$ and therefore we assume that x_2 has type $(\mathsf{Int} \vee \mathsf{String}) \wedge \mathsf{String}$ (i.e., again String).

We have seen that we can specialize in both branches the type of the whole expression x_1x_2, the type of the argument x_2, but what about the type of x_1? Well, this depends on the type of x_1 itself. In particular, if instead of an intersection type x_1 is typed by a union type, then the test may give us information about the type of the function in the various branches. So for instance if in

[4]This and most of the following expressions are just given for the sake of example. Determining the type *in each branch* of expressions other than variables is interesting for constructors but less so for destructors such as applications, projections, and selections: any reasonable programmer would not repeat the same application twice, (s)he would store its result in a variable. This becomes meaningful with constructor such as pairs, as we do for instance in the expression in (11).

the expression in (3) x_1 is of type, say, $(s_1 \to t) \vee (s_2 \to \neg t)$, then we can assume that x_1 has type $(s_1 \to t)$ in the branch "then" and $(s_2 \to \neg t)$ in the branch "else". As a more concrete example, if $x_1 : (\mathsf{Int}\vee\mathsf{String} \to \mathsf{Int}) \vee (\mathsf{Bool}\vee\mathsf{String} \to \mathsf{Bool})$ and x_1x_2 is well-typed, then we can deduce for

$$(x_1x_2 \in \mathsf{Int})\,?(x_1(x_1x_2) + 42) : \mathtt{not}(x_1(x_1x_2)) \tag{6}$$

the type $\mathsf{Int} \vee \mathsf{Bool}$: in the "then" branch x_1 has type $\mathsf{Int}\vee\mathsf{String} \to \mathsf{Int}$ and x_1x_2 is of type Int; in the "else" branch x_1 has type $\mathsf{Bool}\vee\mathsf{String} \to \mathsf{Bool}$ and x_1x_2 is of type Bool.

Let us recap. If e is an expression of type t_0 and we are trying to type $(e \in t)?e_1 : e_2$, then we can assume that e has type $t_0 \wedge t$ when typing e_1 and type $t_0 \setminus t$ when typing e_2. If furthermore e is of the form $e'e''$, then we may also be able to specialize the types for e' (in particular if its static type is a union of arrows) and for e'' (in particular if the static type of e' is an intersection of arrows). Additionally, we can repeat the reasoning for all subterms of e' and e'' as long as they are applications, and deduce distinct types for all subexpressions of e that form applications. How to do it precisely—not only for applications, but also for other terms such as pairs, projections, records etc—is explained in the rest of the paper but the key ideas are pretty simple and are explained next.

1.1 Key ideas

First of all, in a strict language we can consider a type as denoting the set of values of that type and subtyping as set-containment of the denoted values. Imagine we are testing whether the result of an application e_1e_2 is of type t or not, and suppose we know that the static types of e_1 and e_2 are t_1 and t_2 respectively. If the application e_1e_2 is well typed, then there is a lot of useful information that we can deduce from it: first, that t_1 is a functional type (i.e., it denotes a set of well-typed lambda-abstractions, the values of functional type) whose domain, denoted by $\mathsf{dom}(t_1)$, is a type denoting the set of all values that are accepted by any function in t_1; second that t_2 must be a subtype of the domain of t_1; third, we also know the type of the application, that is the type that denotes all the values that may result from the application of a function in t_1 to an argument in t_2, type that we denote by $t_1 \circ t_2$. For instance, if $t_1 = \mathsf{Int} \to \mathsf{Bool}$ and $t_2 = \mathsf{Int}$, then $\mathsf{dom}(t_1) = \mathsf{Int}$ and $t_1 \circ t_2 = \mathsf{Bool}$. Notice that, introducing operations such as dom() and $\circ$ is redundant when working with simple types, but becomes necessary in the presence of set-theoretic types. If for instance t_1 is the type of (2), that is, $t_1 = (\mathsf{Int}\to\mathsf{Int}) \wedge (\mathsf{String}\to\mathsf{String})$, then $\mathsf{dom}(t) = \mathsf{Int} \vee \mathsf{String}$, that is the union of all the possible input types, while the precise return type of such a function depends on the type of the argument the function is applied to: either an integer, or a string, or both (i.e., the argument has union type $\mathsf{Int} \vee \mathsf{String}$). So we have $t_1 \circ \mathsf{Int} = \mathsf{Int}$, $t_1 \circ \mathsf{String} = \mathsf{String}$, and $t_1 \circ (\mathsf{Int} \vee \mathsf{String}) = \mathsf{Int} \vee \mathsf{String}$ (see Section 2.5.2 for the formal definition of $\circ$).

What we want to do is to refine the types of e_1 and e_2 (i.e., t_1 and t_2) for the cases where the test that e_1e_2 has type t succeeds or fails. Let us start with refining the type t_2 of e_2 for the case in which the test succeeds. Intuitively, we want to remove from t_2 all the values for which the application will surely return a result not in t, thus making the test fail. Consider t_1 and let s be the largest subtype of $\mathsf{dom}(t_1)$ such that

$$t_1 \circ s \leq \neg t \tag{7}$$

In other terms, s contains all the arguments that make any function in t_1 return a result not in t. Then we can safely remove from t_2 all the values in s or, equivalently, keep in t_2 all the values of $\mathsf{dom}(t_1)$ that are not in s. Let us implement the second viewpoint: the set of all elements of $\mathsf{dom}(t_1)$ for which an application *does not* surely give a result in $\neg t$ is denoted $t_1 \blacksquare t$ (read, "t_1 worra t") and defined as $\min\{u \leq \mathsf{dom}(t_1) \mid t_1 \circ (\mathsf{dom}(t_1) \setminus u) \leq \neg t\}$: it is easy to see that according to this definition $\mathsf{dom}(t_1) \setminus (t_1 \blacksquare t)$ is the largest subset of $\mathsf{dom}(t_1)$ satisfying (7). Then we can refine the type of e_2 for when the test is successful by using the type $t_2 \wedge (t_1 \blacksquare t)$: we intersect all the possible results of e_2, that is t_2, with the elements of the domain that *may* yield a result in t, that is $t_1 \blacksquare t$.

It is now easy to see how to refine the type of e_2 for when the test fails: simply use all the other possible results of e_2, that is $t_2 \setminus (t_1 \blacksquare t)$. To sum up, to refine the type of an argument in the test of an application, all we need is to define $t_1 \blacksquare t$, the set of arguments that when applied to a function of type t_1 *may* return a result in t; then we can refine the type of e_2 as $t_2^+ \stackrel{\text{def}}{=} t_2 \wedge (t_1 \blacksquare t)$ in the "then" branch (we call it the *positive* branch) and as $t_2^- \stackrel{\text{def}}{=} t_2 \setminus (t_1 \blacksquare t)$ in the "else" branch (we call it the *negative* branch). As a side remark note that the set $t_1 \blacksquare t$ is different from the set of elements that return a result in t (though it is a supertype of it). To see that, consider for t the type String and for t_1 the type $(\mathsf{Bool} \to \mathsf{Bool}) \wedge (\mathsf{Int} \to (\mathsf{String} \vee \mathsf{Int}))$, that is, the type of functions that when applied to a Boolean return a Boolean and when applied to an integer return either an integer or a string; then we have that $\mathsf{dom}(t_1) = \mathsf{Int} \vee \mathsf{Bool}$ and $t_1 \blacksquare \mathsf{String} = \mathsf{Int}$, but there is no (non-empty) type that ensures that an application of a function in t_1 will surely yield a String result.

Once we have determined t_2^+, it is then not very difficult to refine the type t_1 for the positive branch, too. If the test succeeded, then we know two facts: first, that the function was applied to a value in t_2^+ and, second, that the application did not diverge and, thus, returned a result in $t_1 \circ t_2^+$ (which is a subtype of $t_1 \circ t_2$). Therefore, we can exclude from t_1 all the functions that when applied to an argument in t_2^+ either diverge or yield a result not in $t_1 \circ t_2^+$. Both of these things can be obtained simply by removing from t_1 the functions in $(t_2^+ \to \neg(t_1 \circ t_2^+))$, that is, we refine the type of e_1 in the "then" branch as $t_1^+ = t_1 \setminus (t_2^+ \to \neg(t_1 \circ t_2^+))$. The fact that this removes the functions that applied to t_2^+ arguments yield results not in $t_1 \circ t_2^+$ should be pretty obvious. That this also removes functions diverging on t_2^+ arguments is subtler and deserves some explanation. In particular, the interpretation of a type $t \to s$ is the set of all functions that when applied to an argument of type t either diverge or return a value in s. As such the interpretation of $t \to s$ contains all the functions that diverge (at least) on t. Therefore removing $t \to s$ from a type u removes from u not only all the functions that when applied to a t argument return a result in s, but also all the functions that diverge on t. Ergo $t_1 \setminus (t_2^+ \to \neg(t_1 \circ t_2^+))$ removes, among others, all functions in t_1 that diverge on t_2^+. Let us see all this on our example (6), in particular, by showing how this technique deduces that the type of x_1 in the positive branch is (a subtype of) $\mathsf{Int} \vee \mathsf{String} \to \mathsf{Int}$. Take the static type of x_1, that is $(\mathsf{Int} \vee \mathsf{String} \to \mathsf{Int}) \vee (\mathsf{Bool} \vee \mathsf{String} \to \mathsf{Bool})$ and intersect it with $(t_2^+ \to \neg(t_1 \circ t_2^+))$, that is, $\mathsf{String} \to \neg\mathsf{Int}$. Since intersection distributes over unions we obtain

$$((\mathsf{Int} \vee \mathsf{String} \to \mathsf{Int}) \wedge \neg(\mathsf{String} \to \neg\mathsf{Int})) \vee ((\mathsf{Bool} \vee \mathsf{String} \to \mathsf{Bool}) \wedge \neg(\mathsf{String} \to \neg\mathsf{Int}))$$

and since $(\mathsf{Bool} \vee \mathsf{String} \to \mathsf{Bool}) \wedge \neg(\mathsf{String} \to \neg\mathsf{Int})$ is empty (because $\mathsf{String} \to \neg\mathsf{Int}$ contains $\mathsf{Bool} \vee \mathsf{String} \to \mathsf{Bool}$), then what we obtain is the left summand, a strict subtype of $\mathsf{Int} \vee \mathsf{String} \to \mathsf{Int}$, namely the functions of type $\mathsf{Int} \vee \mathsf{String} \to \mathsf{Int}$ minus those that diverge on all String arguments.

This is essentially what we formalize in Section 2, in the type system by the rule [PAppL] and in the typing algorithm with the case (19) of the definition of the function Constr.

1.2 Technical challenges

In the previous section we outlined the main ideas of our approach to occurrence typing. However, devil is in the details. So the formalization we give in Section 2 is not so smooth as we just outlined: we must introduce several auxiliary definitions to handle some corner cases. This section presents by tiny examples the main technical difficulties we had to overcome and the definitions we introduced to handle them. As such it provides a kind of road-map for the technicalities of Section 2.

Typing occurrences. As it should be clear by now, not only variables but also generic expression are given different types in the "then" and "else" branches of type tests. For instance, in (5) the expression $x_1 x_2$ has type Int in the positive branch and type Bool in the negative one. In this specific

case it is possible to deduce these typings from the refined types of the variables (in particular, thanks to the fact that x_2 has type Int the positive branch and Bool in the negative one), but this is not possible in general. For instance, consider $x_1 : \mathsf{Int} \to (\mathsf{Int} \vee \mathsf{Bool})$, $x_2 : \mathsf{Int}$, and the expression

$$(x_1 x_2 \in \mathsf{Int})\,?\,\ldots x_1 x_2 \ldots : \ldots x_1 x_2 \ldots \tag{8}$$

It is not possible to specialize the type of the variables in the branches. Nevertheless, we want to be able to deduce that $x_1 x_2$ has type Int in the positive branch and type Bool in the negative one. In order to do so in Section 2 we will use special type environments that map not only variables but also generic expressions to types. So to type, say, the positive branch of (8) we extend the current type environment with the hypothesis that the expression $x_1 x_2$ has type Int.

When we test the type of an expression we try to deduce the type of some subexpressions occurring in it. Therefore we must cope with subexpressions occurring multiple times. A simple example is given by using product types and pairs as in $((x, x) \in t_1 \times t_2)\,?\,e_1 : e_2$. It is easy to see that the positive branch e_1 is selected only if x has type t_1 *and* type t_2 and deduce from that that x must be typed in e_1 by their intersection, $t_1 \wedge t_2$. To deal with multiple occurrences of a same subexpression the type inference system of Section 2 will use the classic rule for introducing intersections [Inter], while the algorithmic counterpart will use the operator Refine() that intersects the static type of an expression with all the types deduced for the multiple occurrences of it.

Type preservation. We want our type system to be sound in the sense of Wright and Felleisen [1994], that is, that it satisfies progress and type preservation. The latter property is challenging because, as explained just above, our type assumptions are not only about variables but also about expressions. Two corner cases are particularly difficult. The first is shown by the following example

$$(e(42) \in \mathsf{Bool})\,?\,e : \ldots \tag{9}$$

If e is an expression of type $\mathsf{Int} \to t$, then, as discussed before, the positive branch will have type $(\mathsf{Int} \to t) \setminus (\mathsf{Int} \to \neg\mathsf{Bool})$. If furthermore the negative branch is of the same type (or of a subtype), then this will also be the type of the whole expression in (9). Now imagine that the application $e(42)$ reduces to a Boolean value, then the whole expression in (9) reduces to e; but this has type $\mathsf{Int} \to t$ which, in general, is *not* a subtype of $(\mathsf{Int} \to t) \setminus (\mathsf{Int} \to \neg\mathsf{Bool})$, and therefore type is not preserved by the reduction. To cope with this problem, in Section 2 we resort to *type schemes* a technique introduced by Frisch et al. [2008] to type expressions by sets of types, so that the expression in (9) will have both the types at issue.

The second corner case is a modification of the example above where the positive branch is $e(42)$, e.g., $(e(42) \in \mathsf{Bool})\,?\,e(42) : \mathsf{true}$. In this case the type deduced for the whole expression is Bool, while after reduction we would obtain the expression $e(42)$ which is not of type Bool but of type t (even though it will eventually reduce to a Bool). This problem will be handled in the proof of type preservation by considering parallel reductions (e.g, if $e(42)$ reduces in a step to, say, false, then $(e(42) \in \mathsf{Bool})\,?\,e(42) : \mathsf{true}$ reduces in one step to $(\mathsf{false} \in \mathsf{Bool})\,?\,\mathsf{false} : \mathsf{true}$): see Appendix A.2.

Interdependence of checks. The last class of technical problems arise from the mutual dependence of different type checks. In particular, there are two cases that pose a problem. The first can be shown by two functions f and g both of type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathbb{1} \to \mathsf{Bool})$, x of type $\mathbb{1}$ and the test:

$$((f\,x, g\,x) \in \mathsf{Int} \times \mathsf{Bool})\,?\,\ldots : \ldots \tag{10}$$

If we independently check $f\,x$ against Int and $g\,x$ against Bool we deduce Int for the first occurrence of x and $\mathbb{1}$ for the second. Thus we would type the positive branch of (10) under the hypothesis that x is of type Int. But if we use the hypothesis generated by the test of $f\,x$, that is, that x is of type Int, to check $g\,x$ against Bool, then the type deduced for x is $\mathbb{0}$—i.e., the branch is

never selected. In other words, we want to produce type environmments for occurrence typing by taking into account all the available hypotheses, even when these hypotheses are formulated later in the flow of control. This will be done in the type systems of Section 2 by the rule [Path] and will require at algorithmic level to look for a fix-point solution of a function, or an approximation thereof.

Finally, a nested check may help refining the type assumptions on some outer expressions. For instance, when typing the positive branch e of

$$((x, y) \in ((\mathsf{Int} \vee \mathsf{Bool}) \times \mathsf{Int}))\,?e : \ldots \tag{11}$$

we can assume that the expression (x, y) is of type $(\mathsf{Int} \vee \mathsf{Bool}) \times \mathsf{Int}$ and put it in the type environment. But if in e there is a test like $(x \in \mathsf{Int})?(x, y) : (\ldots)$ then we do not want use the assumption in the type environment to type the expression (x, y) occurring in the inner test (in red). Instead we want to give to that occurrence of the expression (x, y) the type $\mathsf{Int} \times \mathsf{Int}$. This will be done by temporary removing the type assumption about (x, y) from the type environment and by retyping the expression without that assumption (see rule $[\textsc{Env}_{\not\alpha}]$ in Section 2.5.4).

Outline. In Section 2 we formalize the ideas presented in this introduction: we define the types and expressions of our system, their dynamic semantics and a type system that implements occurrence typing together with the algorithms that decide whether an expression is well typed or not. Section 3 extends our formalism to record types and presents two applications of our analysis: the inference of arrow types for functions and a static analysis to reduce the number of casts inserted by a compiler of a gradually-typed language. Practical aspects are discussed in Section 4 where we give several paradigmatic examples of code typed by our prototype implementation, that can be interactively tested at https://occtyping.github.io/. Section 5 presents related work. A discussion of future work concludes this presentation. For space reasons several technical definitions and all the proofs are omitted from this presentation and can be found in the appendix available online.

2 LANGUAGE

In this section we formalize the ideas we outlined in the introduction. We start by the definition of types followed by the language and its reduction semantics. The static semantics is the core of our work: we first present a declarative type system that deduces (possibly many) types for well-typed expressions and then the algorithms to decide whether an expression is well typed or not.

2.1 Types

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 use the following abbreviations: $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge \neg t_2$, $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$. 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). It also ensures that the binary relation $\rhd \subseteq \textbf{Types}^2$ defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian. This

gives an induction principle on **Types** that we will use without any further explicit reference to the relation.[5] We refer to b, $\times$, and $\to$ as *type constructors* and to $\vee$, $\wedge$, $\neg$, and $\setminus$ as *type connectives*.

The subtyping relation for these types, noted $\leq$, is the one defined by Frisch et al. [2008] to which the reader may refer. A detailed description of the algorithm to decide it can be found in [Castagna 2019]. 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[6] 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 Syntax

The expressions e and values v of our language are inductively generated by the following grammars:

$$\begin{array}{lrcl} \textbf{Expr} & e & ::= & c \mid x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid \pi_j e \mid (e,e) \mid (e \in t)\,?\,e : e \\ \textbf{Values} & v & ::= & c \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid (v,v) \end{array} \tag{12}$$

for $j = 1, 2$. In (12), c ranges over constants (e.g., `true`, `false`, `1`, `2`, ...) which are values of basic types (we use $\boldsymbol{b}_c$ to denote the basic type of the constant c); x ranges over variables; (e, e) denote pairs and $\pi_i e$ their projections; $(e \in t)\,?\,e_1 : e_2$ denotes the type-case expression that evaluates either e_1 or e_2 according to whether the value returned by e (if any) is of type t or not; $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ is a value of type $\wedge_{i\in I} s_i \to t_i$ and denotes the function of parameter x and body e. An expression has an intersection type if and only if it has all the types that compose the intersection. Therefore, intuitively, $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ is a well-typed value if for all $i \in I$ the hypothesis that x is of type s_i implies that the body e has type t_i, that is to say, it is well typed if $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ has type $s_i \to t_i$ for all $i \in I$. Every value is associated to a type: the type of c is $\boldsymbol{b}_c$; the type of $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ is $\wedge_{i\in I} s_i \to t_i$; and, inductively, the type of a pair of values is the product of the types of the values.

2.3 Dynamic semantics

The dynamic semantics is defined as a classic left-to-right call-by-value reduction for a λ-calculus with pairs, enriched with specific rules for type-cases. We have the following notions of reduction:

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

The semantics of type-cases uses the relation $v \in t$ that we informally defined in the previous section. We delay its formal definition to Section 2.5.1 (where it deals with some corner cases for negated arrow types). Contextual reductions are defined by the following evaluation contexts:

$$C[] ::= [\,] \mid Ce \mid vC \mid (C, e) \mid (v, C) \mid \pi_i C \mid (C \in t)\,?\,e : e$$

As usual we denote by $C[e]$ the term obtained by replacing e for the hole in the context C and we have that $e \rightsquigarrow e'$ implies $C[e] \rightsquigarrow C[e']$.

[5] In a nutshell, we can do proofs by induction on the structure of unions and negations—and, thus, intersections—but arrows, products, and basic types are the base cases for the induction.

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

2.4 Static semantics

While the syntax and reduction semantics are, on the whole, pretty standard for the type system, we will have to introduce several unconventional features that we anticipated in Section 1.2 and are at the core of our work. Let us start with the standard part, that is the typing of the functional core and the use of subtyping, given by the following typing rules:

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b}_c} \qquad [\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs+}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\textsc{Subs}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

These rules are quite standard and do not need any particular explanation besides those already given in Section 2.2. Just notice that we used a classic subsumption rule (i.e., [Subs]) to embed subtyping in the type system. Let us next focus on the unconventional aspects of our system, from the simplest to the hardest.

The first unconventional aspect is that, as explained in Section 1.2, our type assumptions are about expressions. Therefore, in our rules the type environments, ranged over by Γ, map *expressions*—rather than just variables—into types. This explains why the classic typing rule for variables is replaced by a more general [Env] rule defined below:

$$[\textsc{Env}]\ \frac{}{\Gamma \vdash e : \Gamma(e)}\ e \in \mathrm{dom}(\Gamma) \qquad [\textsc{Inter}]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

The [Env] rule is coupled with the standard intersection introduction rule [Inter] which allows us to deduce for a complex expression the intersection of the types recorded by the occurrence typing analysis in the environment Γ with the static type deduced for the same expression by using the other typing rules. This same intersection rule is also used to infer the second unconventional aspect of our system, that is, the fact that λ-abstractions can have negated arrow types, as long as these negated types do not make the type deduced for the function empty:

$$[\textsc{Abs-}]\ \frac{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : t}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \neg(t_1 \to t_2)}\ ((\wedge_{i\in I} s_i \to t_i) \wedge \neg(t_1 \to t_2)) \not\simeq \mathbb{0}$$

As explained in Section 1.2, we need to be able to deduce for, say, the function $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.x$ a type such as $(\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ (in particular, if this is the term e in equation (9) we need to deduce for it the type $(\mathsf{Int} \to t) \wedge \neg(\mathsf{Int} \to \neg\mathsf{Bool})$, that is, $(\mathsf{Int} \to t) \setminus (\mathsf{Int} \to \neg\mathsf{Bool})$). But the sole rule [Abs+] above does not allow us to deduce negations of arrows for abstractions: the rule [Abs-] makes this possible. As an aside, note that this kind of deduction was already present in the system by Frisch et al. [2008] though in that system this presence was motivated by the semantics of types rather than, as in our case, by the soundness of the type system.

Rules [Abs+] and [Abs-] are not enough to deduce for λ-abstractions all the types we wish. In particular, these rules alone are not enough to type general overloaded functions. For instance, consider this simple example of a function that applied to an integer returns its successor and applied to anything else returns true:

$$\lambda^{(\mathsf{Int}\to\mathsf{Int})\wedge(\neg\mathsf{Int}\to\mathsf{Bool})} x\,.\,(x{\in}\mathsf{Int})\,?\,x+1 : \mathsf{true}$$

Clearly, the expression above is well typed, but the rule [Abs+] alone is not enough to type it. In particular, according to [Abs+] we have to prove that under the hypothesis that x is of type Int the expression $((x{\in}\mathsf{Int})\,?\,x+1 : \mathsf{true})$ is of type Int, too. That is, that under the hypothesis that x has type $\mathsf{Int}\wedge\mathsf{Int}$ (we apply occurrence typing) the expression $x+1$ is of type Int (which holds) and that under the hypothesis that x has type $\mathsf{Int}\setminus\mathsf{Int}$, that is $\mathbb{0}$ (we apply once more occurrence typing),

true is of type Int (which *does not* hold). The problem is that we are trying to type the second case of a type-case even if we know that there is no chance that, when x is bound to an integer, that case will be ever selected. The fact that it is never selected is witnessed by the presence of a type hypothesis with $\mathbb{0}$ type. To avoid this problem (and type the term above) we add the rule [Efq] (*ex falso quodlibet*) that allows the system to deduce any type for an expression that will never be selected, that is, for an expression whose type environment contains an empty assumption:

$$[\textsc{Efq}]\ \frac{}{\Gamma, (e : \mathbb{0}) \vdash e' : t}$$

Once more, this kind of deduction was already present in the system by Frisch et al. [2008] to type full fledged overloaded functions, though it was embedded in the typing rule for the type-case. Here we need the rule [Efq], which is more general, to ensure the property of subject reduction.

Finally, there remains one last rule in our type system, the one that implements occurrence typing, that is, the rule for the type-case:

$$[\textsc{Case}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_1 \qquad \Gamma_1 \vdash e_1 : t' \qquad \Gamma \vdash^{\mathsf{Env}}_{e,\neg t} \Gamma_2 \qquad \Gamma_2 \vdash e_2 : t'}{\Gamma \vdash (e{\in}t)\,?\,e_1 : e_2 : t'}$$

The rule [Case] checks whether the expression e, whose type is being tested, is well-typed and then performs the occurrence typing analysis that produces the environments Γ_i's under whose hypothesis the expressions e_i's are typed. The production of these environments is represented by the judgments $\Gamma \vdash^{\mathsf{Env}}_{e,(\neg)t} \Gamma_i$. The intuition is that when $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_1$ is provable then Γ_1 is a version of Γ extended with type hypotheses for all expressions occurring in e, type hypotheses that can be deduced assuming that the test $e \in t$ succeeds. Likewise, $\Gamma \vdash^{\mathsf{Env}}_{e,\neg t} \Gamma_2$ (notice the negation on t) extends Γ with the hypothesis deduced assuming that $e \in \neg t$, that is, for when the test $e \in t$ fails.

All it remains to do is to show how to deduce judgments of the form $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$. For that we first define how to denote occurrences of an expression. These are identified by paths in the syntax tree of the expressions, that is, by possibly empty strings of characters denoting directions starting from the root of the tree (we use ϵ for the empty string/path, which corresponds to the root of the tree).

Let e be an expression and $\varpi \in \{0, 1, l, r, f, s\}^*$ a *path*; we denote $e{\downarrow}\varpi$ the occurrence of e reached by the path ϖ, that is (for $i = 1, 2$, and undefined otherwise)

$$\begin{array}{rclcrclcrcl} e{\downarrow}\epsilon & = & e & & (e_0, e_1){\downarrow}l.\varpi & = & e_0{\downarrow}\varpi & & \pi_1 e{\downarrow}f.\varpi & = & e{\downarrow}\varpi \\ e_0 e_1{\downarrow}i.\varpi & = & e_i{\downarrow}\varpi & & (e_0, e_1){\downarrow}r.\varpi & = & e_1{\downarrow}\varpi & & \pi_2 e{\downarrow}s.\varpi & = & e{\downarrow}\varpi \end{array}$$

To ease our analysis we used different directions for each kind of term. So we have 0 and 1 for the function and argument of an application, l and r for the *l*eft and *r*ight expressions forming a pair, and f and s for the argument of a *f*irst or of a *s*econd projection. Note also that we do not consider occurrences under λ's (since their type is frozen in their annotations) and type-cases (since they reset the analysis). The judgments $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ are then deduced by the following two rules:

$$[\textsc{Base}]\ \frac{}{\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma} \qquad [\textsc{Path}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma',e,t} \varpi : t' \qquad \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'}{\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma', (e{\downarrow}\varpi : t')}$$

These rules describe how to produce by occurrence typing the type environments while checking that an expression e has type t. They state that (i) we can deduce from Γ all the hypothesis already in Γ (rule [Base]) and that (ii) if we can deduce a given type t' for a particular occurrence ϖ of the expression e being checked, then we can add this hypothesis to the produced type environment (rule [Path]). The rule [Path] uses a (last) auxiliary judgement $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'$ to deduce the type t' of the occurrence $e{\downarrow}\varpi$ when checking e against t under the hypotheses Γ. This rule [Path] is subtler than it may appear at first sight, insofar as the deduction of the type for ϖ may already use some hypothesis on $e{\downarrow}\varpi$ (in Γ') and, from an algorithmic viewpoint, this will imply the computation of a fix-point (see Section 2.5.3). The last ingredient for our type system is the deduction of the

judgements of the form $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'$ where ϖ is a path to an expression occurring in e. This is given by the following set of rules.

$$[\textsc{PSubs}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \qquad t_1 \leq t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2} \qquad [\textsc{PInter}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \qquad \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \wedge t_2} \qquad [\textsc{PTypeof}]\ \frac{\Gamma \vdash e{\downarrow}\varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}$$

$$[\textsc{PEps}]\ \frac{}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \epsilon : t} \qquad [\textsc{PAppR}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.0 : t_1 \to t_2 \qquad \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.1 : \neg t_1}\ t_2 \wedge t_2' \simeq \mathbb{0}$$

$$[\textsc{PAppL}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.1 : t_1 \qquad \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.0 : \neg(t_1 \to \neg t_2)} \qquad [\textsc{PPairL}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \times t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.l : t_1}$$

$$[\textsc{PPairR}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \times t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.r : t_2} \qquad [\textsc{PFst}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.f : t' \times \mathbb{1}} \qquad [\textsc{PSnd}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.s : \mathbb{1} \times t'}$$

These rules implement the analysis described in Section 1.1 for functions and extend it to products. Let us comment each rule in detail. [PSubs] is just subsumption for the deduction $\vdash^{\mathsf{Path}}$. The rule [PInter] combined with [PTypeof] allows the system to deduce for an occurrence ϖ the intersection of the static type of $e{\downarrow}\varpi$ (deduced by [PTypeof]) with the type deduced for ϖ by the other $\vdash^{\mathsf{Path}}$ rules. The rule [PEps] is the starting point of the analysis: if we are assuming that the test $e \in t$ succeeds, then we can assume that e (i.e., $e{\downarrow}\epsilon$) has type t (recall that assuming that the test $e \in t$ fails corresponds to having $\neg t$ at the index of the turnstyle). The rule [PAppR] implements occurrence typing for the arguments of applications, since it states that if a function maps arguments of type t_1 in results of type t_2 and an application of this function yields results (in t_2') that cannot be in t_2 (since $t_2 \wedge t_2' \simeq \mathbb{0}$), then the argument of this application cannot be of type t_1. [PAppR] performs the occurrence typing analysis for the function part of an application, since it states that if an application has type t_2 and the argument of this application has type t_1, then the function in this application cannot have type $t_1 \to \neg t_2$. Rules [PPair_] are straightforward since they state that the i-th projection of a pair that is of type $t_1 \times t_2$ must be of type t_i. So are the last two rules that essentially state that if $\pi_1 e$ (respectively, $\pi_2 e$) is of type t', then the type of e must be of the form $t' \times \mathbb{1}$ (respectively, $\mathbb{1} \times t'$).

This concludes the presentation of our type system, which satisfies the property of safety, deduced, as customary, from the properties of progress and subject reduction (*cf.* Appendix A.3).

Theorem 2.2 (type safety). *For every expression e such that $\varnothing \vdash e : t$ either e diverges or there exists a value v of type t such that $e \rightsquigarrow^* v$.*

2.5 Algorithmic system

The system we defined in the previous section implements the ideas we illustrated in the introduction and it is safe. Now the problem is to decide whether an expression is well typed or not, that is, to find an algorithm that given a type environment Γ and an expression e decides whether there exists a type t such that $\Gamma \vdash e : t$ is provable. For that we need to solve essentially two problems: (*i*) how to handle the fact that it is possible to deduce several types for the same well-typed expression and (*ii*) how to compute the auxiliary deduction system for paths.

Multiple types have two distinct origins each requiring a distinct technical solution. The first origin is the rule [Abs-] by which it is possible to deduce for every well-typed lambda abstraction infinitely many types, that is the annotation of the function intersected with as many negations of arrow types as it is possible without making the type empty. To handle this multiplicity we use and adapt the technique of *type schemes* defined by Frisch et al. [2008]. Type schemes—whose

definition we recall in Section 2.5.1—are canonical representations of the infinite sets of types of λ-abstractions. The second origin is due to the presence of structural rules[7] such as [Subs] and [Inter]. We handle this presence in the classic way: we define an algorithmic system that tracks the miminum type—actually, the minimum *type scheme*—of an expression; this system is obtained from the original system by removing the two structural rules and by distributing suitable checks of the subtyping relation in the remaining rules. To do that in the presence of set-theoretic types we need to define some operators on types, which are given in Section 2.5.2.

For what concerns the use of the auxiliary derivation for the judgments, we present in Section 2.5.3 an algorithm that is sound and satisfies a limited form of completeness. All these notions are then used in the algorithmic typing system given in Section 2.5.4.

2.5.1 Type schemes. We introduce the new syntactic category of *types schemes* which are the terms $\mathbb{t}$ inductively produced by the following grammar.

$$\textbf{Type schemes} \quad \mathbb{t} \quad ::= \quad t \mid [t \to t; \cdots; t \to t] \mid \mathbb{t} \otimes \mathbb{t} \mid \mathbb{t} \between \mathbb{t} \mid \Omega$$

Type schemes denote sets of types, as formally stated by the following definition:

Definition 2.3 (Interpretation of type schemes). We define the function $\{\_\}$ that maps type schemes into sets of types.

$$\begin{array}{lcl}
\{t\} & = & \{s \mid t \leq s\} \\
\{[t_i \to s_i]_{i=1..n}\} & = & \{s \mid \exists s_0 = \bigwedge_{i=1..n} t_i \to s_i \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j).\ \mathbb{0} \not\simeq s_0 \leq s\} \\
\{\mathbb{t}_1 \otimes \mathbb{t}_2\} & = & \{s \mid \exists t_1 \in \{\mathbb{t}_1\}\ \exists t_2 \in \{\mathbb{t}_2\}.\ t_1 \times t_2 \leq s\} \\
\{\mathbb{t}_1 \between \mathbb{t}_2\} & = & \{s \mid \exists t_1 \in \{\mathbb{t}_1\}\ \exists t_2 \in \{\mathbb{t}_2\}.\ t_1 \vee t_2 \leq s\} \\
\{\Omega\} & = & \varnothing
\end{array}$$

Note that $\{\mathbb{t}\}$ is closed under subsumption and intersection and that Ω, which denotes the empty set of types is different from $\mathbb{0}$ whose interpretation is the set of all types.

Lemma 2.4 ([Frisch et al. 2008]). *Let $\mathbb{t}$ be a type scheme and t a type. It is possible to decide the assertion $t \in \{\mathbb{t}\}$, which we also write $\mathbb{t} \leq t$.*

We can now formally define the relation $v \in t$ used in Section 2.3 to define the dynamic semantics of the language. First, we associate each (possibly, not well-typed) value to a type scheme representing the best type information about the value. By induction on the definition of values: $\mathsf{typeof}(c) = \boldsymbol{b}_c$, $\mathsf{typeof}(\lambda^{\wedge_{i\in I} s_i \to t_i} x.e) = [s_i \to t_i]_{i\in I}$, $\mathsf{typeof}((v_1, v_2)) = \mathsf{typeof}(v_1) \otimes \mathsf{typeof}(v_1)$. Then we have $v \in t \iff^{\text{def}} \mathsf{typeof}(v) \leq t$.

We also need to perform intersections of type schemes so as to intersect the static type of an expression (i.e., the one deduced by conventional rules) with the one deduced by occurrence typing (i.e., the one derived by $\vdash^{\mathsf{Path}}$). For our algorithmic system (see [$\mathrm{Env}_{\mathcal{A}}$] in Section 2.5.4) all we need to define is the intersection of a type scheme with a type:

Lemma 2.5 ([Frisch et al. 2008]). *Let $\mathbb{t}$ be a type scheme and t a type. We can compute a type scheme, written $t \between \mathbb{t}$, such that $\{t \between \mathbb{t}\} = \{s \mid \exists t' \in \{\mathbb{t}\}.\ t \wedge t' \leq s\}$*

Finally, given a type scheme $\mathbb{t}$ it is straightforward to choose in its interpretation a type $\mathsf{Repr}(\mathbb{t})$ which serves as the canonical representative of the set (i.e., $\mathsf{Repr}(\mathbb{t}) \in \{\mathbb{t}\}$):

Definition 2.6 (Representative). We define a function $\mathsf{Repr}(\_)$ that maps every non-empty type scheme into a type, *representative* of the set of types denoted by the scheme.

[7] In logic, logical rules refer to a particular connective (here, a type constructor, that is, either $\to$, or $\times$, or b), while identity rules (e.g., axioms and cuts) and structural rules (e.g., weakening and contraction) do not.

$$\begin{array}{lcllcl} \mathsf{Repr}(t) & = & t & \mathsf{Repr}(\mathbb{t}_1 \otimes \mathbb{t}_2) & = & \mathsf{Repr}(\mathbb{t}_1) \times \mathsf{Repr}(\mathbb{t}_2) \\ \mathsf{Repr}([t_i \to s_i]_{i\in I}) & = & \bigwedge_{i\in I} t_i \to s_i & \mathsf{Repr}(\mathbb{t}_1 \oslash \mathbb{t}_2) & = & \mathsf{Repr}(\mathbb{t}_1) \vee \mathsf{Repr}(\mathbb{t}_2) \\ \mathsf{Repr}(\Omega) & & \textit{undefined} \end{array}$$

2.5.2 Operators for type constructors. In order to define the algorithmic typing of expressions like applications and projections we need to define the operators on types we used in Section 1.1. Consider the rule [App] for applications. It essentially does three things: (1) it checks that the function has functional type; (2) it checks that the argument is in the domain of the function, and (3) it returns the type of the application. In systems without set-theoretic types these operations are quite straightforward: (1) corresponds to checking that the function has an arrow type, (2) corresponds to checking that the argument is in the domain of the arrow deduced for the function and (3) corresponds to returning the codomain of that same arrow. With set-theoretic types things get more difficult, since a function can be typed by, say, a union of intersection of arrows and negations of types. Checking that the function 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 dom() and $\circ$ we informally described in Section 1.1 where we also introduced the operator $\blacksquare$. These three operators are used by our algorithm and formally defined as:

$$\begin{array}{rcll} \mathsf{dom}(t) & = & \max\{u \mid t \leq u \to \mathbb{1}\} & (13) \\ t \circ s & = & \min\{u \mid t \leq s \to u\} & (14) \\ t \blacksquare s & = & \min\{u \mid t \circ (\mathsf{dom}(t) \setminus u) \leq \neg s\} & (15) \end{array}$$

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 $t \leq \mathbb{1}\times\mathbb{1}$, then we define:

$$\boldsymbol{\pi_1}(t) = \min\{u \mid t \leq u \times \mathbb{1}\} \qquad \boldsymbol{\pi_2}(t) = \min\{u \mid t \leq \mathbb{1} \times u\} \quad (16)$$

All the operators above but $\blacksquare$ are already present in the theory of semantic subtyping: the reader can find how to compute them and how to extend their definition to type schemes in [Frisch et al. 2008, Section 6.11] (see also [Castagna 2019, §4.4] for a detailed description). Below we just show the formula that computes $t \blacksquare s$ for a t subtype of $\mathbb{0} \to \mathbb{1}$. For that, we use a result of semantic subtyping that states that every type t is equivalent to a type in disjunctive normal form and that if furthermore $t \leq \mathbb{0} \to \mathbb{1}$, then $t \simeq \bigvee_{i\in I}\left(\bigwedge_{p\in P_i}(s_p \to t_p)\bigwedge_{n\in N_i}\neg(s'_n \to t'_n)\right)$ with $\bigwedge_{p\in P_i}(s_p \to t_p)\bigwedge_{n\in N_i}\neg(s'_n \to t'_n) \not\simeq \mathbb{0}$ for all i in I. For such a t and any type s then we have:

$$t \blacksquare s = \mathsf{dom}(t) \wedge \bigvee_{i\in I}\left(\bigwedge_{\{P\subseteq P_i \mid s\leq\bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right) \quad (17)$$

The formula considers only the positive arrows of each summand that forms t and states that, for each summand, whenever you take a subset P of its positive arrows that cannot yield results in s (since s does not overlap the intersection of the codomains of these arrows), then the success of the test cannot depend on these arrows and therefore the intersection of the domains of these arrows—i.e., the values that would precisely select that set of arrows—can be removed from $\mathsf{dom}(t)$. The proof that this type satisfies (15) is given in the Appendix A.4.

2.5.3 Type environments for occurrence typing. The last step for our presentation is to define the algorithm for the deduction of $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$, that is an algorithm that takes as input Γ, e, and t, and returns an environment that extends Γ with hypotheses on the occurrences of e that are the most general that can be deduced by assuming that $e \in t$ succeeds. For that we need the notation $\mathsf{typeof}_\Gamma(e)$ which denotes the type scheme deduced for e under the type environment Γ in the algorithmic type system of Section 2.5.4. That is, $\mathsf{typeof}_\Gamma(e) = \mathbb{t}$ if and only if $\Gamma \vdash_{\mathcal{A}} e : \mathbb{t}$ is provable.

We start by defining the algorithm for each single occurrence, that is for the deduction of $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'$. This is obtained by defining two mutually recursive functions Constr and Intertype:

$$\mathsf{Constr}_{\Gamma,e,t}(\epsilon) = t \tag{18}$$
$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.0) = \neg(\mathsf{Intertype}_{\Gamma,e,t}(\varpi.1) \to \neg\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{19}$$
$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.1) = \mathsf{Repr}(\mathsf{typeof}_\Gamma(e{\downarrow}\varpi.0)) \blacksquare \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \tag{20}$$
$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.l) = \boldsymbol{\pi_1}(\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{21}$$
$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.r) = \boldsymbol{\pi_2}(\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{22}$$
$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.f) = \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \times \mathbb{1} \tag{23}$$
$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.s) = \mathbb{1} \times \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \tag{24}$$

$$\mathsf{Intertype}_{\Gamma,e,t}(\varpi) = \mathsf{Repr}(\mathsf{Constr}_{\Gamma,e,t}(\varpi) \mathbin{⊛} \mathsf{typeof}_\Gamma(e{\downarrow}\varpi)) \tag{25}$$

All the functions above are defined if and only if the initial path ϖ is valid for e (i.e., $e{\downarrow}\varpi$ is defined) and e is well-typed (which implies that all $\mathsf{typeof}_\Gamma(e{\downarrow}\varpi)$ in the definition are defined).[8]

Each case of the definition of the Constr function corresponds to the application of a logical rule (*cf.* Footnote 7) in the deduction system for $\vdash^{\mathsf{Path}}$: case (18) corresponds to the application of [PEps]; case (19) implements [Appl] straightforwardly; the implementation of rule [AppR] is subtler: instead of finding the best t_1 to subtract (by intersection) from the static type of the argument, (20) finds directly the best type for the argument by applying the $\blacksquare$ operator to the static types of the function and the refined type of the application. The remaining (21–24) cases are the straightforward implementations of the rules [PPairL], [PPairR], [Pfst], and [Psnd], respectively.

The other recursive function, Intertype, implements the two structural rules [Pinter] and [PTypeof] by intersecting the type obtained for ϖ by the logical rules, with the static type deduced by the type system for the expression occurring at ϖ. The remaining structural rule, [Psubs], is accounted for by the use of the operators $\blacksquare$ and $\boldsymbol{\pi}_i$ in the definition of Constr.

It remains to explain how to compute the environment Γ' produced from Γ by the deduction system for $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$. Alas, this is the most delicate part of our algorithm. In a nutshell what we want to do is to define a function $\mathsf{Refine}_{\_,\_}(\_)$ that takes a type environment Γ, an expression e and a type t and returns the best type environment Γ' such that $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ holds. By the best environment we mean the one in which the occurrences of e are associated to the largest possible types (type environments are hypotheses so they are contravariant: the larger the type the better the hypothesis). Recall that in Section 1.2 we said that we want our analysis to be able to capture all the information available from nested checks. If we gave up such a kind of precision then the definition of Refine would be pretty easy: it must map each subexpression of e to the intersection of the types deduced by $\vdash^{\mathsf{Path}}$ (i.e., by Intertype) for each of its occurrences. That is, for each expression e' occurring in e, $\mathsf{Refine}_{e,t}(\Gamma)$ would be the type environment that maps e' into $\bigwedge_{\{\varpi \mid e{\downarrow}\varpi \equiv e'\}} \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$. As we explained in Section 1.2 the intersection is needed to apply occurrence typing to expressions such as $((x,x){\in}t_1 \times t_2)\,?\,e_1 : e_2$ where some expressions—here x—occur multiple times.

In order to capture most of the type information from nested queries the rule [Path] allows the deduction of the type of some occurrence ϖ to use a type environment Γ' that may contain information about some suboccurrences of ϖ. On the algorithm this would correspond to applying the Refine defined above to an environment that already is the result of Refine, and so on. Therefore, ideally our algorithm should compute the type environment as a fixpoint of the function $X \mapsto \mathsf{Refine}_{e,t}(X)$. Unfortunately, an iteration of Refine may not converge. As an example, consider the (dumb) expression $(x_1x_2{\in}\mathbb{1})\,?\,e_1 : e_2$. If $x_1 : \mathbb{1} \to \mathbb{1}$, then every iteration of Refine yields for x_1 a type strictly more precise than the type deduced in the previous iteration.

[8] Note that the definition is well-founded. This can be seen by analyzing the rule [Case$_\mathcal{A}$]: the definition of $\mathsf{Refine}_{e,t}(\Gamma)$ and $\mathsf{Refine}_{e,\neg t}(\Gamma)$ use $\mathsf{typeof}_\Gamma(e{\downarrow}\varpi)$, and this is defined for all ϖ since the first premisses of [Case$_\mathcal{A}$] states that $\Gamma \vdash e : \mathbb{t}_0$ (and this is possible only if we were able to deduce under the hypothesis Γ the type of every occurrence of e.)

The solution we adopt here is to bound the number of iterations to some number n_o. From a formal point of view, this means to give up the completeness of the algorithm: Refine will be complete (i.e., it will find all solutions) for the deductions of $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ of depth at most n_o. This is obtained by the following definition of Refine

$$\mathsf{Refine}_{e,t} \stackrel{\text{def}}{=} (\mathsf{RefineStep}_{e,t})^{n_o}$$

$$\text{where } \mathsf{RefineStep}_{e,t}(\Gamma)(e') = \begin{cases} \bigwedge_{\{\varpi \mid e\downarrow\varpi \equiv e'\}} \mathsf{Intertype}_{\Gamma,e,t}(\varpi) & \text{if } \exists\varpi.\ e\downarrow\varpi \equiv e' \\ \Gamma(e') & \text{otherwise, if } e' \in \mathsf{dom}(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note in particular that $\mathsf{Refine}_{e,t}(\Gamma)$ extends Γ with hypotheses on the expressions occurring in e, since $\mathsf{dom}(\mathsf{Refine}_{e,t}(\Gamma)) = \mathsf{dom}(\mathsf{RefineStep}_{e,t}(\Gamma)) = \mathsf{dom}(\Gamma) \cup \{e' \mid \exists\varpi.\ e\downarrow\varpi \equiv e'\}$.

In other terms, we try to find a fixpoint of $\mathsf{RefineStep}_{e,t}$ but we bound our search to n_o iterations. Since $\mathsf{RefineStep}_{e,t}$ is monotone (w.r.t. the subtyping pre-order extended to type environments pointwise), then every iteration yields a better solution.

While this is unsatisfactory from a formal point of view, in practice the problem is a very mild one. Divergence may happen only when refining the type of a function in an application: not only such a refinement is meaningful only when the function is typed by a union type (which is quite rare in practice)[9], but also we had to build the expression that causes the divergence in quite an *ad hoc* way which makes divergence even more unlikely.

2.5.4 Algorithmic typing rules. We now have all the notions we need for our typing algorithm, which is defined by the following rules.

$$[\textsc{Efq}_{\mathcal{A}}]\ \frac{}{\Gamma, (e : \mathbb{0}) \vdash_{\mathcal{A}} e' : \mathbb{0}}\ \text{with priority over all the other rules} \qquad [\textsc{Var}_{\mathcal{A}}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

$$[\textsc{Env}_{\mathcal{A}}]\ \frac{\Gamma \setminus \{e\} \vdash_{\mathcal{A}} e : \mathbb{t}}{\Gamma \vdash_{\mathcal{A}} e : \Gamma(e) \mathbin{\mathbb{\&}} \mathbb{t}}\ e \in \mathsf{dom}(\Gamma) \text{ and } e \text{ not a variable} \qquad [\textsc{Const}_{\mathcal{A}}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} c : \boldsymbol{b_c}}\ c \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{Abs}_{\mathcal{A}}]\ \frac{\Gamma, x : s_i \vdash_{\mathcal{A}} e : \mathbb{t}'_i \qquad \mathbb{t}'_i \leq t_i}{\Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : [s_i \to t_i]_{i\in I}}\ \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{App}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e_1 : \mathbb{t}_1 \qquad \Gamma \vdash_{\mathcal{A}} e_2 : \mathbb{t}_2 \qquad \mathbb{t}_1 \leq \mathbb{0} \to \mathbb{1} \qquad \mathbb{t}_2 \leq \mathsf{dom}(\mathbb{t}_1)}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \mathbb{t}_1 \circ \mathbb{t}_2}\ e_1 e_2 \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{Case}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e : \mathbb{t}_0 \qquad \mathsf{Refine}_{e,t}(\Gamma) \vdash_{\mathcal{A}} e_1 : \mathbb{t}_1 \qquad \mathsf{Refine}_{e,\neg t}(\Gamma) \vdash_{\mathcal{A}} e_2 : \mathbb{t}_2}{\Gamma \vdash_{\mathcal{A}} (e \in t)\,?\,e_1 : e_2 : \mathbb{t}_1 \mathbin{\mathbb{\vee}} \mathbb{t}_2}\ (e \in t)\,?e_1 : e_2 \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{Proj}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e : \mathbb{t} \qquad \mathbb{t} \leq \mathbb{1}\times\mathbb{1}}{\Gamma \vdash_{\mathcal{A}} \pi_i e : \boldsymbol{\pi_i}(\mathbb{t})}\ \pi_i e \notin \mathsf{dom}(\Gamma) \qquad [\textsc{Pair}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e_1 : \mathbb{t}_1 \qquad \Gamma \vdash_{\mathcal{A}} e_2 : \mathbb{t}_2}{\Gamma \vdash_{\mathcal{A}} (e_1, e_2) : \mathbb{t}_1 \otimes \mathbb{t}_2}\ (e_1, e_2) \notin \mathsf{dom}(\Gamma)$$

The side conditions of the rules ensure that the system is syntax directed, that is, that at most one rule applies when typing a term: priority is given to $[\textsc{Eqf}_{\mathcal{A}}]$ over all the other rules and to $[\textsc{Env}_{\mathcal{A}}]$ over all remaining logical rules. Type schemes are used to account for the type-multiplicity stemming from λ-abstractions as shown in particular by rule $[\textsc{Abs}_{\mathcal{A}}]$ (in what follows we use the word "type" also for type schemes). The subsumption rule is no longer in the system; it is replaced by: (*i*) using a union type in $[\textsc{Case}_{\mathcal{A}}]$, (*ii*) checking in $[\textsc{Abs}_{\mathcal{A}}]$ that the body of the function is typed by a subtype of the type declared in the annotation, and (*iii*) using type operators and checking

[9] The only impact of adding a negated arrow type to the type of a function is when we test whether the function has a given arrow type: in practice this never happens since programming languages test whether a value *is* a function, rather than the type of a given function.

subtyping in the elimination rules [$\text{App}_{\mathcal{A}}$,$\text{Proj}_{\mathcal{A}}$]. In particular, for [$\text{App}_{\mathcal{A}}$] notice that it checks that the type of the function is a functional type, that the type of the argument is a subtype of the domain of the function, and then returns the result type of the application of the two types. The intersection rule is replaced by the use of type schemes in [$\text{Abs}_{\mathcal{A}}$] and by the rule [$\text{Env}_{\mathcal{A}}$]. The latter intersects the type deduced for an expression e by occurrence typing and stored in Γ with the type deduced for e by the logical rules: this is simply obtained by removing any hypothesis about e from Γ, so that the deduction of the type $\mathbb{t}$ for e cannot but end by a logical rule. Of course this does not apply when the expression e is a variable, since an hypothesis in Γ is the only way to deduce the type of a variable, which is why the algorithm reintroduces the classic rule for variables.

The algorithmic system above is sound with respect to the deductive one of Section 2.4

Theorem 2.7 (Soundness). *For every Γ, e, t, n_o, if $\Gamma \vdash_{\mathcal{A}} e : \mathbb{t}$, then $\Gamma \vdash e : t$ for every $t \in \mathbb{t}$.*

We were not able to prove full completeness, just a partial form of it. As anticipated, the problems are twofold: (*i*) the recursive nature of rule [Path] and (*ii*) the use of nested [PAppL] that yield a precision that the algorithm loses by applying Repr() in the definition of Constr (case (19) is the critical one). Completeness is recovered by (*i*) limiting the depth of the derivations and (*ii*) forbidding nested negated arrows on the left-hand side of negated arrows.

Definition 2.8 (Rank-0 negation). A derivation of $\Gamma \vdash e : t$ is *rank-0 negated* if [Abs–] never occurs in the derivation of a left premise of a [PAppL] rule.

The use of this terminology is borrowed from the ranking of higher-order types, since, intuitively, it corresponds to typing a language in which in the types used in dynamic tests, a negated arrow never occurs on the left-hand side of another negated arrow.

Theorem 2.9 (Partial Completeness). *For every Γ, e, t, if $\Gamma \vdash e : t$ is derivable by a rank-0 negated derivation, then there exists n_o such that $\Gamma \vdash_{\mathcal{A}} e : t'$ and $t' \leq t$.*

The use of type schemes and of possibly diverging iterations yields a system that may seem overly complicated. But it is important to stress that this system is defined only to study the type inference system of Section 2.4 and in particular to probe how close we can get to a complete algorithm for it. But for practical applications type schemes are not needed, since they are necessary only when type cases may specify types with negative arrows and this, in practice, never happens (see Footnote 9 and Corollary A.30). This is why for our implementation we use the CDuce library in which type schemes are absent and functions are typed only by intersections of positive arrows. We present the implementation in Section 4 but before let us study some extensions.

3 EXTENSIONS

3.1 Adding structured types

The previous analysis already covers a large gamut of realistic cases. For instance, the analysis already handles list data structures, since products and recursive types can encode them as right associative nested pairs, as it is done in the language CDuce [Benzaken et al. 2003] (e.g., $X = \mathsf{Nil} \vee (\mathsf{Int} \times X)$ is the type of the lists of integers): see Code 8 in Table 1 of Section 4 for a concrete example. And even more since the presence of union types makes it possible to type heterogeneous lists whose content is described by regular expressions on types as proposed by Hosoya et al. [2000]. Since the main application of occurrence typing is to type dynamic languages, then it is worth showing how to extend our work to records. We use the record types as they are defined in CDuce and which are obtained by extending types with the following two type constructors:

$$\textbf{Types}\ t \ ::= \ \{\ell_1 = t \ldots \ell_n = t,\ \_ = t\} \mid \mathsf{Undef}$$

where ℓ ranges over an infinite set of labels Labels and Undef is a special singleton type whose

only value is the constant `undef` which is a constant not in $\mathbb{1}$. The type $\{\ell_1 = t_1 \ldots \ell_n = t_n,\ \_ = t\}$ is a *quasi-constant function* that maps every ℓ_i to the type t_i and every other $\ell \in \mathsf{Labels}$ to the type t (all the ℓ_i's must be distinct). Quasi constant functions are the internal representation of record types in CDuce. These are not visible to the programmer who can use only two specific forms of quasi constant functions, open record types and closed record types, provided by the following syntactic sugar and that form the *record types* of our language[10]

- $\mathbf{\{}\ell_1 = t_1, \ldots, \ell_n = t_n\mathbf{\}}$ for $\{\ell_1 = t_1 \ldots \ell_n = t_n,\ \_ = \mathsf{Undef}\}$ (closed records).
- $\mathbf{\{}\ell_1 = t_1, \ldots, \ell_n = t_n\ \mathbf{..\}}$ for $\{\ell_1 = t_1 \ldots \ell_n = t_n,\ \_ = \mathbb{1} \vee \mathsf{Undef}\}$ (open records).

plus the notation $\ell\!=\!?t$ to denote optional fields, which corresponds to using in the quasi-constant function notation the field $\ell = t \vee \mathsf{Undef}$.

For what concerns expressions, we adapt CDuce records to our analysis. In particular records are built starting from the empty record expression `{}` and by adding, updating, or removing fields:

$$\textbf{Expr}\quad e \ ::=\ \ \{\} \mid \{e\ \texttt{with}\ \ell = e\} \mid e\backslash\ell \mid e.\ell$$

in particular $e\backslash\ell$ deletes the field ℓ from e, $\{e\ \texttt{with}\ \ell = e\}'$ adds the field $\ell = e'$ to the record e (deleting any existing ℓ field), while $e.\ell$ is field selection with the reduction: $\{\ldots, \ell = e, \ldots\}.\ell \rightsquigarrow e$.

To define record type subtyping and record expression type inference we need the following operators on record types (refer to Frisch [2004] for more details):

$$t.\ell \quad = \quad \begin{cases} \min\{u \mid t \leq \mathbf{\{}\ell = u\ \mathbf{..\}}\} & \text{if } t \leq \mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{26}$$

$$t_1 + t_2 \quad = \quad \min\left\{u \;\middle|\; \forall \ell \in \mathsf{Labels}. \begin{cases} u.\ell \geq t_2.\ell & \text{if } t_2.\ell \leq \neg\mathsf{Undef} \\ u.\ell \geq t_1.\ell \vee (t_2.\ell \setminus \mathsf{Undef}) & \text{otherwise} \end{cases}\right\} \tag{27}$$

$$t\backslash\ell \quad = \quad \min\left\{u \;\middle|\; \forall \ell' \in \mathsf{Labels}. \begin{cases} u.\ell' \geq \mathsf{Undef} & \text{if } \ell' = \ell \\ u.\ell' \geq t.\ell' & \text{otherwise} \end{cases}\right\} \tag{28}$$

Then two record types t_1 and t_2 are in subtyping relation, $t_1 \leq t_2$, if and only if for all $\ell \in \mathsf{Labels}$ we have $t_1.\ell \leq t_2.\ell$. In particular $\mathbf{\{..\}}$ is the largest record type.

Expressions are then typed by the following rules (already in algorithmic form).

$$[\textsc{Record}]\ \frac{}{\Gamma \vdash \{\} : \mathbf{\{\}}} \qquad [\textsc{Update}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad t_1 \leq \mathbf{\{..\}} \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \{e_1\ \texttt{with}\ \ell = e_2\} : t_1 + \mathbf{\{}\ell = t_2\mathbf{\}}}\ \{e_1\,\texttt{with}\,\ell = e_2\} \notin \mathsf{dom}(\Gamma)$$

$$[\textsc{Delete}]\ \frac{\Gamma \vdash e : t \qquad t \leq \mathbf{\{..\}}}{\Gamma \vdash e\backslash\ell : t\backslash\ell}\ e\backslash\ell \notin \mathsf{dom}(\Gamma) \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e : t \qquad t \leq \mathbf{\{}\ell = \mathbb{1}\ \mathbf{..\}}}{\Gamma \vdash e.\ell : t.\ell}\ e.\ell \notin \mathsf{dom}(\Gamma)$$

To extend occurrence typing to records we add the following values to paths: $\varpi \in \{\ldots, a_\ell, u^1_\ell, u^2_\ell, r_\ell\}^*$, with $e.\ell \downarrow a_\ell.\varpi = e{\downarrow}\varpi$, $e\backslash\ell \downarrow r_\ell.\varpi = e{\downarrow}\varpi$, and $\{e_1\ \texttt{with}\ \ell = e_2\} \downarrow u^i_\ell.\varpi = e_i{\downarrow}\varpi$ and add the following rules for the new paths:

$$[\textsc{PSel}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.a_\ell : \mathbf{\{}\ell : t'\ \mathbf{..\}}} \qquad [\textsc{PDel}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.r_\ell : (t'\backslash\ell) + \mathbf{\{}\ell =?\,\mathbb{1}\mathbf{\}}}$$

$$[\textsc{PUpd1}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.u^1_\ell : (t'\backslash\ell) + \mathbf{\{}\ell =?\,\mathbb{1}\mathbf{\}}} \qquad [\textsc{PUpd2}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.u^2_\ell : t.\ell'}$$

Deriving the algorithm from these rules is then straightforward:

[10]Note that in the definitions "..." is meta-syntax to denote the presence of other fields while in the open records "**..**" is the syntax that distinguishes them from closed ones.

$\mathsf{Constr}_{\Gamma,e,t}(\varpi.a_\ell) = \{\ell : \mathsf{Intertype}_{\Gamma,e,t}(\varpi)\ ..\}$ $\quad \mathsf{Constr}_{\Gamma,e,t}(\varpi.r_\ell) = (\mathsf{Intertype}_{\Gamma,e,t}(\varpi))\backslash\ell + \{\ell =? \mathbb{1}\}$
$\mathsf{Constr}_{\Gamma,e,t}(\varpi.u^2_\ell) = (\mathsf{Intertype}_{\Gamma,e,t}(\varpi)).\ell$ $\quad \mathsf{Constr}_{\Gamma,e,t}(\varpi.u^1_\ell) = (\mathsf{Intertype}_{\Gamma,e,t}(\varpi))\backslash\ell + \{\ell =? \mathbb{1}\}$

Notice that the effect of doing $t\backslash\ell + \{\ell =? \mathbb{1}\}$ corresponds to setting the field ℓ of the (record) type t to the type $\mathbb{1} \vee \mathsf{Undef}$, that is, to the type of all undefined fields in an open record. So [PDel] and [PUpd1] mean that if we remove, add, or redefine a field ℓ in an expression e then all we can deduce for e is that its field ℓ is undefined: since the original field was destroyed we do not have any information on it apart from the static one. For instance, consider the test:

$$(\{x \texttt{ with } a = 0\} \in \{a = \mathsf{Int}, b = \mathsf{Bool}\ ..\} \vee \{a = \mathsf{Bool}, b = \mathsf{Int}\ ..\})\,?x.b : \mathsf{False}$$

By $\mathsf{Constr}_{\Gamma,e,t}(\varpi.u^1_\ell)$—i.e., by [Ext1], [PTypeof], and [PInter]—the type for x in the positive branch is $((\{a = \mathsf{Int}, b = \mathsf{Bool}\ ..\} \vee \{a = \mathsf{Bool}, b = \mathsf{Int}\ ..\}) \wedge \{a = \mathsf{Int}\ ..\}) + \{a =? \mathbb{1}\}$. It is equivalent to the type $\{b = \mathsf{Bool}\ ..\}$, and thus we can deduce that $x.b$ has the type Bool.

3.2 Refining function types

As we explained in the introduction, both TypeScript and Flow deduce the type `(number∨string) → (number∨string)` for the first definition of the function `foo` in (1), and the more precise type

$$\texttt{(number→number)} \wedge \texttt{(string→string)} \tag{29}$$

can be deduced by these languages only if they are instructed to do so: the programmer has to explicitly annotate `foo` with the type (29): we did it in (2) using Flow—the TypeScript annotation for it is much hevier. But this seems like overkill, since a simple analysis of the body of `foo` in (1) shows that its execution may have two possible behaviors according to whether the parameter `x` has type `number` or not (i.e., or `(number∨string)\number`, that is `string`), and this is should be enough for the system to deduce the type (29) even in the absence the annotation given in (2). In this section we show how to do it by using the theory of occurrence typing we developed in the first part of the paper. In particular, we collect the different types that are assigned to the parameter of a function in its body, and use this information to partition the domain of the function and to re-type its body. Consider a more involved example

```
function (x : τ) {
 (x ∈ Real) ? { (x ∈ Int) ? succ x : sqrt x } : { ¬x }
}
```
(30)

When τ is `Real|Bool` (we assume that `Int` is a subtype of `Real`) we want to deduce for this function the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Real}\backslash\mathsf{Int} \to \mathsf{Real}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$. When τ is $\mathbb{1}$, then the function must be rejected (since it tries to type $\neg$`x` under the assumption that `x` has type $\neg\mathsf{Real}$. Notice that typing the function under the hypothesis that τ is $\mathbb{1}$, allows us to capture user-defined discrimination as defined by Tobin-Hochstadt and Felleisen [2010] since, for instance

```
let is_int x = (x∈Int)? true : false
 in if is_int z then z+1 else 42
```

is well typed since the function `is_int` is given type $(\mathsf{Int} \to \mathsf{True}) \wedge (\neg\mathsf{Int} \to \mathsf{False})$. We choose a more general approach allowing the programmer to hint a particular type for the argument and deducing, if possible, an intersection type for the function.

We start by considering the system where λ-abstractions are typed by a single arrow and later generalize it to the case of intersections of arrows. First, we define the auxiliary judgement $\Gamma \vdash e \triangleright \psi$ where Γ is a typing environement, e an expression and ψ a mapping from variables to sets of types. Intuitively $\psi(x)$ denotes the set that contains the types of all the occurrences of x in e. This judgement can be deduced by the following deduction system that collects type information on the variables that are λ-abstracted (i.e., those in the domain of Γ, since lambdas are our only binders):

$$[\textsc{Var}]\ \frac{}{\Gamma \vdash x \triangleright \{x \mapsto \{\Gamma(x)\}\}} \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c \triangleright \varnothing} \qquad [\textsc{Abs}]\ \frac{\Gamma, x : s \vdash e \triangleright \psi}{\Gamma \vdash \lambda x : s.e \triangleright \psi \setminus \{x\}}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 \triangleright \psi_1 \qquad \Gamma \vdash e_2 \triangleright \psi_2}{\Gamma \vdash e_1 e_2 \triangleright \psi_1 \cup \psi_2} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 \triangleright \psi_1 \qquad \Gamma \vdash e_2 \triangleright \psi_2}{\Gamma \vdash (e_1, e_2) \triangleright \psi_1 \cup \psi_2} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e \triangleright \psi}{\Gamma \vdash \pi_i e \triangleright \psi}$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash e \triangleright \psi_\circ \qquad \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_1 \qquad \Gamma_1 \vdash e_1 \triangleright \psi_1 \qquad \Gamma \vdash^{\mathsf{Env}}_{e,\neg t} \Gamma_2 \qquad \Gamma_2 \vdash e_2 \triangleright \psi_2}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 \triangleright \psi_\circ \cup \psi_1 \cup \psi_2}$$

Where $\psi \setminus \{x\}$ is the function defined as ψ but undefined on x and $\psi_1 \cup \psi_2$ denotes component-wise union, that is :

$$(\psi_1 \cup \psi_2)(x) = \begin{cases} \psi_1(x) & \text{if } x \notin \mathrm{dom}(\psi_2) \\ \psi_2(x) & \text{if } x \notin \mathrm{dom}(\psi_1) \\ \psi_1(x) \cup \psi_2(x) & \text{otherwise} \end{cases}$$

All that remains to do is replace the rule [Abs+] with the following rule

$$[\textsc{AbsInf+}]\ \frac{\Gamma, x : s \vdash e \triangleright \psi \qquad \Gamma, x : s \vdash e : t \qquad T = \{(s,t)\} \cup \{(u,w) \mid u \in \psi(x) \wedge \Gamma, x : u \vdash e : w\}}{\Gamma \vdash \lambda x : s.e : \bigwedge_{(u,w)\in T} u \to w}$$

Note the invariant that the domain of ψ is always equal to the domain of Γ restricted to variables. Simply put, this rule first collects all possible types that are deduced for a variable x during typing and then uses them to re-type the body of the lambda under this new refined hypothesis for the type of x. The re-typing ensures that the type safety property carries over to this new rule.

This system is enough to type our case study (30) for the case τ defined as `Real|Bool`. Indeed the analysis of the body yields $\psi(x) = \{\mathsf{Int}, \mathsf{Real} \setminus \mathsf{Int}\}$ for the branch `(x∈Int) ? succ x : sqrt x` and, since $(\mathsf{Bool} \vee \mathsf{Real}) \setminus \mathsf{Real} = \mathsf{Bool}$, yields $\psi(x) = \{\mathsf{Bool}\}$ for the branch `¬x`. So the function will be checked for the input types Int, Real \ Int, and Bool, yielding the expected result.

It is not too difficult to generalize this rule when the lambda is typed by an intersection type:

$$[\textsc{AbsInf+}]\ \frac{\forall i \in I\ \Gamma, x : s_i \vdash e \triangleright \psi_i \qquad \Gamma, x : s_i \vdash e : t_i \qquad T_i = \{(u,w) \mid u \in \psi_i(x) \wedge \Gamma, x : u \vdash e : w\}}{\Gamma \vdash \lambda^{\bigwedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I}(s_i \to t_i) \wedge \bigwedge_{(u,w)\in T_i}(u \to w)}$$

Here, for each arrow declared in the interface of the function, we first typecheck the body of the function as usual (to check that the arrow is valid) and collect the refined types for the parameter x. Then we deduce all possible output types for this refined input types and add the resulting arrows to the type we deduce for the whole function (see Appendix B for an even more precise rule).

In summary, in order to type a function we use the type-cases on its parameter to partition the domain of the function and we type-check the function on each single partitions rather than on the union thereof. Of course, we could use a much finer partition: the finest (but impossible) one is to check the function against the singleton types of all its inputs. But any finer partition would return, in many cases, not a much better information, since most partitions would collapse on the same return type: type-cases on the parameter are the tipping points that are likely to make a difference, by returning different types for different partitions thus yielding more precise typing. But they are not the only such tipping points: see rule [OverApp] in Section 4.

3.3 Integrating gradual typing

Gradual typing is an approach proposed by Siek and Taha [2006] to combine the safety guarantees of static typing with the programming flexibility of dynamic typing. The idea is to introduce an *unknown* (or *dynamic*) type, denoted $?$, used to inform the compiler that some static type-checking can be omitted, at the cost of some additional runtime checks. The use of both static typing and

dynamic typing in a same program creates a boundary between the two, where the compiler automatically adds—often costly [Takikawa et al. 2016]—dynamic type-checks to ensure that a value crossing the barrier is correctly typed.

Occurrence typing and gradual typing are two complementary disciplines which have a lot to gain to be integrated, although we are not aware of any study in this sense. We study it for the formalism of Section 2 for which the integration of gradual typing was defined by Castagna et al. [2019]. In a sense, occurrence typing is a discipline designed to push forward the frontiers beyond which gradual typing is needed, thus reducing the amount of runtime checks needed. For instance, the the JavaScript code of (1) and (2) in the introduction can be typed by using gradual typing:

```
function foo(x : ?) {
  (typeof(x) === "number")? x++ : x.trim()
}
```
(31)

"Standard" gradual typing inserts two dynamic checks since it compiles the code above into:

```
function foo(x) {
  (typeof(x) === "number")? (x⟨number⟩)++ : (x⟨string⟩).trim()
}
```

where $e\langle t\rangle$ is a type-cast that dynamically checks whether the value returned by e has type t.[11] We already saw that thanks to occurrence typing we can annotate the parameter `x` by `number|string` instead of `?` and avoid the insertion of any cast. But occurrence typing can be used also on the gradually typed code in order to statically detect the insertion of useless casts. Using occurrence typing to type the gradually-typed version of `foo` in (31), allows the system to avoid inserting the first cast `x⟨number⟩` since, thanks to occurrence typing, the occurrence of `x` at issue is given type `number` (the second cast is still necessary however). But removing only this cast is far from being satisfactory, since when this function is applied to an integer there are some casts that still need to be inserted outside of the function. The reason is that the compiled version of the function has type `?→number`, that is, it expects an argument of type `?`, and thus we have to apply a cast (either to the argument or to the function) whenever this is not the case. In particular, the application `foo(42)` will be compiled as `foo(42⟨?⟩)`. Now, the main problem with such a cast is not that it produces some unnecessary overhead by performing useless checks (a cast to `?` can easily be detected and safely ignored at runtime). The main problem is that the combination of such a cast with type-cases will lead to unintuitive results under the standard operational semantics of type-cases and casts. Indeed, consider the standard semantics of the type-case `(typeof(e)==="t")` which consists in reducing e to a value and checking whether the type of the value is a subtype of t. In standard gradual semantics, `42⟨?⟩` is a value. And this value is of type `?`, which is not a subtype of `number`. Therefore the check in `foo` would fail for `42⟨?⟩`, and so would the whole function call. Although this behavior is type safe, this is the opposite of what every programmer would expect: one would expect the test `(typeof(e)==="number")` to return true for `42⟨?⟩` and false for, say, `true⟨?⟩`, whereas the standard semantics of type-cases would return false in both cases.

A solution is to modify the semantics of type-cases, and in particular of `typeof`, to strip off all the casts in a value, even nested ones. This however adds a new overhead at runtime. Another solution is to simply accept this counter-intuitive result, which has at least the benefit of promoting the dynamic type to a first class type, instead of just considering it as a directive to the front-end. Indeed, this would allow to dynamically check whether some argument has the dynamic type `?` (i.e., whether it was applied to a cast to such a type) simply by `(typeof(e)==="?")`. Whatever solution we choose it is clear that in both cases it would be much better if the application `foo(42)`

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

were compiled as is, thus getting rid of a cast that at best is useless and at worse gives a counter-intuitive and unexpected semantics.

This is where the previous section about refining function types comes in handy. To get rid of all superfluous casts, we have to fully exploit the information provided to us by occurrence typing and deduce for the function in (31) the type $(\texttt{number}\to\texttt{number})\wedge((?\backslash\texttt{number})\to\texttt{string})$, so that no cast is inserted when the function is applied to a number. To achieve this, we simply modify the typing rule for functions that we defined in the previous section to accommodate for gradual typing. Let σ and τ range over *gradual types*, that is the types produced by the grammar in Definition 2.1 to which we add $?$ as basic type (see Castagna et al. [2019] for the definition of the subtyping relation on these types). For every gradual type τ, define $\tau^{\Uparrow}$ as the (non graudal) type obtained from τ by replacing all covariant occurrences of $?$ by $\mathbb{1}$ and all contravariant ones by $\mathbb{0}$. The type $\tau^{\Uparrow}$ can be seen as the *maximal* interpretation of τ, that is, every expression that can safely be cast to τ is of type $\tau^{\Uparrow}$. In other words, if a function expects an argument of type τ but can be typed under the hypothesis that the argument has type $\tau^{\Uparrow}$, then no casts are needed, since every cast that succeeds will be a subtype of $\tau^{\Uparrow}$. Taking advantage of this property, we modify the rule for functions as:

$$[\textsc{AbsInf+}]\ \frac{T = \{(\sigma',\tau')\}\cup\{(\sigma,\tau)\mid \sigma\in\psi(x)\wedge\Gamma,x:\sigma\vdash e:\tau\}\cup\{(\sigma^{\Uparrow},\tau)\mid \sigma\in\psi(x)\wedge\Gamma,x:\sigma^{\Uparrow}\vdash e:\tau\} \qquad \Gamma,x:\sigma'\vdash e\triangleright\psi \qquad \Gamma,x:\sigma'\vdash e:\tau'}{\Gamma\vdash\lambda x:\sigma'.e:\bigwedge_{(\sigma,\tau)\in T}\sigma\to\tau}$$

The main idea behind this rule is the same as before: we first collect all the information we can into ψ by analyzing the body of the function. We then retype the function using the new hypothesis $x:\sigma$ for every $\sigma\in\psi(x)$. Furthermore, we also retype the function using the hypothesis $x:\sigma^{\Uparrow}$: as explained before the rule, whenever this typing suceeds it eliminates unnecessary gradual types and, thus, unecessary casts. Let us see how this works on the function `foo` in (31). First, we deduce the refined hypothesis $\psi(\texttt{x}) = \{\,\texttt{number}\wedge?\ ,\ ?\backslash\texttt{number}\,\}$. Typing the function using this new hypothesis but without considering the maximal interpretation would yield $(?\to\texttt{number}\vee\texttt{string})\wedge((\texttt{number}\wedge?)\to\texttt{number})\wedge((?\backslash\texttt{number})\to\texttt{string})$. However, as we stated before, this would introduce an unnecessary cast if the function were to be applied to an integer. Hence the need for the second part of Rule [AbsInf+]: the maximal interpretation of $\texttt{number}\wedge?$ is `number`, and it is clear that, if `x` is given type `number`, the function type-checks, thanks to occurrence typing. Thus, after some routine simplifications, we can actually deduce the desired type $(\texttt{number}\to\texttt{number})\wedge((?\backslash\texttt{number})\to\texttt{string})$.

4 TOWARDS A PRACTICAL IMPLEMENTATION

We have implemented a simplified version of the algorithm presented in Section 2.5 that does not make use of type schemes and is, therefore, incomplete w.r.t. the system of Section 2.4. In particular, as we explained in Section 2.5, in the absence of type schemes it is not always possible to prove that $\forall v,\forall t, v\in t$ or $v\notin\neg t$. Since this property ceases to hold only for λ-abstractions, then not using type schemes yields less precise typing only for tests $(e\in t)\,?e_1:e_2$ where e has a functional type, that is, the value tested will be a λ-abstraction. This seems like a reasonable compromise between the complexity of an implementation involving type schemes and the programs we want to type-check in practice. Indeed, if we restrict the language so that the only functional type t allowed in a test $(e\in t)\,?e_1:e_2$ is $\mathbb{0}\to\mathbb{1}$—i.e., if we allow to check whether a value is a function but not whether it has a specific function type (*cf.*, Footnote 9)—, then our implementation becomes complete (see Corollary A.30 in the appendix for a formal proof).

Our implementation is written in OCaml and uses CDuce as a library to provide the semantic subtyping machinery. Besides a type-checking algorithm defined on the base language, our implementation supports record types (Section 3.1) and the refinement of function types (Section

	Code	Inferred type
1	`let basic_inf = fun (y : Int | Bool) ->` `  if y is Int then incr y else lnot y`	(Int → Int) ∧ (Bool → Bool)

2	`let any_inf = fun (x : Any) ->`
	`  if x is Int then incr x else`
	`  if x is Bool then lnot x else x`
		(Int → Int) ∧ (¬Int → ¬Int)∧
		(Bool → Bool) ∧ (¬(Int ∨ Bool) → ¬(Int ∨ Bool))

3	`let is_int = fun (x : Any) ->`
	`  if x is Int then true else false`
	
	`let is_bool = fun (x : Any) ->`
	`  if x is Bool then true else false`
	
	`let is_char = fun (x : Any) ->`
	`  if x is Char then true else false`
		(Int → True) ∧ (¬Int → False)
		
		(Bool → True) ∧ (¬Bool → False)
		
		(Char → True) ∧ (¬Char → False)

4	`let not_ = fun (x : Any) ->` `  if x is True then false else true`	(True → False) ∧ (¬True → True)

5	`let or_ = fun (x : Any) -> fun (y: Any) ->`
	`  if x is True then true`
	`  else if y is True then true else false`
		(True → 𝟙 → True) ∧ (𝟙 → True → True)∧
		(¬True → ¬True → False)

6	`let and_ = fun (x : Any) -> fun (y : Any) ->`
	`  if not_ (or_ (not_ x) (not_ y)) is True`
	`  then true else false`
		(True → ((¬True → False) ∧ (True → True))
		∧(¬True → 𝟙 → False)

7	`let f = fun (x : Any) -> fun (y : Any) ->`
	`  if and_ (is_int x) (is_bool y) is True`
	`  then 1 else`
	`  if or_ (is_char x) (is_int y) is True`
	`  then 2 else 3`
	
	`let test_1 = f 3 true`
	`let test_2 = f (42,42) 42`
	`let test_3 = f nil nil`
		(Int → (Int → 2) ∧ (¬Int → 1 ∨ 3) ∧ (Bool → 1)∧
		(¬(Bool ∨ Int) → 3) ∧ (¬Bool → 2 ∨ 3)) ∧
		(Char → (Int → 2) ∧ (¬Int → 2) ∧ (Bool → 2)∧
		(¬(Bool ∨ Int) → 2) ∧ (¬Bool → 2)) ∧
		(¬(Int ∨ Char) → (Int → 2) ∧ (¬Int → 3)∧
		(Bool → 3) ∧ (¬(Bool ∨ Int) → 3) ∧ (¬Bool → 2 ∨ 3)) ∧ . . .
		(two other redundant cases ommitted)
		1
		2
		3

8	`type Document = { nodeType=9 ..}`
	`and Element  = { childNodes=NodeList,`
	`  nodeType=1 ..}`
	`and Text = { isElementContentWhiteSpace=Bool,`
	`  nodeType=3, ..}`
	`and Node = Document | Element | Text`
	`and NodeList = Nil | (Node, NodeList)`
	
	`let is_empty_node = fun (x : Node) ->`
	`  if x.nodeType is 9 then false`
	`  else if x.nodeType is 3 then`
	`  x.isElementContentWhiteSpace`
	`  else`
	`  if x.childNodes is Nil then true else false`
		(Document → False) ∧
		({nodeType = 1, childNodes = Nil ••} → True) ∧
		({nodeType = 1, childNodes = (Node, NodeList) ••} → False) ∧
		(Text → Bool) ∧
		(omitted redundant arrows)

9	`let xor_ = fun (x : Any) -> fun (y : Any) ->`
	`  if and_ (or_ x y) (not_ (and_ x y)) is True`
	`  then true else false`
		True → ((True → False) ∧ (¬True → True)) ∧
		(¬True → ((True → True) ∧ (¬True → False))

10	`(* f, g have type: (Int->Int) & (Any->Bool) *)`
	
	`let example10 = fun (x : Any) ->`
	`  if (f x, g x) is (Int, Bool) then 1 else 2`
		(Int → Empty) ∧ (¬Int → 2)
		`Warning: line 4, 39-40: unreachable expression`

Table 1. Types inferred by implementation

3.2 with the rule of Appendix B). The implementation is rather crude and consist of 2000 lines of OCaml code, including parsing, type-checking of programs, and pretty printing of types. We demonstrate the output of our type-checking implementation in Table 1. These examples and others can be tested in the online toplevel available at https://occtyping.github.io/. In this table, the second column gives a code fragment and the third column the type deduced by our implementation. Code 1 is a straightforward function similar to our introductory example `foo` in (1,2). Here the programmer annotates the parameter of the function with a coarse type Int ∨ Bool. Our implementation first type-checks the body of the function under this assumption, but doing so collects that the type of `x` is specialized to Int in the "then" case and to Bool in the "else" case. The function is thus type-checked twice more under each hypothesis for `x`, yielding the precise type (Int → Int) ∧ (Bool → Bool). Note that w.r.t. rule [AbsInf+] of Section 3.2, we improved the output of the computed type. Indeed using rule [AbsInf+] we would obtain the type

(Int → Int) ∧ (Bool → Bool) ∧ (Bool ∨ Int → Bool ∨ Int) with a redundant arrow. Here we can see that since we deduced the first two arrows (Int → Int) ∧ (Bool → Bool), and since the union of their domain exactly covers the domain of the third arrow, the latter is not needed. Code 2 shows what happens when the argument of the function is left unannotated (i.e., it is annotated by the top type 𝟙, written `Any` in our implementation). Here type-checking and refinement also work as expected, but the function only type checks if all cases for `x` are covered (which means that the function must handle the case of inputs that are neither in Int nor in Bool).

The following examples paint a more interesting picture. First (Code 3) it is easy in our formalism to program type predicates such as those hard-coded in the λ_{TR} language of Tobin-Hochstadt and Felleisen [2010]. Such type predicates, which return `true` if and only if their input has a particular type, are just plain functions with an intersection type inferred by the system of Section 3.2. We next define Boolean connectives as overloaded functions. The `not_` connective (Code 4) just tests whether its argument is the Boolean `true` by testing that it belongs to the singleton type True (the type whose only value is `true`) returning `false` for it and `true` for any other value (recall that ¬True is equivalent to `Any\True`). It works on values of any type, but we could restrict it to Boolean values by simply annotating the parameter by Bool (which in CDuce is syntactic sugar for True∨False) yielding the type (True→False) ∧ (False→True). The `or_` connective (Code 5) is straightforward as far as the code goes, but we see that the overloaded type precisely captures all possible cases. Again we use a generalized version of the `or_` connective that accepts and treats any value that is not `true` as `false` and again, we could easily restrict the domain to Bool if desired.

To showcase the power of our type system, and in particular of the " ▪ " type operator, we define `and_` (Code 6) using De Morgan's Laws instead of using a direct definition. Here the application of the outermost `not_` operator is checked against type True. This allows the system to deduce that the whole `or_` application has type False, which in turn leads to `not_ x` and `not_ y` to have type ¬True and therefore both `x` and `y` to have type True. The whole function is typed with the most precise type (we present the type as printed by our implementation, but the first arrow of the resulting type is equivalent to (True → ¬True → False) ∧ (True → True → True)).

All these type predicates and Boolean connectives can be used together to write complex type tests, as in Code 7. Here we define a function `f` that takes two arguments `x` and `y`. If `x` is an integer and `y` a Boolean, then it returns the integer `1`; if `x` is a character or `y` is an integer, then it returns `2`; otherwise the function returns `3`. Our system correctly deduces a (complex) intersection type that covers all cases (plus several redundant arrow types). That this type is as precise as possible can be shown by the fact that when applying `f` to arguments of the expected type, the *type* deduced for the whole expression is the singleton type `1`, or `2`, or `3`, depending on the type of the arguments.

Code 8 allows us to demonstrate the use and typing of record paths. We model, using open records, the type of DOM objects that represent XML or HTML documents. Such objects possess a common field `nodeType` containing an integer constant denoting the kind of the node (e.g., `9` for the root element, `1` for an element node, `3` for a text node, …). Depending on the kind, the object will have different fields and methods. It is common practice to perform a test on the value of the `nodeType` field. In dynamic languages such as JavaScript, the relevant field or method can directly be accessed after having checked for the appropriate `nodeType`. In mainstream statically typed languages, such as Java, a downward cast from the generic `Node` type to the expected precise type of the object is needed. We can see that using the extension presented in Section 3.1 we can deduce the correct type for `x` in all cases. Of particular interest is the last case, since we use a type case to check the emptiness of the list of child nodes. This splits, at the type level, the case for the Element type depending on whether the content of the `childNodes` field is the empty list or not.

Our implementation features another enhancement that allows us to further improve the precision of the inferred type. Consider the definition of the `xor_` operator (Code 9). Here the rule [AbsInf+] is not sufficient to precisely type the function, and using only this rule would yield a type $\mathbb{1} \to \mathbb{1} \to \mathsf{Bool}$. Let us follow the behavior of the " ▪ " operator. Here the whole `and_` is requested to have type True, which implies that `or_ x y` must have type True. This can always happen, whether `x` is True or not (but then depends on the type of `y`). The " ▪ " operator correctly computes that the type for `x` in the "`then`" branch is $\mathsf{True} \vee \neg\mathsf{True} \vee \mathsf{True} \simeq \mathbb{1}$, and a similar reasoning holds for `y`. To solve this problem, we can first remark that even though type cases in the body of a function are tipping points that may change the type of the result of the function, they are not the only ones: applications of overloaded functions play exactly the same role. We therefore extend deduction system for $\Gamma \vdash e \triangleright \psi$ defined in Section 3.2 with the following rule

$$[\textsc{OverApp}]\quad \frac{\Gamma \vdash e : \bigvee \bigwedge_{i\in I} t_i \to s_i \qquad \Gamma \vdash x : t \qquad \Gamma \vdash e \triangleright \psi_1 \qquad \Gamma \vdash x \triangleright \psi_2}{\Gamma \vdash e\ x \triangleright \psi_1 \cup \psi_2 \cup \bigcup_{i\in I}\{x \mapsto t \wedge t_i\}}$$

Whenever a function parameter is the argument of an overloaded function, we record as possible types for this parameter all the possible domains of the arrows that type the overloaded function, restricted by the static type of the parameter. In Code 9, since, `or_` has type

$$(\mathsf{True} \to \mathbb{1} \to \mathsf{True}) \wedge (\mathbb{1} \to \mathsf{True} \to \mathsf{True}) \wedge (\neg\mathsf{True} \to \neg\mathsf{True} \to \mathsf{False})$$

We consider True, $\mathbb{1}$, and $\neg$True as candidate types for `x` which, in turn allows us to deduce a precise type given in the table. Finally, thanks to this rule it is no longer necessary to use a type case to force refinement. As a consequence we can define the functions `and_` and `xor_` more naturally as:

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

for which the very same types as in Table 1 are deduced.

Last but not least Code 10 (corresponding to our introductory example (10)) illustrates the need for iterative refinement of type environments, as defined in Section 2.5.3. As explained, a single pass analysis would deduce for `x` a type Int from the `f x` application and $\mathbb{1}$ from the `g x` application. Here by iterating a second time, the algorithm deduces that `x` has type $\mathbb{0}$ (i.e., Empty), that is that the first branch can never be selected (and our implementation warns the user accordingly). In hindsight, the only way for a well-typed overloaded function to have type $(\mathsf{Int}\to\mathsf{Int}) \wedge (\mathbb{1}\to\mathsf{Bool})$ is to diverge when the argument is of type Int: since this intersection type states that whenever the input is Int, *both* branches can be selected, yielding a result that is at the same time an integer and a Boolean. This is precisely reflected by the case $\mathsf{Int} \to \mathbb{0}$ in the result. Indeed our `example10` function can be applied to an integer, but at runtime the application of `f x` will diverge.

5 RELATED WORK

Occurrence typing was introduced by Tobin-Hochstadt and Felleisen [2008] and further advanced in [Tobin-Hochstadt and Felleisen 2010] in the context of the Typed Racket language. This latter work in particular is close to ours, with some key differences. Tobin-Hochstadt and Felleisen [2010] define λ_{TR}, a core calculus for Typed Racket. In this language types are annotated by 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. Such information is used selectively in the "then" and "else" branches of a test. One area where their work goes further than ours is that the type information also flows outside of the tests to the surrounding context. In contrast, our type system only refines the type of variables strictly in the branches of a test. However, using semantic-subtyping as a foundation we believe our approach has several merits

over theirs. First, in our case, type predicates are not built-in. A user may define any type predicate she wishes by using an overloaded function, as we have shown in Section 4. Second, in our setting, *types* play the role of formulÃę. Using set-theoretic types, we can express the complex types of variables without resorting to a meta-logic. This allows us to type all but two of their key examples (the notable exceptions being Example 8 and 14 in their paper, which use the propagation of type information outside of the branches of a test). Also, while they extend their core calculus with pairs, they only provide a simple `cons?` predicate that allows them to test whether some value is a pair. It is therefore unclear whether their systems allows one to write predicates over list types (e.g., test whether the input is a list of integers), which we can easily do thanks to our support for recursive types.

Kent et al. [2016] introduce the λ_{RTR} core calculus, an extension of λ_{TR} where the logical formulÃę embedded in types are not limited to built-in type predicates, but accept predicates of arbitrary theories. This allows them to provide some form of dependent typing (and in particular they provide an implementation supporting bitvector and linear arithmetic theories). While static invariants that can be enforced by such logic go well beyond what can be proven by a static "non dependent" type system, it does so at the cost of having the programmer write logical annotations (to help the external provers). While this work provides richer logical statements than those by Tobin-Hochstadt and Felleisen [2010], it still remains restricted to refining the types of variables, and not of arbitrary constructs such as pairs, records or recursive types.

Chaudhuri et al. [2017] present the design and implementation of Flow by formalizing a relevant fragment of the language. Since they target an industrial-grade implementation, they must account for aspects that we could afford to postpone to future work, notably side effects and responsiveness of the type checker on very large code base. The degree of precision of their analysis is really impressive and they achieve most of what we did here and, since they perform flow analysis and use an effect system (to track mutable variables), even more. However, this results in a specific and very complex system. Their formalization includes only union types (though, Flow accepts also intersection types as in (2)) which are used in *ad hoc* manner by the type system, for instance to type record types. This allows Flow to perform an analysis similar to the one we did for Code 8 in Table 1, but also has as a consequence that in some cases unions do not behave as expected. In contrast, our approach is more classic and foundational: we really define a type system, typing rules looks like classic ones and are easy to understand, unions are unions of values (and so are intersections and negations), and the algorithmic part is—excepted for fix points—relatively simple (algorithmically Flow relies on constraint generation and solving). This is the reason why our system is more adapted to study and understand occurrence typing and to extend it with additional features (e.g., gradual typing and polymorphism) and we are eager to test how much of their analysis we can capture and enhance by formalizing it in our system.

6 FUTURE WORK AND CONCLUSION

In this work we presented to core of our analysis of occurrence typing, extended it to record types and proposed a couple of novel applications of the theory, namely the inference of intersection types for functions and a static analysis to reduce the number of casts inserted when compiling gradually-typed programs. One of the by-products of our work is the ability to define type predicates such as those used in [Tobin-Hochstadt and Felleisen 2010] as plain functions and have the inference procedure deduce automatically the correct overloaded function type.

There is still a lot of work to do to fill the gap with real-word programming languages. For example, our analysis cannot handle flow of information. In particular, the result of a type test can flow only to the branches but not outside the test. As a consequence the current system cannot type a let binding such as `let x = (y∈Int)?'yes:'no in (x∈'yes)?y+1:not(y)` which is

clearly safe when $y : \mathsf{Int} \vee \mathsf{Bool}$. Nor can this example be solved by partial evaluation since we do not handle nesting of tests in the condition `( ((y∈Int)?'yes:'no)∈'yes) ? y+1 : not(y)`, and both are issues that the system by Tobin-Hochstadt and Felleisen [2010] can handle. We think that it is possible to reuse some of their ideas to perform an information flow analysis on top of our system to remove these limitations. Some of the exensions we hinted to in Section 4 warrant a formal treatment. In particular, the rule [OverApp] only detects the application of an overloaded function once, when type-checking the body of the function against the coarse input type (i.e., ψ is computed only once). But we could repeat this process whilst type-checking the inferred arrows (i.e., we would enrich ψ while using it to find the various arrow types of the lambda abstraction). Clearly, if untamed, such a process may never reach a fix point. Studying whether this iterative refining can be made to converge and, foremost, whether it is of use in practice is among our objectives.

But the real challenges that lie ahead are the handling of side effects and the addition of polymorphic types. Our analysis works in a pure system and extending it to cope with side-effects is not immediate. We plan to do it by defining effect systems—notably, we will try to graft the effect system of Chaudhuri et al. [2017] on ours—and/or by performing some information flow analysis typically by enriching the one we will develop to overcome the limitations above. But our plan is not more defined than that. For polymorphism, instead, we can easily adapt the main idea of this work to the polymorphic setting. Indeed, the main idea is to remove from the type of an expression all the results of the expression that would make some test fail (or succeed, if we are typing a negative branch). This is done by applying an intersection to the type of the expression, so as to keep only the values that may yield success (or failure) of the test. For polymorphism the idea is the same, with the only difference that besides applying an intersection we can also apply an instantiation. The idea is to single out the two most general type substitutions for which some test may succeed and fail, respectively, and apply these substitutions to refine the types of the corresponding occurrences in the "then" and "else" branches. Concretely, consider the test $x_1 x_2 \in t$ where t is a closed type and x_1, x_2 are variables of type $x_1 : s \to t$ and $x_2 : u$ with $u \leq s$. For the positive branch we first check whether there exists a type substitution σ such that $t\sigma \leq \neg\tau$. If it does not exists, then this means that for all possible assignments of polymorphic type variables of $s \to t$, the test may succeed, that is, the success of the test does not depend on the particular instance of $s \to t$ and, thus, it is not possible to pick some substitution for refining the occurrence typing. If it exists, then we find a type substitution $\sigma_\circ$ such that $\tau \leq t\sigma_\circ$ and we refine for the positive branch the types of x_1, of x_2, and of $x_1 x_2$ by applying $\sigma_\circ$ to their types. While the idea is clear (see Appendix C for a more detailed explanation), the technical details are quite involved, especially when considering functions typed by intersection types and/or when integrating gradual typing. This needs a whole gamut of non trivial research that we plan to develop in the near future.

ACKNOWLEDGMENTS

The authors thank Paul-AndrÃĭ MelliÃĺs for his help on type ranking.

This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045- DIGICOSME) operated by ANR as part of the program ÂńInvestissement d'AvenirÂż Idex Paris-Saclay (ANR-11-IDEX-0003-02) and by a Google PhD fellowship for the second author.

REFERENCES

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-Centric General-Purpose Language. In *ICFP '03, 8th ACM International Conference on Functional Programming*. ACM Press, Uppsala, Sweden, 51–63. http://doi.acm.org/10.1145/944746.944711

Giuseppe Castagna. 2019. Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Method in Computer Science* (2019). (Revised edition: first version 2013). To appear. https://arxiv.org/abs/1809.01427.

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

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

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 7 Denis Diderot. http://www.cduce.org/papers/frisch_phd.pdf

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

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2000. Regular Expression Types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP) (SIGPLAN Notices)*, Vol. 35(9).

Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 296–309. https://doi.org/10.1145/2908080.2908091

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

Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, 456–468. https://doi.org/10.1145/2914770.2837630

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 (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 (ICFP '10)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1863543.1863561

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

A FULL PROOFS AND DEFINITIONS

A.1 Full declarative type system

$$[\textsc{Env}]\ \frac{}{\Gamma \vdash e : \Gamma(e)}\ e \in \mathrm{dom}(\Gamma) \qquad [\textsc{Inter}]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\textsc{Subs}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \boldsymbol{b}_c} \qquad [\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$[\textsc{Abs+}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\textsc{Abs-}]\ \frac{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : t}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \neg(t_1 \to t_2)}\ ((\wedge_{i\in I} s_i \to t_i) \wedge \neg(t_1 \to t_2)) \not\simeq \mathbb{0}$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_1 \qquad \Gamma_1 \vdash e_1 : t' \qquad \Gamma \vdash^{\mathsf{Env}}_{e,\neg t} \Gamma_2 \qquad \Gamma_2 \vdash e_2 : t'}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 : t'}$$

$$[\textsc{Efq}]\ \frac{}{\Gamma, (e : \mathbb{0}) \vdash e' : t} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\textsc{Base}]\ \frac{}{\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma} \qquad [\textsc{Path}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma',e,t} \varpi : t' \qquad \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'}{\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma', (e{\downarrow}\varpi : t')}$$

$$[\textsc{PSubs}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \qquad t_1 \leq t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2} \qquad [\textsc{PInter}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \qquad \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \wedge t_2} \qquad [\textsc{PTypeof}]\ \frac{\Gamma \vdash e{\downarrow}\varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}$$

$$[\textsc{PEps}]\ \frac{}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \epsilon : t} \qquad [\textsc{PAppR}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.0 : t_1 \to t_2 \qquad \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.1 : \neg t_1}\ t_2 \wedge t_2' \simeq \mathbb{0}$$

$$[\textsc{PAppL}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.1 : t_1 \qquad \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.0 : \neg(t_1 \to \neg t_2)} \qquad [\textsc{PPairL}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \times t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.l : t_1}$$

$$[\textsc{PPairR}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \times t_2}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.r : t_2} \qquad [\textsc{PFst}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.f : t' \times \mathbb{1}} \qquad [\textsc{PSnd}]\ \frac{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'}{\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.s : \mathbb{1} \times t'}$$

A.2 Full parallel semantics

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

$$\textbf{Context} \quad C[] \quad ::= \quad e[] \mid []v \mid (e,[]) \mid ([],v)$$

For convenience, we denote $e \overset{e\mapsto e'}{\rightsquigarrow} e'$ by $e \overset{Id}{\rightsquigarrow} e'$.

$$[\textsc{Ctx}]\ \frac{e \overset{e_r\mapsto e'_r}{\rightsquigarrow} e'}{C[e] \overset{e_r\mapsto e'_r}{\rightsquigarrow} C[e']} \qquad [\textsc{App}]\ \frac{}{(\lambda^t x.e)v \overset{Id}{\rightsquigarrow} e\{x\mapsto v\}} \qquad [\textsc{Proj}]\ \frac{}{\pi_i(v_1,v_2) \overset{Id}{\rightsquigarrow} v_i}$$

$$[\textsc{TestCtx}]\ \frac{e \overset{e_r\mapsto e'_r}{\rightsquigarrow} e'}{(e{\in}t)\,?\,e_1 : e_2 \overset{Id}{\rightsquigarrow} (e\{e_r\mapsto e'_r\}{\in}t)\,?\,e_1\{e_r\mapsto e'_r\} : e_2\{e_r\mapsto e'_r\}}$$

$$[\textsc{Case}]\ \frac{i\in\{1,2\} \qquad (i=1 \Leftrightarrow v\in[\![t]\!]_{\mathcal{V}})}{(v{\in}t)\,?\,e_1 : e_2 \overset{Id}{\rightsquigarrow} e_i}$$

$$[\![t]\!]_{\mathcal{V}} = \{v \mid \ \vdash_{\mathcal{V}} v : t\}$$

$$[\textsc{Subsum}]\ \frac{\vdash_{\mathcal{V}} v : t' \qquad t'\le t}{\vdash_{\mathcal{V}} v : t} \qquad [\textsc{Const}]\ \frac{}{\vdash_{\mathcal{V}} c : \boldsymbol{b_c}}$$

$$[\textsc{Abs}]\ \frac{t = (\wedge_{i\in I} s_i\to t_i)\wedge(\wedge_{j\in J}\neg(s'_j\to t'_j)) \qquad t\not\simeq\mathbb{0}}{\vdash_{\mathcal{V}} \lambda^{\wedge_{i\in I} s_i\to t_i} x.e : t}$$

A.3 Proofs for the declarative type system

In this section, the substitutions on expressions that we introduce are up to alpha-renaming and perform only one pass. For instance, if our substitution is $\rho = \{(\lambda^t x.x)y \mapsto y\}$, we have $((\lambda^t x.x)((\lambda^t z.z)y))\rho = (\lambda^t x.x)y$.

The environments also operate up to alpha-renaming.

Finally, the only environments that we consider are well-formed environments (see definition below). We can easily check that every derivation only contains well-formed environments, provided that the initial judgement also use a well-formed environment. It is a consequence of the fact that rule [Case] require e to be typeable and that it only refines subexpressions of e.

A.3.1 Environments — definitions.

Definition A.1 (Well-formed environment). We say that an environment Γ is well-formed iff $\forall e \in \mathrm{dom}(\Gamma)$ s.t. e is not a variable. $\exists t.\ \Gamma\setminus\{e\} \vdash e : t$.

In other words, an environment can refine the type of an expression, but only if this expression is already typeable without this entry in the environment (possibly with a strictly weaker type).

Definition A.2 (Bottom environment). Let Γ an environment.
Γ is bottom (noted $\Gamma = \bot$) iff $\exists e \in \mathrm{dom}(\Gamma).\ \Gamma(e) \simeq \mathbb{0}$.

Definition A.3 ((Pre)order on environments). Let Γ and Γ' two environments. We write $\Gamma' \leq \Gamma$ iff:

$$\Gamma' = \bot \text{ or } (\Gamma \neq \bot \text{ and } \forall e \in \mathrm{dom}(\Gamma).\ \Gamma' \vdash e : \Gamma(e))$$

This relation is a preorder (proof below).

Definition A.4 (Application of a substitution to an environment). Let Γ an environment and ρ a substitution from expressions to expressions. The environment $\Gamma\rho$ is defined by:

$$\mathrm{dom}(\Gamma\rho) = \mathrm{dom}(\Gamma)\rho$$
$$\forall e \in \mathrm{dom}(\Gamma\rho), (\Gamma\rho)(e) = \bigwedge_{\{e' \in \mathrm{dom}(\Gamma) \mid e'\rho \equiv e\}} \Gamma(e')$$

Definition A.5 (Ordinary environments). We say that an environment Γ is ordinary iff its domain only contains variables.

A.3.2 Subject Reduction.

Property 1 ($[\![\_]\!]_{\mathcal{V}}$ properties).

$$\forall s.\ \forall t.\ [\![s]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}} \Leftrightarrow s \leq t$$
$$[\![\mathbb{0}]\!]_{\mathcal{V}} = \varnothing$$
$$\forall t.\ [\![\neg t]\!]_{\mathcal{V}} = \mathcal{V} \setminus [\![t]\!]_{\mathcal{V}}$$
$$\forall s.\ \forall t.\ [\![s \vee t]\!]_{\mathcal{V}} = [\![s]\!]_{\mathcal{V}} \cup [\![t]\!]_{\mathcal{V}}$$

Proof. See theorem 5.5, lemmas 6.19, 6.22, 6.23 of [Frisch et al. 2008]. □

Lemma A.6 (Alpha-renaming). *Both the type system and the semantics are invariant by alpha-renaming.*

Proof. Straightforward. For the type system, it is a consequence of the fact that environments are up to alpha-renaming. For the semantics, it is a consequence of the fact that parallel substitutions in [TestCtx] are up to alpha-renaming. □

Lemma A.7 (Completeness and soundness for typing of values). *Let v a value, t a type and Γ an environment.*
If $v \in [\![t]\!]_{\mathcal{V}}$ and v is well-typed in Γ, then $\Gamma \vdash v : t$.
If $\Gamma \vdash v : t$ and $\Gamma \neq \bot$, then $v \in [\![t]\!]_{\mathcal{V}}$.

Proof. Immediate by definition of $[\![.]\!]_{\mathcal{V}}$. □

Lemma A.8 (Monotonicity). *Let Γ and Γ' two environments such that $\Gamma' \leq \Gamma$. Then, we have:*

$$\forall e, t.\ \Gamma \vdash e : t \Rightarrow \Gamma' \vdash e : t$$
$$\forall e, t, \Gamma_1.\ \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_1 \Rightarrow \exists \Gamma_1'.\ \Gamma_1' \leq \Gamma_1.\ \Gamma' \vdash^{\mathsf{Env}}_{e,t} \Gamma_1'$$
$$\forall e, t, \varpi, t'.\ \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t' \Rightarrow \vdash^{\mathsf{Path}}_{\Gamma',e,t} \varpi : t'$$

Proof. Immediate, by replacing every occurrence of rule [Env] in the derivation with Γ by the corresponding derivation with Γ', followed by an application of rule [Subs] if needed. □

Corollary A.9 (Preorder relation). *The relation $\leq$ on environements is a preorder.*

Lemma A.10 (Value refinement 1). *If we have $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.x : t'$ with $x \in \{0, 1, l, r, f, s\}$ (and e well-typed in Γ) such that $\forall y.\ e\downarrow\varpi.y$ is a value and $v = e\downarrow\varpi.x \notin [\![t']\!]_{\mathcal{V}}$, we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.*

Proof. We proceed by induction on the derivation of $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.x : t'$.
We perform a case analysis on the last rule:

[PTypeof] In this case we have $\Gamma \vdash e{\downarrow}\varpi.x : t'$ with $v \notin [\![t']\!]_{\mathcal{V}}$. Thus we can derive $\Gamma \vdash e{\downarrow}\varpi.x : \mathbb{0}$ by using the rule [Inter] and the rules [Abs+], [Abs-] or [Const].
Let's show that we also have $\Gamma \vdash e{\downarrow}\varpi : \mathbb{0}$.
- If $x = 0$, we know that $e{\downarrow}\varpi$ is an application, and we can conclude easily given that $\mathbb{0} \leq \mathbb{1} \to \mathbb{0}$.
- If $x = 1$, we know that $e{\downarrow}\varpi$ is an application, and we can conclude easily given that $\mathbb{0} \to \mathbb{0} \simeq \mathbb{0} \to \mathbb{1}$.
- If $x = f$ or $x = s$, we know that $e{\downarrow}\varpi$ is a projection, and we can conclude easily given that $\mathbb{0} \simeq \mathbb{0} \times \mathbb{0}$.
- If $x = l$ or $x = r$, we know that $e{\downarrow}\varpi$ is a pair, and we can conclude easily given that $\mathbb{0} \times \mathbb{1} \simeq \mathbb{1} \times \mathbb{0} \simeq \mathbb{0}$.

Hence we can derive $\Gamma \vdash e{\downarrow}\varpi : \mathbb{0}$.

[PInter] We must have $v \notin [\![t_1 \wedge t_2]\!]_{\mathcal{V}}$. It implies $v \notin [\![t_1]\!]_{\mathcal{V}} \cap [\![t_2]\!]_{\mathcal{V}}$ and thus $v \notin [\![t_1]\!]_{\mathcal{V}}$ or $v \notin [\![t_2]\!]_{\mathcal{V}}$. Hence, we can conclude just by applying the induction hypothesis.

[PSubs] Trivial (we use the induction hypothesis).

[PEps] This case is impossible.

[PAppL] We have $v \notin [\![\neg(t_1 \to \neg t_2)]\!]_{\mathcal{V}}$. Thus, we have $v \in [\![t_1 \to \neg t_2]\!]_{\mathcal{V}}$ and in consequence we can derive $\Gamma \vdash v : t_1 \to \neg t_2$ (because e is well-typed in Γ).
Recall that $e{\downarrow}\varpi.1$ is necessarily a value (by hypothesis). By using the induction hypothesis on $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.1 : t_1$, we can suppose $e{\downarrow}\varpi.1 \in [\![t_1]\!]_{\mathcal{V}}$ (otherwise, we can conclude directly). Thus, we can derive $\Gamma \vdash e{\downarrow}\varpi.1 : t_1$.
From $\Gamma \vdash v : t_1 \to \neg t_2$ and $\Gamma \vdash e{\downarrow}\varpi.1 : t_1$, we can derive $\Gamma \vdash e{\downarrow}\varpi : \neg t_2$ using the rule [App].
Now, by starting from the premise $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_2$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[PAppR] We have $v \notin [\![\neg t_1]\!]_{\mathcal{V}}$. Thus, we have $v \in [\![t_1]\!]_{\mathcal{V}}$ and in consequence we can derive $\Gamma \vdash v : t_1$.
Recall that $e{\downarrow}\varpi.0$ is necessarily a value (by hypothesis). By using the induction hypothesis on $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.0 : t_1 \to t_2$, we can suppose $e{\downarrow}\varpi.0 \in [\![t_1 \to t_2]\!]_{\mathcal{V}}$ (otherwise, we can conclude directly). Thus, we can derive $\Gamma \vdash e{\downarrow}\varpi.0 : t_1 \to t_2$ (because e is well-typed in Γ).
From $\Gamma \vdash v : t_1$ and $\Gamma \vdash e{\downarrow}\varpi.0 : t_1 \to t_2$, we can derive $\Gamma \vdash e{\downarrow}\varpi : t_2$ using the rule [App].
Now, by starting from the premise $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'_2$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[PPairL] We have $v \notin [\![t_1]\!]_{\mathcal{V}}$. Thus, we have $v \in [\![\neg t_1]\!]_{\mathcal{V}}$ and in consequence we can derive $\Gamma \vdash v : \neg t_1$.
Hence, we can derive $\Gamma \vdash e{\downarrow}\varpi : \neg t_1 \times \mathbb{1}$ (e is well-typed in Γ).
Now, by starting from the premise $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t_1 \times t_2$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[PPairR] Similar to the previous case.

[PFst] We have $v \notin [\![t' \times \mathbb{1}]\!]_{\mathcal{V}}$. As we also have $v \in [\![\mathbb{1} \times \mathbb{1}]\!]_{\mathcal{V}}$ (because e is well-typed in Γ), we can deduce $v \in [\![(\neg t') \times \mathbb{1}]\!]_{\mathcal{V}}$.
Hence, we can derive $\Gamma \vdash v : (\neg t') \times \mathbb{1}$ and then $\Gamma \vdash e{\downarrow}\varpi : \neg t'$.
Now, by starting from the premise $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[PSnd] Similar to the previous case.

□

Corollary A.11 (Value refinement 2). *For any derivable judgement of the form* $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ *(with* e *well-typed in* Γ*), we can construct a derivation of* $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma''$ *with* $\Gamma'' \leq \Gamma'$ *that never uses the rule [Path] on a path* $\varpi.x$ *such that* $\forall y.\ e\downarrow\varpi.y$ *refers to a value.*

Proof. We can easily remove every such rule from the derivation. If $e\downarrow\varpi.x \in [\![t']\!]_{\mathcal{V}}$, the [Path] rule is useless and we can freely remove it. Otherwise, if $e\downarrow\varpi.x \notin [\![t']\!]_{\mathcal{V}}$, we can use the previous lemma to replace it with a [Path] rule on ϖ. □

Lemma A.12 (Value testing). *For any derivable judgement of the form* $\Gamma \vdash^{\mathsf{Env}}_{v,t} \Gamma'$ *(with* v *a value), we have* $v \in [\![t]\!]_{\mathcal{V}} \Rightarrow \Gamma \leq \Gamma'$.

Proof. As v is a value, the applications of [Path] have a path ϖ only composed of l and r and such that $e\downarrow\varpi$ is a value.

Thus, any derivation $\vdash^{\mathsf{Path}}_{\Gamma,v,t} \varpi : t'$ can only contains the rules [PTypeof], [PInter], [PSubs], [PEps], [PPairL] and [PPairR].

Moreover, as $v \in [\![t]\!]_{\mathcal{V}}$, the rules [PEps] can be replaced by a [PTypeof]. Thus we can easily derive $\Gamma \vdash v : t'$ (we replace [PTypeof] by [Typeof], [PInter] by [Inter], etc.). □

Lemma A.13 (Substitution). *Let* Γ *an environment. Let* e_a *and* e_b *two expressions.*
Let's suppose that e_b *is closed and that* e_a *has one of the following form:*

- x *(variable)*
- $(e \in t)\ ?\ e_1 : e_2$ *(if expression)*
- v *(value)*
- vv *(application of two values)*
- (v, v) *(product of two values)*

Let's also suppose that $\forall t.\ \Gamma \vdash e_a : t \Rightarrow \Gamma\{e_a \mapsto e_b\} \vdash e_b : t$.
Then, by noting $\rho = \{e_a \mapsto e_b\}$ *we have:*

$$\forall e, t.\ \Gamma \vdash e : t \Rightarrow \Gamma\rho \vdash e\rho : t$$

Proof. Let Γ, e_a, e_b as in the statement.

We note ρ the substitution $\{e_a \mapsto e_b\}$.

We consider a derivation of $\Gamma \vdash e : t$.

By using the value refinement lemma, we can assume without loss of generality that our derivation does not contain any rule [Path] on a path $\varpi.x$ such that $\forall y.\ e\downarrow\varpi.y$ refers to a value.

We can also assume w.l.o.g. that every application of the [Path] rule is such that $\Gamma', (e\downarrow\varpi : t') \leq \Gamma'$. If it is not the case, we can easily transform the derivation by intersecting t' with $\Gamma'(e\downarrow\varpi)$ using the rules [PInter], [PTypeof] and [Env]. The rest of the derivation can easily be adapted by adding some [Subs] rules when needed.

Finally, we can assume that, in any environment appearing in the derivation, if the environnement is not bottom, then a value v can only be mapped to a type t such that $v \in [\![t]\!]_{\mathcal{V}}$. If it is not the case, then we just have to change the [Path] rule that introduce $(v : t)$ into a path rule that introduce $(v : \mathbb{0})$, by using the rules [PInter] and [PTypeof] (if $v \notin [\![t]\!]_{\mathcal{V}}$, then $v \in [\![\neg t]\!]_{\mathcal{V}}$ and thus $\Gamma \vdash v : \neg t$ is derivable).

Now, let's prove by induction on the derivation the following properties:

$\forall e, t.\ \Gamma \vdash e : t \Rightarrow \Gamma\rho \vdash e\rho : t$

$\forall e, t, \Gamma'.\ \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma' \Rightarrow \Gamma\rho \vdash^{\mathsf{Env}}_{e\rho,t} \Gamma'\rho$ and we still have $\forall t.\ \Gamma' \vdash e_a : t \Rightarrow \Gamma'\rho \vdash e_b : t$

$\forall e, t, \varpi, t'$ s.t. $e\rho{\downarrow}\varpi$ is defined. $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t' \Rightarrow \vdash^{\mathsf{Path}}_{\Gamma\rho,e\rho,t} \varpi : t'$

We proceed by case analysis on the last rule of the derivation at the left of the $\Rightarrow$ in order to construct the derivation at the right.

If the last judgement is of the form $\Gamma \vdash e_a : t$, then we can directly conclude with the hypotheses of the lemma. Thus, we can suppose it is not the case.

There are many cases depending on the last rule:

[Env] If $e \in \mathrm{dom}(\Gamma)$, then we have $e\rho \in \mathrm{dom}(\Gamma\rho)$ and $(\Gamma\rho)(e\rho) \leq \Gamma(e)$. Thus we can easily derive $\Gamma\rho \vdash e\rho : t$ with the rule [Env] and [Subs].

[Efq] If there exists $e \in \mathrm{dom}(\Gamma)$ such that $\Gamma(e) = \mathbb{0}$, then $(\Gamma\rho)(e\rho) = \mathbb{0}$ so we can easily derive $\Gamma\rho \vdash e\rho : t$ with the rule [Efq].

[Inter] Trivial (by using the induction hypothesis).

[Subs] Trivial (by using the induction hypothesis).

[Const] In this case, $c\rho = c$ (because $c \neq e_a$). Thus it is trivial.

[App] We have $(e_1e_2)\rho = (e_1\rho)(e_2\rho)$ (because $e_1e_2 \neq e_a$). Thus we can directly conclude by using the induction hypothesis.

[Abs+] We have $(\lambda^{t'}x.e)\rho = \lambda^{t'}x.(e\rho)$ (because $\lambda^{t'}x.e \neq e_a$).
By alpha-renaming, we can suppose that the variable x is a new fresh variable that does not appear in e_a nor e_b (e_b is closed).
We can thus use the induction hypothesis on all the judgements $\Gamma, x : s_i \vdash e : t_i$.

[Abs-] Trivial (by using the induction hypothesis).

[Proj] We have $(\pi_i e)\rho = \pi_i(e\rho)$ (because $\pi_i e \neq e_a$). Thus we can directly conclude by using the induction hypothesis.

[Pair] We have $(e_1, e_2)\rho = (e_1\rho, e_2\rho)$ (because $(e_1, e_2) \neq e_a$). Thus we can directly conclude by using the induction hypothesis.

[Case] We have $((e{\in}t_{if})\,?\,e_1 : e_2)\rho = (e\rho{\in}t_{if})\,?\,e_1\rho : e_2\rho$ (because $(e{\in}t_{if})\,?\,e_1 : e_2 \neq e_a$).
We apply the induction hypothesis on the judgements $\Gamma \vdash e : t_0$ and $\Gamma \vdash^{\mathsf{Env}}_{e,t_{if}} \Gamma_1$. We get $\Gamma\rho \vdash e\rho : t_0$, $\Gamma\rho \vdash^{\mathsf{Env}}_{e\rho,t_{if}} \Gamma_1\rho$ and $\forall t'.\ \Gamma_1 \vdash e_a : t' \Rightarrow \Gamma_1\rho \vdash e_b : t'$. Now, we can apply the induction hypothesis on $\Gamma_1 \vdash e_1 : t$ and we have $\Gamma_1\rho \vdash e_1\rho : t$.
We proceed similarly on the judgments $\Gamma \vdash^{\mathsf{Env}}_{e,\neg t_{if}} \Gamma_2$ and $\Gamma_2 \vdash e_2 : t$, and so we have all the premises to apply the [Case] rule in order to get $\Gamma\rho \vdash (e\rho{\in}t_{if})\,?\,e_1\rho : e_2\rho : t'$.

[Base] Trivial.

[Path] We have by using the induction hypothesis $\Gamma\rho \vdash^{\mathsf{Env}}_{e\rho,t} \Gamma'\rho$ and $\forall t''.\ \Gamma' \vdash e_a : t'' \Rightarrow \Gamma'\rho \vdash e_b : t''$.
First, let's show that we can derive $\Gamma\rho \vdash^{\mathsf{Env}}_{e\rho,t} \Gamma''\rho$ with $\Gamma'' = \Gamma', (e{\downarrow}\varpi : t')$.
There are two cases:

- $e{\downarrow}\varpi$ is a strict subexpression of e_a.
  In this case, it means that among its three possible forms, e_a is of the form vv or (v, v). According to the assumptions we made on the derivation at the beginning of the proof, it implies that $\varpi = \epsilon$. Hence, e does not contain any occurence of e_a, so it is easy to conclude.
- $e{\downarrow}\varpi$ is not a strict subexpression of e_a.
  In this case, we know that $e\rho{\downarrow}\varpi$ is defined.

Thus we can apply the induction hypothesis on $\vdash^{\mathsf{Path}}_{\Gamma',e,t} \varpi : t'$. It gives $\vdash^{\mathsf{Path}}_{\Gamma'\rho,e\rho,t} \varpi : t'$. If $e\rho{\downarrow}\varpi \in \mathrm{dom}(\Gamma'\rho)$, and $(\Gamma'\rho)(e\rho{\downarrow}\varpi) = t'' \not\geq t'$, then we can derive $\vdash^{\mathsf{Path}}_{\Gamma'\rho,e\rho,t} \varpi : t' \wedge t''$ just by using the rules [PInter], [PTypeof] and [Env].

Using this last judgement together with $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$, we can derive with the rule [Path] the wanted $\Gamma\rho \vdash^{\mathsf{Env}}_{e\rho,t} \Gamma''\rho$.

Now, let's show that $\forall t'.\ \Gamma'' \vdash e_a : t' \Rightarrow \Gamma''\rho \vdash e_b : t'$.

Let t' such that $\Gamma'' \vdash e_a : t'$.

Recall that we have $\Gamma' \vdash e_a : t' \Rightarrow \Gamma'\rho \vdash e_b : t'$.

If $\Gamma'' = \bot$, then $\Gamma''\rho = \bot$ so we are done. So lets's suppose $\Gamma'' \neq \bot$.

Let's separate the proof in two cases:

- If $e{\downarrow}\varpi \not\equiv e_a$. In this case, let's show that we have $\Gamma' \vdash e_a : t'$. Indeed, in the typing derivation of $\Gamma'' \vdash e_a : t'$, the [Env] rules can only be applied on subexpressions of e_a.
  If $e{\downarrow}\varpi$ is not a strict subexpression of e_a (and thus not a subexpression as $e{\downarrow}\varpi \not\equiv e_a$), there is no [Env] rule applied to $e{\downarrow}\varpi$ in the derivation of $\Gamma'' \vdash e_a : t'$ and thus we can easily derive $\Gamma' \vdash e_a : t'$.
  If $e{\downarrow}\varpi$, is a strict subexpression of e_a, it must be a value (given the possible forms of e_a). Moreover, as $\Gamma'' \neq \bot$, we have $\forall v \in \mathrm{dom}(\Gamma'').\ v \in [\![\Gamma''(v)]\!]_{\mathcal{V}}$ (recall the assumptions at the beginning of the proof) and thus $\forall v \in \mathrm{dom}(\Gamma'').\ \Gamma' \vdash v : \Gamma''(v)$. Thus we can derive $\Gamma' \vdash e_a : t'$ just by replacing every [Env] rule applied to $e{\downarrow}\varpi$ in the derivation of $\Gamma'' \vdash e_a : t'$ by the relevant derivation.
  From $\Gamma' \vdash e_a : t'$ we deduce $\Gamma'\rho \vdash e_b : t'$. As $\Gamma'' \leq \Gamma'$ (according to the assumptions we made on the derivation at the beginning of the proof) and $\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma'')$, we have $\Gamma''\rho \leq \Gamma'\rho$ and thus, by monotonicity, $\Gamma''\rho \vdash e_b : t'$.
- If $e{\downarrow}\varpi \equiv e_a$. Let's note $t_a = \Gamma''(e_a)$. This time, we can't derive $\Gamma' \vdash e_a : t'$ from $\Gamma'' \vdash e_a : t'$ because the rule [Env] could be used on $e{\downarrow}\varpi = e_a$ (which may not be a value).
  However, the rule [Env] can only be used on e_a at the end of the derivation of $\Gamma'' \vdash e_a : t'$: there can't be any [App], [Abs+], [Proj], [Pair] or [Case] after because the premises of these rules only contain strict subexpressions of their consequence. Thus, we can easily transform the derivation so that every [Env] applied on e_a is directly followed by an [Inter]: if there is any [Abs-] or [Subs] between, we can move it after.
  Then, we can (temporarily) remove from the derivation all [Env] applied on e_a: for each, we just replace the following [Inter] rule by its other premise.
  It yields a derivation for $\Gamma'' \vdash e_a : t''$ such that $t'' \wedge t_a \leq t'$ and without any [Env] applied to e_a. Thus, we can transform it into a derivation of $\Gamma' \vdash e_a : t''$ as in the previous point, and we get $\Gamma'\rho \vdash e_b : t''$. Still as before, we get a derivation for $\Gamma''\rho \vdash e_b : t''$ by monotonicity.
  Now, we can append at the end of this derivation a rule [Inter] with a rule [Env] applied to e_b. As $(\Gamma''\rho)(e_b) \leq \Gamma''(e_a) = t_a$, we obtain a derivation for $\Gamma''\rho \vdash e_b : t'$ (we can add a final [Subs] rule if needed).

[PTypeof] Trivial (by using the induction hypothesis).

[P$\cdots$] All the remaining rules are trivial.

□

Theorem A.14 (Subject reduction). *Let Γ an ordinary environment, e and e' two expressions and t a type.*

If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Proof. Let Γ, e, e' and t as in the statement.

We construct a derivation for $\Gamma \vdash e' : t$ by induction on the derivation of $\Gamma \vdash e : t$.
If $\Gamma = \bot$ this theorem is trivial, so we can suppose $\Gamma \neq \bot$.
We proceed by case analysis on the last rule of the derivation:

[Env] As Γ is ordinary, it means that e is a variable. It contradicts the fact that e reduces to e' so this case is impossible.

[Efq] This case is impossible as $\Gamma \neq \bot$.

[Inter] Trivial (by using the induction hypothesis).

[Subs] Trivial (by using the induction hypothesis).

[Const] Impossible case (no reduction possible).

[App] In this case, $e \equiv e_1 e_2$. There are three possible cases:

- e_2 is not a value. In this case, we must have $e_2 \rightsquigarrow e_2'$ and $e' \equiv e_1 e_2'$. We can easily conclude using the induction hypothesis.
- e_2 is a value and e_1 is not. In this case, we must have $e_1 \rightsquigarrow e_1'$ and $e' \equiv e_1' e_2$. We can easily conclude using the induction hypothesis.
- Both e_1 and e_2 are values. This is the difficult case. We have $e_1 \equiv \lambda^{\bigwedge_{i\in I} s_i \to t_i} x.e_x$ with $\bigwedge_{i\in I} s_i \to t_i \leq s \to t$ and $\Gamma \vdash e_2 : s$. We can suppose that x is a new fresh variable that does not appear in our environment (if it is not the case, we can alpha-rename e_1).
  This means that $s \leq \bigvee_{i\in I} s_i$ and that for any non-empty I' such that $s \not\leq \bigvee_{i\in I\setminus I'} s_i$, we have $\bigwedge_{i\in I'} t_i \leq t$ (see lemma 6.8 of [Frisch et al. 2008]). Let's take $I' = \{i \in I \mid e_2 \in [\![s_i]\!]_{\mathcal{V}}\}$. We have I' not empty: $e_2 \in [\![s]\!]_{\mathcal{V}}$ and $s \leq \bigvee_{i\in I} s_i$, so according to $[\![\_]\!]_{\mathcal{V}}$ properties we have at least one i such that $e_2 \in [\![s_i]\!]_{\mathcal{V}}$. We also have $s \not\leq \bigvee_{i\in I\setminus I'} s_i$, otherwise there would be a $i \notin I$ such that $e_2 \in [\![s_i]\!]_{\mathcal{V}}$ (contradiction with the definition of I'). As a consequence, we get $\bigwedge_{i\in I'} t_i \leq t$.
  Now, let's prove that $\Gamma \vdash e' : \bigwedge_{i\in I'} t_i$ (which, by subsumption, yields $\Gamma \vdash e' : t$). For that, we show that for any $i \in I'$, $\Gamma \vdash e' : t_i$ (it is then easy to conclude by using the [Inter] rule).
  Let $i \in I'$. We have $e_2 \in [\![s_i]\!]_{\mathcal{V}}$, and so $\Gamma \vdash e_2 : s_i$ (e_2 is well-typed in Γ). As e_1 is well-typed in Γ, there must be in its derivation an application of the rule [Abs+] which guarantees $\Gamma, (x : s_i) \vdash e_x : t_i$ (recall that $\Gamma \neq \bot$ and Γ is ordinary so there is no abstraction in $\mathrm{dom}(\Gamma)$). Let's note $\Gamma' = \Gamma, (x : s_i)$. We can deduce, using the substitution lemma, that $\Gamma'\{x \mapsto e_2\} \vdash e_x\{x \mapsto e_2\} : t_i$.
  Moreover, $\Gamma'\{x \mapsto e_2\} = \Gamma, (e_2 : s_i)$ and $\Gamma \leq \Gamma, (e_2 : s_i)$. Thus, by monotonicity, we deduce $\Gamma \vdash e_x\{x \mapsto e_2\} : t_i$, that is $\Gamma \vdash e' : t_i$.

[Abs+] Impossible case (no reduction possible).

[Abs-] Impossible case (no reduction possible).

[Proj] In this case, $e \equiv \pi_i e_0$. There are two possible cases:

- e_0 is not a value. In this case, we must have $e_0 \rightsquigarrow e_0'$ and $e' \equiv \pi_i e_0'$. We can easily conclude using the induction hypothesis.
- e_0 is a value. Given that $e_0 \leq \mathbb{1} \times \mathbb{1}$, we have $e_0 = (v_1, v_2)$ with v_1 and v_2 two values. We also have $e \rightsquigarrow v_i$.
  The derivation of $\Gamma \vdash (v_1, v_2) : t_1 \times t_2$ must contain a rule [Pair] which guarantees $\Gamma \vdash v_i : t_i$ (recall that $\Gamma \neq \bot$ and Γ is ordinary so there is no pair in $\mathrm{dom}(\Gamma)$). It concludes this case.

[Pair] In this case, $e \equiv (e_1, e_2)$. There are two possible cases:

- e_2 is not a value. In this case, we must have $e_2 \rightsquigarrow e_2'$ and $e' \equiv (e_1, e_2')$. We can easily conclude using the induction hypothesis.
- e_2 is a value and e_1 is not. In this case, we must have $e_1 \rightsquigarrow e_1'$ and $e' \equiv (e_1', e_2)$. We can easily conclude using the induction hypothesis.

[Case] In this case, $e \equiv (e_0 \in t_{if})\,?\,e_1 : e_2$. There are three possible cases:

- e_0 is a value and $e_0 \in [\![t_{if}]\!]_{\mathcal{V}}$. In this case we have $e' \equiv e_1$. We have derivations for $\Gamma \vdash e_0 : t_0$, $\Gamma \vdash^{\mathsf{Env}}_{e_0, t_{if}} \Gamma'$ and $\Gamma' \vdash e_1 : t$.
  As e_0 is a value and $e_0 \in [\![t_{if}]\!]_{\mathcal{V}}$, we have $\Gamma \leq \Gamma'$ by using the value testing lemma. Thus, by monotonicity, we have $\Gamma \vdash e_1 : t$.
- e_0 is a value and $e_0 \notin [\![t]\!]_{\mathcal{V}}$. This case is similar to the previous one (we replace t_{if} by $\neg t_{if}$ and e_1 by e_2).
- e_0 is not a value. In this case, we have $e_0 \overset{e_a \mapsto e_b}{\rightsquigarrow} e_0'$ and $e' \equiv (e_0\rho \in t_{if})\,?\,e_1\rho : e_2\rho \equiv e\rho$ with $\rho = \{e_a \mapsto e_b\}$.
  First, let's notice that we have e_b closed (only closed expressions are reducible), and e_a has one of the following forms:
  – $(e \in t)\,?\,e_1 : e_2$ (if expression)
  – vv (application of two values)
  – (v, v) (product of two values)
  It can be easily proved by induction on the derivation of the reduction step.
  Secondly, as $e_a \rightsquigarrow e_b$ and as the derivation of this reduction is a strict subderivation of that of $e \rightsquigarrow e'$, we can use the induction hypothesis on $e_a \rightsquigarrow e_b$ and we obtain $\forall t'.\ \Gamma \vdash e_a : t' \Rightarrow \Gamma\rho \vdash e_b : t'$.
  Thus, we can conclude directly by using the substitution lemma on e and ρ.

□

A.3.3 *Progress.*

Lemma A.15 (Inversion).

$$[\![t_1 \times t_2]\!]_{\mathcal{V}} = \{(v_1, v_2) \mid \vdash_{\mathcal{V}} v_1 : t_1, \vdash_{\mathcal{V}} v_2 : t_2\}$$
$$[\![b]\!]_{\mathcal{V}} = \{c \mid \boldsymbol{b}_c \leq b\}$$
$$[\![t \to s]\!]_{\mathcal{V}} = \{\lambda^{\bigwedge_{i\in I} t_i \to s_i} x.e \mid \bigwedge_{i\in I} t_i \to s_i \leq t \to s\}$$

Proof. See lemma 6.21 of [Frisch et al. 2008] □

Theorem A.16 (Progress). *If $\varnothing \vdash e : t$, then either e is a value or there exists e' such that $e \rightsquigarrow e'$.*

Proof. We proceed by induction on the derivation $\varnothing \vdash e : t$. We consider the last rule of this derivation:

[Env] This case is impossible (the environment is empty).
[Efq] This case is impossible (the environment is empty).
[Inter] Straightforward application of the induction hypothesis.
[Subs] Straightforward application of the induction hypothesis.
[Const] In this case, e must be a constant so e is a value.
[App] We have $e = e_1\ e_2$, with $\varnothing \vdash e_1 : s \to t$ and $\varnothing \vdash e_2 : s$. If one of the e_i can be reduced, then e can also be reduced using the reduction rule [Ctx].
Otherwise, by using the induction hypothesis we get that both e_1 and e_2 are values. Moreover, by using the inversion lemma, we know that e_1 has the form $\lambda^{\bigwedge_{i\in I} t_i \to s_i} x.e_0$. In consequence, e is reducible (the reduction rule [App] can be applied).
[Abs+] In this case, e must be a lambda abstraction, so e is a value.
[Abs-] Straightforward application of the induction hypothesis.

[Case] We have $e = (e_0 \in t')\ ?\ e_1 : e_2$. If e_0 can be reduced, then e can also be reduced using the reduction rule [TestCtx].
Otherwise, by using the induction hypothesis we get that e_0 is a value. In consequence, e is reducible (the reduction rule [Case] can be applied).

[Proj] We have $e = \pi_i e_0, t = t_i, \varnothing \vdash e_0 : t_1 \times t_2$. If e_0 can be reduced, then e can also be reduced using the rule [Ctx].
Otherwise, by using the induction hypothesis we get that e_0 is a value. Moreover, by using the inversion lemma, we know that e_0 has the form (v_1, v_2). In consequence, e is reducible (the reduction rule [Proj] can be applied).

[Pair] We have $e = (e_1, e_2)$. If one of the e_i can be reduced, then e can also be reduced using the reduction rule [Ctx].
Otherwise, by using the induction hypothesis we get that both e_1 and e_2 are values. In consequence, e is also a value.

□

A.4 Operator ▪

In this section, we will use the algorithmic definition of ▪ and show that it is equivalent to its descriptive definition.

$$t \simeq \bigvee_{i \in I} \left(\bigwedge_{p \in P_i} (s_p \to t_p) \bigwedge_{n \in N_i} \neg (s'_n \to t'_n) \right)$$

$$t \blacksquare s = \mathsf{dom}(t) \wedge \bigvee_{i \in I} \left(\bigwedge_{\{P \subseteq P_i \mid s \leq \bigvee_{p \in P} \neg t_p\}} \left(\bigvee_{p \in P} \neg s_p \right) \right)$$

Lemma A.17 (Correctness of ▪). $\forall t, s.\ t \circ (\mathit{dom}(t) \setminus (t \blacksquare s)) \leq \neg s$

Proof. Let t an arrow type. $t \simeq \bigvee_{i \in I} \left(\bigwedge_{p \in P_i} (s_p \to t_p) \bigwedge_{n \in N_i} \neg (s'_n \to t'_n) \right)$
Let s be any type.

Let's prove that $t \circ (\mathsf{dom}(t) \setminus (t \blacksquare s)) \leq \neg s$ (with the algorithmic definition for ▪).
Equivalently, we want $(t \circ (\mathsf{dom}(t) \setminus (t \blacksquare s))) \wedge s \simeq \mathbb{0}$.

Let u be a type such that $u \leq \mathsf{dom}(t)$ and $(t \circ u) \wedge s \not\simeq \mathbb{0}$ (if such a type does not exist, we are done).
Let's show that $u \wedge (t \blacksquare s) \not\simeq \mathbb{0}$ (we can easily deduce the wanted property from that, by the absurd).
For that, we should prove the following:

$$\exists i \in I.\ u \wedge \bigwedge_{\{P \subseteq P_i \mid s \leq \bigvee_{p \in P} \neg t_p\}} \left(\bigvee_{p \in P} \neg s_p \right) \not\simeq \mathbb{0}$$

From $(t \circ u) \wedge s \not\simeq \mathbb{0}$, we can take (using the algorithmic definition of $\circ$) $i \in I$ and $Q \subsetneq P_i$ such that:

$$u \not\leq \bigvee_{q \in Q} s_q \quad \text{and} \quad \left(\bigwedge_{p \in P_i \setminus Q} t_p \right) \wedge s \not\simeq \mathbb{0}$$

For any $P \subseteq P_i$ such that $s \leq \bigvee_{p \in P} \neg t_p$ (equivalently, $s \wedge \bigwedge_{p \in P} t_p \simeq \mathbb{0}$),
we have $P \cap Q \neq \varnothing$ (by the absurd, because $(\bigwedge_{p \in P_i \setminus Q} t_p) \wedge s \not\simeq \mathbb{0}$).

Consequently, we have:

$$\forall P \subseteq P_i.\ s \leq \bigvee_{p\in P} \neg t_p \Rightarrow \bigwedge_{p\in P} s_p \leq \bigvee_{q\in Q} s_q$$

We can deduce that:

$$\bigvee_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}} \left(\bigwedge_{p\in P} s_p\right) \leq \bigvee_{q\in Q} s_q$$

Moreover, as $u \not\leq \bigvee_{q\in Q} s_q$, we have $u \not\leq \bigvee_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}} \left(\bigwedge_{p\in P} s_p\right)$.
This is equivalent to the wanted result. □

Lemma A.18 (▪ alternative definition). *The following algorithmic definition for ▪ is equivalent to the previous one:*

$$\forall t, s.\ t \blacksquare s \simeq \bigvee_{i\in I}\left(\bigvee_{\{P\subseteq P_i \mid s\not\leq \bigvee_{p\in P}\neg t_p\}}\left(dom(t) \wedge \bigwedge_{p\in P_i} s_p \wedge \bigwedge_{n\in P_i\setminus P} \neg s_n\right)\right)$$

Proof.

$$\begin{aligned}
t \blacksquare s &= \mathsf{dom}(t) \wedge \bigvee_{i\in I}\left(\bigwedge_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\mathsf{dom}(t) \wedge \bigwedge_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\left(\mathsf{dom}(t) \wedge \bigvee_{p\in P_i} s_p\right) \wedge \bigwedge_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\left(\mathsf{dom}(t) \wedge \bigvee_{p\in P_i}\left(s_p \wedge \bigvee_{P\subseteq P_i\setminus\{p\}}\left(\bigwedge_{p\in P} s_p \wedge \bigwedge_{n\in (P_i\setminus\{p\})\setminus P}\neg s_n\right)\right)\right) \wedge \bigwedge_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\left(\mathsf{dom}(t) \wedge \bigvee_{p\in P_i}\left(\bigvee_{P\subseteq P_i\setminus\{p\}}\left(s_p \wedge \bigwedge_{p\in P} s_p \wedge \bigwedge_{n\in (P_i\setminus\{p\})\setminus P}\neg s_n\right)\right)\right) \wedge \bigwedge_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\left(\mathsf{dom}(t) \wedge \bigvee_{\substack{P\subseteq P_i\\ P\neq\varnothing}}\left(\bigwedge_{p\in P} s_p \wedge \bigwedge_{n\in P_i\setminus P}\neg s_n\right)\right) \wedge \bigwedge_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\mathsf{dom}(t) \wedge \bigvee_{\substack{P\subseteq P_i\\ P\neq\varnothing}}\left(\bigwedge_{p\in P} s_p \wedge \bigwedge_{n\in P_i\setminus P}\neg s_n\right) \setminus \bigvee_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigwedge_{p\in P} s_p\right)\right)\\
&\simeq \bigvee_{i\in I}\left(\mathsf{dom}(t) \wedge \bigvee_{\substack{P\subseteq P_i\\ P\neq\varnothing}}\left(\bigwedge_{p\in P} s_p \wedge \bigwedge_{n\in P_i\setminus P}\neg s_n\right) \setminus \bigvee_{\{P\subseteq P_i \mid s\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigwedge_{p\in P} s_p \wedge \bigwedge_{n\in P_i\setminus P}\neg s_n\right)\right)
\end{aligned}$$

$$\simeq \bigvee_{i\in I}\left(\mathsf{dom}(t)\wedge \bigvee_{\{P\subseteq P_i \mid s\not\leq \bigvee_{p\in P}\neg t_p\}}\left(\bigwedge_{p\in P_i} s_p \wedge \bigwedge_{n\in P_i\setminus P}\neg s_n\right)\right)$$

$$\simeq \bigvee_{i\in I}\left(\bigvee_{\{P\subseteq P_i \mid s\not\leq \bigvee_{p\in P}\neg t_p\}}\left(\mathsf{dom}(t)\wedge \bigwedge_{p\in P_i} s_p \wedge \bigwedge_{n\in P_i\setminus P}\neg s_n\right)\right)$$

□

Lemma A.19 (Optimality of $\blacksquare$). *Let t, s, two types. For any u such that $t\circ(dom(t)\setminus u)\leq \neg s$, we have $t\blacksquare s\leq u$.*

Proof. Let t an arrow type. $t\simeq \bigvee_{i\in I}\left(\bigwedge_{p\in P_i}(s_p\to t_p)\bigwedge_{n\in N_i}\neg(s'_n\to t'_n)\right)$

Let s be any type.

Let u be such that $t\circ(\mathsf{dom}(t)\setminus u)\leq\neg s$. We want to prove that $t\blacksquare s\leq u$.

We have:

$$t\blacksquare s=\bigvee_{i\in I}\left(\bigvee_{\{P\subseteq P_i \mid s\not\leq \bigvee_{p\in P}\neg t_p\}} a_{i,P}\right)$$

With:

$$a_{i,P}=\mathsf{dom}(t)\wedge\bigwedge_{p\in P_i}s_p\wedge\bigwedge_{n\in P_i\setminus P}\neg s_n$$

Let $i\in I$ and $P\subseteq P_i$ such that $s\not\leq\bigvee_{p\in P}\neg t_p$ (equivalently, $s\wedge\bigwedge_{p\in P}t_p\neq\mathbb{0}$) and such that $a_{i,P}\neq\mathbb{0}$.

For convenience, let $a=a_{i,P}$. We just have to show that $a\leq u$.

By the absurd, let's suppose that $a\setminus u\neq\mathbb{0}$ and show that $(t\circ(\mathsf{dom}(t)\setminus u))\wedge s\neq\mathbb{0}$.

Let's recall the algorithmic definition of $\circ$:

$$t\circ(\mathsf{dom}(t)\setminus u)=\bigvee_{i\in I}\left(\bigvee_{\{Q\subsetneq P_i \mid \mathsf{dom}(t)\setminus u\not\leq\bigvee_{q\in Q}s_q\}}\left(\bigwedge_{p\in P_i\setminus Q}t_p\right)\right)$$

Let's take $Q=P_i\setminus P$. We just have to prove that:

$$\mathsf{dom}(t)\setminus u\not\leq\bigvee_{q\in Q}s_q\quad\text{and}\quad s\wedge\bigwedge_{p\in P_i\setminus Q}t_p\neq\mathbb{0}$$

As $P_i\setminus Q=P$, we immediatly have $s\wedge\bigwedge_{p\in P_i\setminus Q}t_p\neq\mathbb{0}$.

Moreover, we know that $a\leq\bigwedge_{q\in Q}\neg s_q$ (definition of $a_{i,P}$), so we have:

$$a\wedge\bigwedge_{q\in Q}\neg s_q\simeq a$$

Thus:

$$(a\setminus u)\wedge\bigwedge_{q\in Q}\neg s_q\simeq(a\wedge\bigwedge_{q\in Q}\neg s_q)\setminus u\simeq a\setminus u\neq\mathbb{0}$$

And so:

$$a\setminus u\not\leq\bigvee_{q\in Q}s_q$$

As $\mathsf{dom}(t)\setminus u\geq a\setminus u$, we can immediatly obtain the remaining inequality. □

Theorem A.20 (Characterization of $\blacksquare$). $\forall t, s.\ t \blacksquare s = \min\{u \mid t \circ (dom(t) \setminus u) \leq \neg s\}$.

Proof. Immediate consequence of the previous results. □

A.5 Full algorithmic type system

$$[\text{Efq}_{\mathcal{A}}]\ \frac{}{\Gamma, (e : \mathbb{0}) \vdash_{\mathcal{A}} e' : \mathbb{0}}\ \text{with priority over all the other rules} \qquad [\text{Var}_{\mathcal{A}}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} x : \Gamma(x)}\ x \in \text{dom}(\Gamma)$$

$$[\text{Env}_{\mathcal{A}}]\ \frac{\Gamma \setminus \{e\} \vdash_{\mathcal{A}} e : \mathbb{t}}{\Gamma \vdash_{\mathcal{A}} e : \Gamma(e) \mathbin{\mathbb{\&}} \mathbb{t}}\ e \in \text{dom}(\Gamma) \text{ and } e \text{ not a variable} \qquad [\text{Const}_{\mathcal{A}}]\ \frac{}{\Gamma \vdash_{\mathcal{A}} c : \boldsymbol{b_c}}\ c \notin \text{dom}(\Gamma)$$

$$[\text{Abs}_{\mathcal{A}}]\ \frac{\Gamma, x : s_i \vdash_{\mathcal{A}} e : \mathbb{t}'_i \qquad \mathbb{t}'_i \leq t_i}{\Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : [s_i \to t_i]_{i\in I}}\ \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \notin \text{dom}(\Gamma)$$

$$[\text{App}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e_1 : \mathbb{t}_1 \qquad \Gamma \vdash_{\mathcal{A}} e_2 : \mathbb{t}_2 \qquad \mathbb{t}_1 \leq \mathbb{0} \to \mathbb{1} \qquad \mathbb{t}_2 \leq \text{dom}(\mathbb{t}_1)}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \mathbb{t}_1 \circ \mathbb{t}_2}\ e_1 e_2 \notin \text{dom}(\Gamma)$$

$$[\text{Case}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e : \mathbb{t}_0 \qquad \text{Refine}_{e,t}(\Gamma) \vdash_{\mathcal{A}} e_1 : \mathbb{t}_1 \qquad \text{Refine}_{e,\neg t}(\Gamma) \vdash_{\mathcal{A}} e_2 : \mathbb{t}_2}{\Gamma \vdash_{\mathcal{A}} (e \in t)\,?\,e_1 : e_2 : \mathbb{t}_1 \mathbin{\mathbb{\vee}} \mathbb{t}_2}\ (e \in t)\,?e_1 : e_2 \notin \text{dom}(\Gamma)$$

$$[\text{Proj}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e : \mathbb{t} \qquad \mathbb{t} \leq \mathbb{1} \times \mathbb{1}}{\Gamma \vdash_{\mathcal{A}} \pi_i e : \boldsymbol{\pi_i}(\mathbb{t})}\ \pi_i e \notin \text{dom}(\Gamma)$$

$$[\text{Pair}_{\mathcal{A}}]\ \frac{\Gamma \vdash_{\mathcal{A}} e_1 : \mathbb{t}_1 \qquad \Gamma \vdash_{\mathcal{A}} e_2 : \mathbb{t}_2}{\Gamma \vdash_{\mathcal{A}} (e_1, e_2) : \mathbb{t}_1 \otimes \mathbb{t}_2}\ (e_1, e_2) \notin \text{dom}(\Gamma)$$

$$\text{typeof}_\Gamma(e) = \begin{cases} \mathbb{t} & \text{if } \Gamma \vdash_{\mathcal{A}} e : \mathbb{t} \\ \Omega & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\text{Constr}_{\Gamma,e,t}(\epsilon) &= t && (32)\\
\text{Constr}_{\Gamma,e,t}(\varpi.0) &= \neg(\text{Intertype}_{\Gamma,e,t}(\varpi.1) \to \neg\text{Intertype}_{\Gamma,e,t}(\varpi)) && (33)\\
\text{Constr}_{\Gamma,e,t}(\varpi.1) &= \text{Repr}(\text{typeof}_\Gamma(e{\downarrow}\varpi.0)) \blacksquare \text{Intertype}_{\Gamma,e,t}(\varpi) && (34)\\
\text{Constr}_{\Gamma,e,t}(\varpi.l) &= \boldsymbol{\pi_1}(\text{Intertype}_{\Gamma,e,t}(\varpi)) && (35)\\
\text{Constr}_{\Gamma,e,t}(\varpi.r) &= \boldsymbol{\pi_2}(\text{Intertype}_{\Gamma,e,t}(\varpi)) && (36)\\
\text{Constr}_{\Gamma,e,t}(\varpi.f) &= \text{Intertype}_{\Gamma,e,t}(\varpi) \times \mathbb{1} && (37)\\
\text{Constr}_{\Gamma,e,t}(\varpi.s) &= \mathbb{1} \times \text{Intertype}_{\Gamma,e,t}(\varpi) && (38)\\
\text{Intertype}_{\Gamma,e,t}(\varpi) &= \text{Repr}(\text{Constr}_{\Gamma,e,t}(\varpi) \mathbin{\mathbb{\&}} \text{typeof}_\Gamma(e{\downarrow}\varpi)) && (39)
\end{aligned}$$

$\mathsf{RefineStep}_{e,t}(\Gamma) = \Gamma'$ with:

$$\mathsf{dom}(\Gamma') = \mathsf{dom}(\Gamma) \cup \{e' \mid \exists \varpi.\ e{\downarrow}\varpi \equiv e'\}$$

$$\Gamma'(e') = \begin{cases} \bigwedge_{\{\varpi \mid e{\downarrow}\varpi \equiv e'\}} \mathsf{Intertype}_{\Gamma,e,t}(\varpi) & \text{if } \exists \varpi.\ e{\downarrow}\varpi \equiv e' \\ \Gamma(e') & \text{otherwise} \end{cases}$$

$$\mathsf{Refine}_{e,t}(\Gamma) = \mathsf{RefineStep}_{e,t}{}^{n_o}(\Gamma) \qquad \text{with } n \text{ a global parameter}$$

A.6 Proofs for the algorithmic type system

This section is about the algorithmic type system (soundness and some completeness properties).

Note that we use a different but more convenient definition for $\mathsf{typeof}_\Gamma(e)$ that the one in Section 2.5.3:

$$\mathsf{typeof}_\Gamma(e) = \begin{cases} \mathbb{t} & \text{if } \Gamma \vdash_{\mathcal{A}} e : \mathbb{t} \\ \Omega & \text{otherwise} \end{cases}$$

In this way, $\mathsf{typeof}_\Gamma(e)$ is always defined but is equal to Ω when e is not well-typed in Γ.

We will reuse the definitions and notations introduced in the previous proofs. In particular, we only consider well-formed environments, as in the proofs of the declarative type system.

A.6.1 Soundness.

Theorem A.21 (Soundness of the algorithm). *For every Γ, e, t, n_o, if $\mathsf{typeof}_\Gamma(e) \leq t$, then we can derive $\Gamma \vdash e : t$.*

More precisely:

$$\forall \Gamma, e, t.\ \mathsf{typeof}_\Gamma(e) \leq t \Rightarrow \Gamma \vdash e : t$$

$$\forall \Gamma, e, t, \varpi.\ \mathsf{typeof}_\Gamma(e) \neq \Omega \Rightarrow \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$$

$$\forall \Gamma, e, t.\ \mathsf{typeof}_\Gamma(e) \neq \Omega \Rightarrow \Gamma \vdash^{\mathsf{Env}}_{e,t} \mathsf{Refine}_{e,t}(\Gamma)$$

Proof. We proceed by induction over the structure of e and, for two identical e, on the domain of Γ (with the inclusion order).

Let's prove the first property. Let t such that $\{\mathsf{typeof}_\Gamma(e)\} \leq t$.

If $\Gamma = \bot$, we trivially have $\Gamma \vdash e : t$ with the rule [Efq]. Let's assume $\Gamma \neq \bot$.

If $e = x$ is a variable, then the last rule used is [$\mathrm{Var}_{\mathcal{A}}$]. We can derive $\Gamma \vdash x : t$ by using the rule [Env] and [Subs]. So let's assume that e is not a variable.

If $e \in \mathsf{dom}(\Gamma)$, then the last rule used is [$\mathrm{Env}_{\mathcal{A}}$]. Let $t' \in \{\mathbb{t}\}$ such that $t' \wedge \Gamma(e) \leq t$. The induction hypothesis gives $\Gamma \setminus \{e\} \vdash e : t'$ (the premise uses the same e but the domain of Γ is strictly smaller). Thus, we can build a derivation $\Gamma \vdash e : t$ by using the rules [Subs], [Inter], [Env] and the derivation $\Gamma \setminus \{e\} \vdash e : t'$.

Now, let's suppose that $e \notin \mathsf{dom}(\Gamma)$.

- $e = c$ The last rule is [$\mathrm{Const}_{\mathcal{A}}$]. We derive easily $\Gamma \vdash c : t$ with [Const] and [Subs].
- $e = x$ Already treated.
- $e = \lambda^{\bigwedge_{i\in I} t_i \to s_i} x.e'$ The last rule is [$\mathrm{Abs}_{\mathcal{A}}$]. We have $\bigwedge_{i\in I} t_i \to s_i \leq t$. Using the definition of type schemes, let $t' = \bigwedge_{i\in I} t_i \to s_i \wedge \bigwedge_{j\in J} \neg t'_j \to s'_j$ such that $\mathbb{0} \neq t' \leq t$. The induction hypothesis gives, for all $i \in I$, $\Gamma, x : s_i \vdash e' : t_i$.
  Thus, we can derive $\Gamma \vdash e : \bigwedge_{i\in I} t_i \to s_i$ using the rule [Abs+], and with [Inter] and [Abs-] we can derive $\Gamma \vdash e : t'$. We can conclude by applying [Subs].

$e = e_1 e_2$ The last rule is [$\text{App}_{\mathcal{A}}$]. We have $\mathbb{t}_1 \circ \mathbb{t}_2 \leq t$. Thus, let t_1 and t_2 such that $\mathbb{t}_1 \leq t_1$, $\mathbb{t}_2 \leq t_2$ and $t_1 \circ t_2 \leq t$. We know, according to the descriptive definition of $\circ$, that there exists $s \leq t$ such that $t_1 \leq t_2 \to s$.
By using the induction hypothesis, we have $\Gamma \vdash e_1 : t_1$ and $\Gamma \vdash e_2 : t_2$. We can thus derive $\Gamma \vdash e_1 : t_2 \to s$ using [Subs], and together with $\Gamma \vdash e_2 : t_2$ it gives $\Gamma \vdash e_1\ e_2 : s$ with [App]. We conclude with [Subs].

$e = \pi_i e'$ The last rule is [$\text{Proj}_{\mathcal{A}}$]. We have $\boldsymbol{\pi}_i \mathbb{t} \leq t$. Thus, let t' such that $\mathbb{t} \leq t'$ and $\boldsymbol{\pi}_i t' \leq t$. We know, according to the descriptive definition of $\boldsymbol{\pi}_i$, that there exists $t_i \leq t$ such that $t' \leq \mathbb{1} \times t_i$ (for $i = 2$) or $t' \leq t_i \times \mathbb{1}$ (for $i = 1$).
By using the induction hypothesis, we have $\Gamma \vdash e' : t'$, and thus we easily conclude using [Subs] and [Proj] (for instance for the case $i = 1$, we can derive $\Gamma \vdash e' : t_i \times \mathbb{1}$ with [Subs] and then use [Proj]).

$e = (e_1, e_2)$ The last rule is [$\text{Pair}_{\mathcal{A}}$]. We conclude easily with the induction hypothesis and the rules [Subs] and [Pair].

$e = (e_0 \in t)\ \mathbf{?}\ e_1 : e_2$ The last rule is [$\text{Case}_{\mathcal{A}}$]. We conclude easily with the induction hypothesis and the rules [Subs] and [Case] (for the application of [Case], t' must be taken equal to $t_1 \vee t_2$ with t_1 and t_2 such that $\mathbb{t}_1 \leq t_1$, $\mathbb{t}_2 \leq t_2$ and $t_1 \vee t_2 \leq t$).

Now, let's prove the second property. We perform a (nested) induction on ϖ.

Recall that $\mathsf{Intertype}_{\Gamma,e,t}(\varpi) = \mathsf{Repr}(\mathsf{Constr}_{\Gamma,e,t}(\varpi) \mathbin{\text{⩓}} \mathsf{typeof}_\Gamma(e{\downarrow}\varpi))$.

For any t' such that $\mathsf{typeof}_\Gamma(e{\downarrow}\varpi) \leq t'$, we can easily derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : t'$ by using the outer induction hypothesis (the first property that we have proved above) and the rule [PTypeof].

Now we have to derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathsf{Constr}_{\Gamma,e,t}(\varpi)$ (then it will be easy to conclude using the rule [PInter]).

$\varpi = \epsilon$ We use the rule [PEps].

$\varpi = \varpi'.1$ Let's note $f = \mathsf{Repr}(\mathsf{typeof}_\Gamma(e{\downarrow}\varpi'.0))$, $s = \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$ and $t_{\text{res}} = f \blacksquare s$.
By using the outer and inner induction hypotheses, we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi'.0 : f$ and $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : s$.
By using the descriptive definition of $\blacksquare$, we have $t' = f \circ (\mathsf{dom}(f) \setminus t_{\text{res}}) \leq \neg s$.
Moreover, by using the descriptive definition of $\circ$ on t', we have $f \leq (\mathsf{dom}(f) \setminus t_{\text{res}}) \to t'$.
As $t' \leq \neg s$, it gives $f \leq (\mathsf{dom}(f) \setminus t_{\text{res}}) \to \neg s$.
Let's note $t_1 = \mathsf{dom}(f) \setminus t_{\text{res}}$ and $t_2 = \neg s$. The above inequality can be rewritten $f \leq t_1 \to t_2$.
Thus, by using [PSubs] on the derivation $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi'.0 : f$, we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi'.0 : t_1 \to t_2$.
We have:
- $t_2 \wedge s \simeq \mathbb{0}$ (as $t_2 = \neg s$)
- $\neg t_1 = t_{\text{res}} \vee \neg \mathsf{dom}(f) = t_{\text{res}}$

In consequence, we can conclude by applying the rule [PAppR] with the premises $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi'.0 : t_1 \to t_2$ and $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : s$.

$\varpi = \varpi'.0$ By using the inner induction hypothesis and the previous case we've just proved, we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$ and $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi'.1 : \mathsf{Intertype}_{\Gamma,e,t}(\varpi'.1)$. Hence we can apply [PAppL].

$\varpi = \varpi'.l$ Let's note $t_1 = \boldsymbol{\pi}_1 \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$. According to the descriptive definition of $\boldsymbol{\pi}_1$, we have $\mathsf{Intertype}_{\Gamma,e,t}(\varpi') \leq t_1 \times \mathbb{1}$.

The inner induction hypothesis gives $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$, and thus using the rule [PSubs] we can derive $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : t_1 \times \mathbb{1}$. We can conclude just by applying the rule [PPairL] to this premise.

- $\varpi = \varpi'.r$ This case is similar to the previous.
- $\varpi = \varpi'.f$ The inner induction hypothesis gives $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$, so we can conclude by applying [PFst].
- $\varpi = \varpi'.s$ The inner induction hypothesis gives $\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$, so we can conclude by applying [PSnd].

Finally, let's prove the third property. Let $\Gamma' = \mathsf{Refine}_{e,t}(\Gamma) = {\mathsf{RefineStep}_{e,t}}^{n_0}(\Gamma)$. We want to show that $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ is derivable.

First, let's note that $\vdash^{\mathsf{Env}}_{e,t}$ is transitive: if $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ and $\Gamma' \vdash^{\mathsf{Env}}_{e,t} \Gamma''$, then $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma''$. The proof is quite easy: we can just start from the derivation of $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$, and we add at the end a slightly modified version of the derivation of $\Gamma' \vdash^{\mathsf{Env}}_{e,t} \Gamma''$ where:

- the initial [Base] rule has been removed in order to be able to do the junction,
- all the Γ' at the left of $\vdash^{\mathsf{Env}}_{e,t}$ are replaced by Γ (the proof is still valid as this Γ' at the left is never used in any rule)

Thanks to this property, we can suppose that $n_0 = 1$ (and so $\Gamma' = \mathsf{RefineStep}_{e,t}(\Gamma)$). If it is not the case, we just have to proceed by induction on n_0 and use the transitivity property.

Let's build a derivation for $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$.

By using the proof of the second property on e that we've done just before, we get: $\forall \varpi.\ \vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi : \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$.

Let's recall a monotonicity property: for any Γ_1 and Γ_2 such that $\Gamma_2 \leq \Gamma_1$, we have $\forall t'.\ \vdash^{\mathsf{Path}}_{\Gamma_1,e,t} \varpi : t' \Rightarrow \vdash^{\mathsf{Path}}_{\Gamma_2,e,t} \varpi : t'$.

Moreover, when we also have $e{\downarrow}\varpi \in \mathsf{dom}(\Gamma_2)$, we can derive $\vdash^{\mathsf{Path}}_{\Gamma_2,e,t} \varpi : t' \wedge \Gamma_2(e{\downarrow}\varpi)$ (just by adding a [PInter] rule with a [PTypeof] and a [Env]).

Hence, we can apply successively a [Path] rule for all valid ϖ in e, with the following premises (Γ_ϖ being the previous environment, that trivially verifies $\Gamma_\varpi \leq \Gamma$):

If $e{\downarrow}\varpi \in \mathsf{dom}(\Gamma_\varpi)$	$\vdash^{\mathsf{Path}}_{\Gamma_\varpi,e,t} \varpi : \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \wedge \Gamma_\varpi(e{\downarrow}\varpi)$	$\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_\varpi$
Otherwise	$\vdash^{\mathsf{Path}}_{\Gamma_\varpi,e,t} \varpi : \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$	$\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_\varpi$

At the end, it gives the judgement $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$, so it concludes the proof. □

A.6.2 Completeness.

Definition A.22 (Bottom environment). Let Γ an environment.
Γ is bottom (noted $\Gamma = \bot$) iff $\exists e \in \mathsf{dom}(\Gamma).\ \Gamma(e) \simeq \mathbb{0}$.

Definition A.23 (Algorithmic (pre)order on environments). Let Γ and Γ' two environments. We write $\Gamma' \leq_{\mathcal{A}} \Gamma$ iff:

$$\Gamma' = \bot \text{ or } (\Gamma \neq \bot \text{ and } \forall e \in \mathsf{dom}(\Gamma).\ \mathsf{typeof}_{\Gamma'}(e) \leq \Gamma(e))$$

For an expression e, we write $\Gamma' \leq^e_{\mathcal{A}} \Gamma$ iff:

$\Gamma' = \bot$ or ($\Gamma \neq \bot$ and $\forall e' \in \mathsf{dom}(\Gamma)$ such that e' is a subexpression of e. $\mathsf{typeof}_{\Gamma'}(e') \leq \Gamma(e')$)

Note that if $\Gamma' \leq_{\mathcal{A}} \Gamma$, then $\Gamma' \leq^e_{\mathcal{A}} \Gamma$ for any e.

Definition A.24 (Order relation for type schemes). Let $\mathbb{t}_1$ and $\mathbb{t}_2$ two type schemes. We write $\mathbb{t}_2 \leq \mathbb{t}_1$ iff $\{\mathbb{t}_1\} \subseteq \{\mathbb{t}_2\}$.

Definition A.25 (Positive derivation). A derivation of the declarative type system is said positive iff it does not contain any rule [Abs-].

Definition A.26 (Rank-0 negated derivation). A derivation of the declarative type system is said rank-0 negated iff any application of [PAppL] has a positive derivation as first premise ($\vdash^{\mathsf{Path}}_{\Gamma,e,t} \varpi.1 : t_1$).

Lemma A.27.

$$\forall t, \mathbb{t}.\ \mathsf{Repr}(t \owedge \mathbb{t}) \leq t \wedge \mathsf{Repr}(\mathbb{t})$$
$$\forall t, \mathbb{t}.\ t \owedge \mathbb{t} \neq \mathbb{0} \Rightarrow \mathsf{Repr}(t \owedge \mathbb{t}) \simeq t \wedge \mathsf{Repr}(\mathbb{t})$$
$$\forall \mathbb{t}_1, \mathbb{t}_2.\ \mathsf{Repr}(\mathbb{t}_1 \circ \mathbb{t}_2) \leq \mathsf{Repr}(\mathbb{t}_1) \circ \mathsf{Repr}(\mathbb{t}_2)$$

Proof. Straightfoward, by induction on the structure of $\mathbb{t}$. □

Lemma A.28 (Monotonicity of the algorithm). *Let* Γ, Γ' *and* e *such that* $\Gamma' \leq^e_{\mathscr{A}} \Gamma$ *and* $\mathsf{typeof}_\Gamma(e) \neq \Omega$. *We have:*

$$\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_\Gamma(e) \text{ and } \mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e)) \leq \mathsf{Repr}(\mathsf{typeof}_\Gamma(e))$$
$$\forall t, \varpi.\ \mathsf{Intertype}_{\Gamma',e,t}(\varpi) \leq \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$$
$$\forall t.\ \mathsf{Refine}_{e,t}(\Gamma') \leq^e_{\mathscr{A}} \mathsf{Refine}_{e,t}(\Gamma)$$

Proof. We proceed by induction over the structure of e and, for two identical e, on the domains of Γ and Γ' (with the lexicographical inclusion order).

Let's prove the first property: $\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_\Gamma(e)$ and $\mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e)) \leq \mathsf{Repr}(\mathsf{typeof}_\Gamma(e))$. We will focus on showing $\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_\Gamma(e)$.

The property $\mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e)) \leq \mathsf{Repr}(\mathsf{typeof}_\Gamma(e))$ can be proved in a very similar way, by using the fact that operators on type schemes like $\owedge$ or $\circ$ are also monotone. (Note that the only rule that introduces the type scheme constructor [_] is [$\mathrm{Abs}_{\mathscr{A}}$].)

If $\Gamma' = \bot$ we can conclude directly with the rule [Efq]. So let's assume $\Gamma' \neq \bot$ and $\Gamma \neq \bot$ (as $\Gamma = \bot \Rightarrow \Gamma' = \bot$ by definition of $\leq^e_{\mathscr{A}}$).

If $e = x$ is a variable, then the last rule used in $\mathsf{typeof}_\Gamma(e)$ and $\mathsf{typeof}_{\Gamma'}(e)$ is [$\mathrm{Var}_{\mathscr{A}}$]. As $\Gamma' \leq^e_{\mathscr{A}} \Gamma$, we have $\Gamma'(e) \leq \Gamma(e)$ and thus we can conclude with the rule [$\mathrm{Var}_{\mathscr{A}}$]. So let's assume that e is not a variable.

If $e \in \mathsf{dom}(\Gamma)$, then the last rule used in $\mathsf{typeof}_\Gamma(e)$ is [$\mathrm{Env}_{\mathscr{A}}$]. As $\Gamma' \leq^e_{\mathscr{A}} \Gamma$, we have $\mathsf{typeof}_{\Gamma'}(e) \leq \Gamma(e)$. Moreover, by applying the induction hypothesis, we get $\mathsf{typeof}_{\Gamma'\setminus\{e\}}(e) \leq \mathsf{typeof}_{\Gamma\setminus\{e\}}(e)$ (we can easily verify that $\Gamma' \setminus \{e\} \leq^e_{\mathscr{A}} \Gamma \setminus \{e\}$).

- If we have $e \in \mathsf{dom}(\Gamma')$, we have according to the rule [$\mathrm{Env}_{\mathscr{A}}$] $\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_{\Gamma'\setminus\{e\}}(e) \leq \mathsf{typeof}_{\Gamma\setminus\{e\}}(e)$.
  Together with $\mathsf{typeof}_{\Gamma'}(e) \leq \Gamma(e)$, we deduce $\mathsf{typeof}_{\Gamma'}(e) \leq \Gamma(e) \owedge \mathsf{typeof}_{\Gamma\setminus\{e\}}(e) = \mathsf{typeof}_\Gamma(e)$.
- Otherwise, we have $e \notin \mathsf{dom}(\Gamma')$. Thus $\mathsf{typeof}_{\Gamma'}(e) = \mathsf{typeof}_{\Gamma'\setminus\{e\}}(e) \leq \Gamma(e) \owedge \mathsf{typeof}_{\Gamma\setminus\{e\}}(e) = \mathsf{typeof}_\Gamma(e)$.

If $e \notin \mathsf{dom}(\Gamma)$ and $e \in \mathsf{dom}(\Gamma')$, the last rule is [$\mathrm{Env}_{\mathscr{A}}$] for $\mathsf{typeof}_{\Gamma'}(e)$. As $\Gamma' \setminus \{e\} \leq^e_{\mathscr{A}} \Gamma \setminus \{e\} = \Gamma$, we have $\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_{\Gamma'\setminus\{e\}}(e) \leq \mathsf{typeof}_\Gamma(e)$ by induction hypothesis.

Thus, let's suppose that $e \notin \mathsf{dom}(\Gamma)$ and $e \notin \mathsf{dom}(\Gamma')$. From now we know that the last rule in the derivation of $\mathsf{typeof}_\Gamma(e)$ and $\mathsf{typeof}_{\Gamma'}(e)$ (if any) is the same.

$e = c$ The last rule is [Const$_{\mathcal{A}}$]. It does not depend on Γ so this case is trivial.

$e = x$ Already treated.

$e = \lambda^{\bigwedge_{i\in I} t_i \to s_i} x.e'$ The last rule is [Abs$_{\mathcal{A}}$]. We have $\forall i \in I.\ \Gamma', (x : s_i) \leq^{e'}_{\mathcal{A}} \Gamma, (x : s_i)$ (quite straightforward) so by applying the induction hypothesis we have $\forall i \in I.\ \mathsf{typeof}_{\Gamma', (x:s_i)}(e') \leq \mathsf{typeof}_{\Gamma, (x:s_i)}(e')$.

$e = e_1 e_2$ The last rule is [App$_{\mathcal{A}}$]. We can conclude immediately by using the induction hypothesis and noticing that $\circ$ is monotonic for both of its arguments.

$e = \pi_i e'$ The last rule is [Proj$_{\mathcal{A}}$]. We can conclude immediately by using the induction hypothesis and noticing that $\boldsymbol{\pi}_i$ is monotonic.

$e = (e_1, e_2)$ The last rule is [Pair$_{\mathcal{A}}$]. We can conclude immediately by using the induction hypothesis.

$e = (e_0 \in t)\ ?\ e_1 : e_2$ The last rule is [Case$_{\mathcal{A}}$]. By using the induction hypothesis we get $\mathsf{Refine}_{e_0,t}(\Gamma') \leq^{e_0}_{\mathcal{A}} \mathsf{Refine}_{e_0,t}(\Gamma)$. We also have $\Gamma' \leq^{e_1}_{\mathcal{A}} \Gamma$ (as e_1 is a subexpression of e).
From those two properties, let's show that we can deduce $\mathsf{Refine}_{e_0,t}(\Gamma') \leq^{e_1}_{\mathcal{A}} \mathsf{Refine}_{e_0,t}(\Gamma)$:
Let $e' \in \mathsf{dom}(\mathsf{Refine}_{e_0,t}(\Gamma))$ a subexpression of e_1.

- If e' is also a subexpression of e_0, we can directly deduce $\mathsf{typeof}_{\mathsf{Refine}_{e_0,t}(\Gamma)'}(e') \leq (\mathsf{Refine}_{e_0,t}(\Gamma))(e')$ by using $\mathsf{Refine}_{e_0,t}(\Gamma') \leq^{e_0}_{\mathcal{A}} \mathsf{Refine}_{e_0,t}(\Gamma)$.
- Otherwise, as $\mathsf{Refine}_{e_0,t}(\_)$ is reductive, we have $\mathsf{Refine}_{e_0,t}(\Gamma') \leq_{\mathcal{A}} \Gamma'$ and thus by using the induction hypothesis $\mathsf{typeof}_{\mathsf{Refine}_{e_0,t}(\Gamma')}(e') \leq \mathsf{typeof}_{\Gamma'}(e')$. We also have $\mathsf{typeof}_{\Gamma'}(e') \leq \Gamma(e')$ by using $\Gamma' \leq^{e_1}_{\mathcal{A}} \Gamma$. We deduce $\mathsf{typeof}_{\mathsf{Refine}_{e_0,t}(\Gamma')}(e') \leq \Gamma(e') = (\mathsf{Refine}_{e_0,t}(\Gamma))(e')$.

So we have $\mathsf{Refine}_{e_0,t}(\Gamma') \leq^{e_1}_{\mathcal{A}} \mathsf{Refine}_{e_0,t}(\Gamma)$. Consequently, we can apply the induction hypothesis again to get $\mathsf{typeof}_{\mathsf{Refine}_{e_0,t}(\Gamma')}(e_1) \leq \mathsf{typeof}_{\mathsf{Refine}_{e_0,t}(\Gamma)}(e_1)$.
We proceed the same way for the last premise.

Now, let's prove the second property. We perform a (nested) induction on ϖ.

Recall that we have $\forall t, \mathbb{t}.\ t \mathbin{⩓} \mathbb{t} \not\simeq \mathbb{0} \Rightarrow \mathsf{Repr}(t \mathbin{⩓} \mathbb{t}) \simeq t \wedge \mathsf{Repr}(\mathbb{t})$.

Thus, in order to prove $\mathsf{Repr}(\mathsf{Constr}_{\Gamma',e,t}(\varpi) \mathbin{⩓} \mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi)) \leq \mathsf{Repr}(\mathsf{Constr}_{\Gamma,e,t}(\varpi) \mathbin{⩓} \mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi))$, we can prove the following:

$$\mathsf{Constr}_{\Gamma',e,t}(\varpi) \leq \mathsf{Constr}_{\Gamma,e,t}(\varpi)$$
$$\mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi) \leq \mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi)$$
$$\mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi)) \leq \mathsf{Repr}(\mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi))$$

The two last inequalities can be proved with the outer induction hypothesis (for $\varpi = \epsilon$ we use the proof of the first property above).

Thus we just have to prove that $\mathsf{Constr}_{\Gamma',e,t}(\varpi) \leq \mathsf{Constr}_{\Gamma,e,t}(\varpi)$. The only case that is interesting is the case $\varpi = \varpi'.1$.

First, we can notice that the $\blacksquare$ operator is monotonic for its second argument (consequence of its declarative definition).

Secondly, let's show that for any function types $t_1 \leq t_2$, and for any type t', we have $(t_1 \blacksquare t') \wedge \mathsf{dom}(t_2) \leq t_2 \blacksquare t'$. By the absurd, let's suppose it is not true. Let's note $t'' = (t_1 \blacksquare t') \wedge \mathsf{dom}(t_2)$. Then we have $t'' \leq \mathsf{dom}(t_2) \leq \mathsf{dom}(t_1)$ and $t_2 \leq t'' \to t'$ and $t_1 \not\leq t'' \to t'$, which contradicts $t_1 \leq t_2$.

Let's note $t_1 = \mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi'.0))$ and $t_2 = \mathsf{Repr}(\mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi'.0))$ and $t' = \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$. As e is well-typed, and using the inner induction hypothesis, we have $\mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi'.1)) \leq \mathsf{Repr}(\mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi'.1)) \leq \mathsf{dom}(t_2)$.

Thus, using this property, we get:

$$\begin{aligned}&(t_1 \blacksquare t') \wedge \mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi'.1))\\ \leq &(t_2 \blacksquare t') \wedge \mathsf{Repr}(\mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi'.1))\end{aligned}$$

Then, using the monotonicity of the second argument of $\blacksquare$ and the outer induction hypothesis:

$$\begin{aligned}&(t_1 \blacksquare \mathsf{Intertype}_{\Gamma',e,t}(\varpi')) \wedge \mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e{\downarrow}\varpi'.1))\\ \leq &(t_2 \blacksquare \mathsf{Intertype}_{\Gamma,e,t}(\varpi')) \wedge \mathsf{Repr}(\mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi'.1))\end{aligned}$$

Finally, we must prove the third property.
It is straightforward by using the previous result and the induction hypothesis:
$\forall e'$ s.t. $\exists \varpi.\ e{\downarrow}\varpi \equiv e'$, we get $\bigwedge_{\{\varpi \mid e{\downarrow}\varpi \equiv e'\}} \mathsf{Intertype}_{\Gamma',e,t}(\varpi) \leq \bigwedge_{\{\varpi \mid e{\downarrow}\varpi \equiv e'\}} \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$.
The rest follows. □

Theorem A.29 (Completeness for positive derivations). *For every Γ, e, t such that we have a positive derivation of $\Gamma \vdash e : t$, there exists a global parameter n_o with which* $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t$.
More precisely:

$\forall \Gamma, e, t.\ \Gamma \vdash e : t$ *has a positive derivation* $\implies \mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t$

$\forall \Gamma, \Gamma', e, t.\ \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ *has a positive derivation* $\implies \mathsf{Refine}_{e,t}(\Gamma) \leq_{\mathcal{A}} \Gamma'$ *(for n_o large enough)*

Proof. We proceed by induction on the derivation.
Let's prove the first property. We have a positive derivation of $\Gamma \vdash e : t$.
If $\Gamma = \bot$, we can conclude directly using [Efq$_{\mathcal{A}}$]. Thus, let's suppose $\Gamma \neq \bot$.
If $e = x$ is a variable, then the derivation only uses [Env], [Inter] and [Subs]. We can easily conclude just be using [Var$_{\mathcal{A}}$]. Thus, let's suppose e is not a variable.
If $e \in \mathsf{dom}(\Gamma)$, we can have the rule [Env] applied to e in our derivation, but in this case there can only be [Inter] and [Subs] after it (not [Abs-] as we have a positive derivation). Thus, our derivation contains a derivation of $\Gamma \vdash e : t'$ that does not use the rule [Env] on e and such that $t' \wedge \Gamma(e) \leq t$ (actually, it is possible for our derivation to typecheck e only using the rule [Env]: in this case we can take $t' = \mathbb{1}$ and use the fact that Γ is well-formed). Hence, we can build a positive derivation for $\Gamma \setminus \{e\} \vdash e : t'$. By using the induction hypothesis we deduce that $\mathsf{Repr}(\mathsf{typeof}_{\Gamma\setminus\{e\}}(e)) \leq t'$. Thus, by looking at the rule [Env$_{\mathcal{A}}$], we deduce $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq \Gamma(e) \wedge \mathsf{Repr}(\mathsf{typeof}_{\Gamma\setminus\{e\}}(e)) \leq t$. It concludes this case, so let's assume $e \notin \mathsf{dom}(\Gamma)$.
Now we analyze the last rule of the derivation:

[Env] Impossible case ($e \notin \mathsf{dom}(\Gamma)$).

[Inter] By using the induction hypothesis we get $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t_1$ and $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t_2$. Thus, we have $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t_1 \wedge t_2$.

[Subs] Trivial using the induction hypothesis.

[Const] We know that the derivation of $\mathsf{typeof}_\Gamma(e)$ (if any) ends with the rule [Const$_{\mathcal{A}}$]. Thus this case is trivial.

[App] We know that the derivation of $\mathsf{typeof}_\Gamma(e)$ (if any) ends with the rule [App$_{\mathcal{A}}$]. Let $\mathbb{t}_1 = \mathsf{typeof}_\Gamma(e_1)$ and $\mathbb{t}_2 = \mathsf{typeof}_\Gamma(e_2)$. With the induction hypothesis we have $\mathsf{Repr}(\mathbb{t}_1) \leq t_1 \to t_2$ and $\mathsf{Repr}(\mathbb{t}_2) \leq t_1$, with $t_2 = t$. According to the descriptive definition of $\circ$, we have $\mathsf{Repr}(\mathbb{t}_1) \circ \mathsf{Repr}(\mathbb{t}_2) \leq t_1 \to t_2 \circ t_1 \leq t_2$. As we also have $\mathsf{Repr}(\mathbb{t}_1 \circ \mathbb{t}_2) \leq \mathsf{Repr}(\mathbb{t}_1) \circ \mathsf{Repr}(\mathbb{t}_2)$, we can conclude that $\mathsf{typeof}_\Gamma(e) \leq t_2 = t$.

[Abs+] We know that the derivation of $\text{typeof}_\Gamma(e)$ (if any) ends with the rule $[\text{Abs}_\mathcal{A}]$. This case is straightforward using the induction hypothesis.

[Abs-] This case is impossible (the derivation is positive).

[Case] We know that the derivation of $\text{typeof}_\Gamma(e)$ (if any) ends with the rule $[\text{Case}_\mathcal{A}]$. By using the induction hypothesis and the monotonicity lemma, we get $\text{Repr}(\mathbb{t}_1) \leq t$ and $\text{Repr}(\mathbb{t}_2) \leq t$. So we have $\text{Repr}(\mathbb{t}_1 \otimes \mathbb{t}_2) = \text{Repr}(\mathbb{t}1) \vee \text{Repr}(\mathbb{t}2) \leq t$.

[Proj] Quite similar to the case [App].

[Pair] We know that the derivation of $\text{typeof}_\Gamma(e)$ (if any) ends with the rule $[\text{Pair}_\mathcal{A}]$. We just use the induction hypothesis and the fact that $\text{Repr}(\mathbb{t}_1 \otimes \mathbb{t}_2) = \text{Repr}(\mathbb{t}1) \times \text{Repr}(\mathbb{t}2)$.

Now, let's prove the second property. We have a positive derivation of $\Gamma \vdash^{\text{Env}}_{e,t} \Gamma'$.

[Base] Any value of n_o will give $\text{Refine}_{e,t}(\Gamma) \leq_\mathcal{A} \Gamma$, even $n_o = 0$.

[Path] We have $\Gamma' = \Gamma_1, (e{\downarrow}\varpi : t')$. By applying the induction hypothesis on the premise $\Gamma \vdash^{\text{Env}}_{e,t} \Gamma_1$, we have $\text{RefineStep}^n_{e,t}(\Gamma) = \Gamma_2$ with $\Gamma_2 \leq_\mathcal{A} \Gamma_1$ for a certain n.
We now proceed by induction on the derivation $\vdash^{\text{Path}}_{\Gamma_1,e,t} \varpi : t'$ to show that we can obtain $\text{Intertype}_{\Gamma'',e,t}(\varpi) \leq t'$ with $\Gamma'' = \text{RefineStep}^{n'}_{e,t}(\Gamma_2)$ for a certain n'. It is then easy to conclude by taking $n_o = n + n'$.

[PSubs] Trivial using the induction hypothesis.

[PInter] By using the induction hypothesis we get:

$$\begin{aligned}
&\text{Intertype}_{\Gamma''_1,e,t}(\varpi) \leq t_1\\
&\text{Intertype}_{\Gamma''_2,e,t}(\varpi) \leq t_2\\
&\text{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_\mathcal{A} \Gamma''_1\\
&\text{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_\mathcal{A} \Gamma''_2
\end{aligned}$$

By taking $n' = \max(n_1, n_2)$, we can have $\Gamma'' = \text{RefineStep}^{n'}_{e,t}(\Gamma_2)$ with $\Gamma'' \leq_\mathcal{A} \Gamma''_1$ and $\Gamma'' \leq_\mathcal{A} \Gamma''_2$. Thus, by using the monotonicity lemma, we can obtain $\text{Intertype}_{\Gamma'',e,t}(\varpi) \leq t_1 \wedge t_2 = t'$.

[PTypeof] By using the outer induction hypothesis we get $\text{Repr}(\text{typeof}_{\Gamma_2}(e{\downarrow}\varpi)) \leq t'$. Moreover we have $\text{Intertype}_{\Gamma_2,e,t}(\varpi) \leq \text{Repr}(\text{typeof}_{\Gamma_2}(e{\downarrow}\varpi))$ (by definition of Intertype), thus we can conclude directly.

[PEps] Trivial.

[PAppR] By using the induction hypothesis we get:

$$\begin{aligned}
&\text{Intertype}_{\Gamma''_1,e,t}(\varpi.0) \leq t_1 \to t_2\\
&\text{Intertype}_{\Gamma''_2,e,t}(\varpi) \leq t'_2\\
&t_2 \wedge t'_2 \simeq \mathbb{0}\\
&\text{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_\mathcal{A} \Gamma''_1\\
&\text{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_\mathcal{A} \Gamma''_2
\end{aligned}$$

By taking $n' = \max(n_1, n_2)+1$, we can have $\Gamma'' = \text{RefineStep}^{n'}_{e,t}(\Gamma_2)$ with $\Gamma'' \leq_\mathcal{A} \text{RefineStep}_{e,t}(\Gamma''_1)$ and $\Gamma'' \leq_\mathcal{A} \text{RefineStep}_{e,t}(\Gamma''_2)$.
In consequence, we have $\text{Repr}(\text{typeof}_{\Gamma''}(e{\downarrow}\varpi.0)) \leq \text{Intertype}_{\Gamma''_1,e,t}(\varpi.0) \leq t_1 \to t_2$ (by definition of $\text{RefineStep}_{e,t}$). We also have, by monotonicity, $\text{Intertype}_{\Gamma'',e,t}(\varpi) \leq t'_2$.

As $t_2 \wedge t_2' \simeq \mathbb{0}$, we have:

$$\begin{aligned}&(t_1 \to t_2) \circ (\mathsf{dom}(t_1 \to t_2) \setminus (\neg t_1))\\ &\simeq (t_1 \to t_2) \circ t_1 \simeq t_2 \leq \neg t_2'\end{aligned}$$

Thus, by using the declarative definition of $\blacksquare$, we know that $(t_1 \to t_2) \blacksquare t_2' \leq \neg t_1$. According to the properties on $\blacksquare$ that we have proved in the proof of the monotonicity lemma, we can deduce:

$$\begin{aligned}&t_1 \wedge \mathsf{Repr}(\mathsf{typeof}_{\Gamma''}(e{\downarrow}\varpi.0)) \blacksquare \mathsf{Intertype}_{\Gamma'',e,t}(\varpi)\\ &\leq t_1 \wedge (t_1 \to t_2) \blacksquare t_2' \leq t_1 \wedge \neg t_1 \simeq \mathbb{0}\end{aligned}$$

And thus $\mathsf{Repr}(\mathsf{typeof}_{\Gamma''}(e{\downarrow}\varpi.0)) \blacksquare \mathsf{Intertype}_{\Gamma'',e,t}(\varpi) \leq \neg t_1$.
It concludes this case.

[PAppL] By using the induction hypothesis we get:

$$\begin{aligned}&\mathsf{Intertype}_{\Gamma_1'',e,t}(\varpi.1) \leq t_1\\ &\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \leq t_2\\ &\mathsf{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_{\mathscr{A}} \Gamma_1''\\ &\mathsf{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_{\mathscr{A}} \Gamma_2''\end{aligned}$$

By taking $n' = \max(n_1, n_2)$, we can have $\Gamma'' = \mathsf{RefineStep}^{n'}_{e,t}(\Gamma_2)$ with $\Gamma'' \leq_{\mathscr{A}} \Gamma_1''$ and $\Gamma'' \leq_{\mathscr{A}} \Gamma_2''$. Thus, by using the monotonicity lemma, we can obtain $\mathsf{Intertype}_{\Gamma'',e,t}(\varpi.0) \leq \neg(t_1 \to \neg t_2) = t'$.

[PPairL] Quite straightforward using the induction hypothesis and the descriptive definition of $\boldsymbol{\pi}_1$.

[PPairR] Quite straightforward using the induction hypothesis and the descriptive definition of $\boldsymbol{\pi}_2$.

[PFst] Trivial using the induction hypothesis.

[PSnd] Trivial using the induction hypothesis.

□

Simple type	t_s	::=	$b \mid t_s \times t_s \mid t_s \vee t_s \mid \neg t_s \mid \mathbb{0} \mid \mathbb{0} \to \mathbb{1}$
Positive type	t_+	::=	$t_s \mid t_+ \vee t_+ \mid t_+ \wedge t_+ \mid t_+ \to t_+ \mid t_+ \to \neg t_+$
Positive abstraction type	t_+^λ	::=	$t_+ \to t_+ \mid t_+ \to \neg t_+ \mid t_+^\lambda \wedge t_+^\lambda$
Positive expression	e_+	::=	$c \mid x \mid e_+ e_+ \mid \lambda^{t_+^\lambda} x.e_+ \mid \pi_j e_+ \mid (e_+, e_+) \mid (e_+ \in t_s)\,?\,e_+ : e_+$

Corollary A.30. *If we restrict the language to positive expressions e_+, the algorithmic type system is complete and type schemes can be removed from it (we can use regular types instead).*
More precisely: $\forall \Gamma, e_+, t.\ \Gamma \vdash e_+ : t \Rightarrow \mathsf{typeof}_\Gamma(e_+) \neq \Omega$

Proof. With such restrictions, the rule [Abs-] is not needed anymore because the negative part of functional types (i.e. the N_i part of their DNF) is useless.

Indeed, when typing an application $e_1 e_2$, the negative part of the type of e_1 is ignored by the operator $\circ$.

Moreover, as there is no negated arrows in the domain of lambda-abstractions, the negative arrows of the type of e_2 can also be ignored.

Similarly, negative arrows can be ignored when refining an application ($\blacksquare$ also ignore the negative part of the type of e_1).

Finally, as the only functional type that we can test is $\mathbb{0} \to \mathbb{1}$, a functional type cannot be refined to $\mathbb{0}$ due to its negative part, and thus we can ignore its negative part (it makes no difference relatively to the rule [Efq]). □

Lemma A.31. *If e is an application, then* $\mathsf{typeof}_\Gamma(e)$ *does not contain any constructor* $[\cdots]$. *Consequently, we have* $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \simeq \mathsf{typeof}_\Gamma(e)$.

Proof. By case analysis: neither [Efq], [$\text{Env}_\mathcal{A}$] nor [$\text{App}_\mathcal{A}$] can produce a type containing a constructor $[\cdots]$. □

Theorem A.32 (Completeness for rank-0 negated derivations). *For every Γ, e, t such that we have a rank-0 negated derivation of $\Gamma \vdash e : t$, there exists a global parameter n_o with which* $\mathsf{typeof}_\Gamma(e) \leq t$.

More precisely:

$\forall \Gamma, e, t.\ \Gamma \vdash e : t$ *has a rank-0 negated derivation* $\implies \mathsf{typeof}_\Gamma(e) \leq t$

$\forall \Gamma, \Gamma', e, t.\ \Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$ *has a rank-0 negated derivation* $\implies \mathsf{Refine}_{e,t}(\Gamma) \leq_\mathcal{A} \Gamma'$ *(for n_o large enough)*

Proof. This proof is quite similar to that of the completeness for positive derivations. In consequence, we will only detail cases that are quite different from those of the previous proof.

Let's begin with the first property. We have a rank-0 negated derivation of $\Gamma \vdash e : t$. We want to show $\mathsf{typeof}_\Gamma(e) \leq t$ (note that this is weaker than showing $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t$).

As in the previous proof, we can suppose that $\Gamma \neq \bot$ and that e is not a variable.

The case $e \in \mathsf{dom}(\Gamma)$ is also very similar, but there is an additional case to consider: the rule [Abs-] could possibly be used after a rule [Env] applied on e. However, this case can easily be eliminated by changing the premise of this [Abs-] with another one that does not use the rule [Env] on e (the type of the premise does not matter for the rule [Abs-], even $\mathbb{1}$ suffices). Thus let's assume $e \notin \mathsf{dom}(\Gamma)$.

Now we analyze the last rule of the derivation (only the cases that are not similar are shown):

[Abs-] We know that the derivation of $\mathsf{typeof}_\Gamma(e)$ (if any) ends with the rule [$\text{Abs}_\mathcal{A}$]. Moreover, by using the induction hypothesis on the premise, we know that $\mathsf{typeof}_\Gamma(e) \neq \Omega$. Thus we have $\mathsf{typeof}_\Gamma(e) \leq \neg(t_1 \to t_2) = t$ (because every type $\neg(s' \to t')$ such that $\neg(s' \to t') \wedge \bigwedge_{i \in I} s_i \to t_i \neq \mathbb{0}$ is in $\{[s_i \to t_i]\}$).

Now let's prove the second property. We have a rank-0 negated derivation of $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma'$.

[Base] Any value of n_o will give $\mathsf{Refine}_{e,t}(\Gamma) \leq_\mathcal{A} \Gamma$, even $n_o = 0$.

[Path] We have $\Gamma' = \Gamma_1, (e{\downarrow}\varpi : t')$.

As in the previous proof of completeness, by applying the induction hypothesis on the premise $\Gamma \vdash^{\mathsf{Env}}_{e,t} \Gamma_1$, we have $\mathsf{RefineStep}^n_{e,t}(\Gamma) = \Gamma_2$ with $\Gamma_2 \leq_\mathcal{A} \Gamma_1$ for a certain n.

However, this time, we can't prove $\mathsf{Intertype}_{\Gamma'',e,t}(\varpi) \leq t'$ with $\Gamma'' = \mathsf{RefineStep}^{n'}_{e,t}(\Gamma_2)$ for a certain n': the induction hypothesis is weaker than in the previous proof (we don't have $\mathsf{Repr}(\mathsf{typeof}_\Gamma(e)) \leq t$ but only $\mathsf{typeof}_\Gamma(e) \leq t$).

Instead, we will prove by induction on the derivation $\vdash^{\mathsf{Path}}_{\Gamma_1,e,t} \varpi : t'$ that $\mathsf{Intertype}_{\Gamma'',e,t}(\varpi) \between \mathsf{typeof}_{\Gamma''}(e{\downarrow}\varpi) \leq t'$. It suffices to conclude in the same way as in the previous proof: by taking $n_o = n + n'$, it ensures that our final environment Γ_{n_o} verifies $\mathsf{typeof}_\Gamma(e{\downarrow}\varpi)_{n_o} \leq t'$ and thus we have $\Gamma_{n_o} \leq \Gamma'$ (given that $\mathsf{Repr}(\mathbb{0}) = \mathbb{0}$, we also easily verify that if $\Gamma' = \bot \implies \Gamma_{n_o} = \bot$).

[PSubs] Trivial using the induction hypothesis.

[PInter] Quite similar to the previous proof (the induction hypothesis is weaker, but it works the same way).

[PTypeof] By using the outer induction hypothesis we get $\mathsf{typeof}_{\Gamma_2}(e{\downarrow}\varpi) \leq t'$ so it is trivial.
[PEps] Trivial.
[PAppR] By using the induction hypothesis, we get:

$$\mathsf{Intertype}_{\Gamma_1'',e,t}(\varpi.0) \between \mathsf{typeof}_{\Gamma_1''}(e{\downarrow}\varpi.0) \leq t_1 \to t_2$$
$$\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \between \mathsf{typeof}_{\Gamma_2''}(e{\downarrow}\varpi) \leq t_2'$$
$$t_2 \wedge t_2' \simeq \mathbb{0}$$
$$\mathsf{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_{\mathcal{A}} \Gamma_1''$$
$$\mathsf{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_{\mathcal{A}} \Gamma_2''$$

Moreover, as $e{\downarrow}\varpi$ is an application, we can use the lemma above to deduce $\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \between \mathsf{typeof}_{\Gamma_2''}(e{\downarrow}\varpi) = \mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi)$ (see definition of Intertype).
Thus we have $\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \leq t_2'$.
We also have $\mathsf{Intertype}_{\Gamma_1'',e,t}(\varpi.0) \leq \mathsf{Repr}(\mathsf{Intertype}_{\Gamma_1'',e,t}(\varpi.0) \between \mathsf{typeof}_{\Gamma_1''}(e{\downarrow}\varpi.0)) \leq t_1 \to t_2$.
Now we can conclude exactly as in the previous proof (by taking $n' = \max(n_1, n_2)$).
[PAppL] We know that the left premise is a positive derivation. Thus, using the previous completeness theorem, we get:

$$\mathsf{Intertype}_{\Gamma_1'',e,t}(\varpi.1) \leq t_1$$
$$\mathsf{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_{\mathcal{A}} \Gamma_1''$$

By using the induction hypothesis, we also get:

$$\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \between \mathsf{typeof}_{\Gamma_2''}(e{\downarrow}\varpi) \leq t_2$$
$$\mathsf{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_{\mathcal{A}} \Gamma_2''$$

Moreover, as $e{\downarrow}\varpi$ is an application, we can use the lemma above to deduce $\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \between \mathsf{typeof}_{\Gamma_2''}(e{\downarrow}\varpi) = \mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi)$ (see definition of Intertype).
Thus we have $\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \leq t_2$.
Now we can conclude exactly as in the previous proof (by taking $n' = \max(n_1, n_2)$).
[PPairL] Quite straightforward using the induction hypothesis and the descriptive definition of $\boldsymbol{\pi}_1$.
[PPairR] Quite straightforward using the induction hypothesis and the descriptive definition of $\boldsymbol{\pi}_2$.
[PFst] Quite straightforward using the induction hypothesis.
[PSnd] Quite straightforward using the induction hypothesis.

□

B A MORE PRECISE RULE FOR INFERENCE

In our prototype we have implemented for the inference of arrow type the following rule:

$$\frac{\begin{array}{c}[\textsc{AbsInf++}]\\ T = \{(s \setminus \bigvee_{s' \in \psi(x)} s', t)\} \cup \{(s', t') \mid s' \in \psi(x) \wedge \Gamma, x : s' \vdash e : t'\}\\ \Gamma, x : s \vdash e \triangleright \psi \qquad \Gamma, x : s \setminus \bigvee_{s' \in \psi(x)} s' \vdash e : t\end{array}}{\Gamma \vdash \lambda x{:}s.e{:}\ \bigwedge_{(s',t') \in T} s' \to t'}$$

instead of the simpler [AbsInf+]. The difference w.r.t. [AbsInf+] is that the typing of the body is made under the hypothesis $x : s \setminus \bigvee_{s' \in \psi(x)} s'$, that is, the domain of the function minus all the input types determined by the ψ-analysis. This yields an even better refinement of the function type that

makes a difference for instance with the inference for the function `xor_` (Code 3 in Table 1): the old rule would have returned a less precise type. The rule above is defined for functions annotated by a single arrow type: the extension to annotations with intersections of multiple arrows is similar to the one we did in the simpler setting of Section 3.2.

C A ROADMAP TO POLYMORPHIC TYPES

Extending our work to the case of a polymorphic language is far from trivial. Let us go back to our typical example expression (3) of the introduction:

$$(x_1x_2 \in t)\,?e_1 : e_2 \tag{40}$$

we have seen that occurrence typing for x_1 and x_2 was possible only in very specific cases which depended on the form of the type of x_1: (1) the type of x_2 may be specialized only if the type of x_1 is an intersection of arrows and (2) the type of x_1 may be specialized only if the type of x_1 is a union of arrows. With polymorphic types the first assertion is, strictly speaking, no longer true. The simplest possible example to show it, is when x_1 is of type $\alpha \to \alpha$ and x_2 is of type α. Then it is clear that we can assume that x_2 has type t when typing e_1 and that it has type $\neg t$ when typing e_2.

The deduction becomes much more difficult when one adds subtyping and set-theoretic types to the game. Let us consider a couple of more examples to finger the key cases.

Take again the expression in (40) and imagine that t is Int and that x_1 has type $\alpha \to (\alpha \vee \mathsf{Bool})$.[12] We suppose the expression x_1x_2 to be well-typed and therefore that x_2 is typed by a subtype of α, say, $\alpha \wedge t_\circ$. The case for the "then" is not very different from the previous one when x_1 had type $\alpha \to \alpha$: the application x_1x_2 has type $\alpha \vee \mathsf{Bool}$ so if the test $x_1x_2 \in \mathsf{Int}$ succeeds, then it is because the value yielded by the application is an integer and this integer must come from the α summand of the union. Since we do not know exactly which integer we may obtain, we include all of them in α, which yields Int as the best possible approximation for α. So when typing e_1 we can safely assume that x_2 has type Int —or, more precisely, $\mathsf{Int} \wedge t_\circ$, since we use the static type information about x_2—. The case for e_2, instead, is different since assuming that x_2 has type $\neg\mathsf{Int}$ (more precisely, $\neg\mathsf{Int} \wedge t_\circ$) would be unsound, insofar as the check may fail because the application returned a Boolean, which can happen even when x_2 is bound to an integer. Therefore, when typing e_2 we cannot specialize the type of x_2 which must thus be assumed to be $\alpha \wedge t_\circ$.

For a concrete example of why this would be unsound take $t_\circ = \mathsf{Int} \vee \mathsf{Bool}$ and consider

$$(x_1x_2 \in \mathsf{Int})\,?x_2 + 1 : \mathtt{not}(x_2)$$

with $x_1 : \alpha \to (\alpha \vee \mathsf{Bool})$ and $x_2 : \alpha \wedge (\mathsf{Int} \vee \mathsf{Bool})$. If when typing the else branch we assume that x_2 has type $\neg\mathsf{Int} \wedge t_\circ$, that is, Bool, then the above expression would be well typed. But at run time x_1 can be bound to the constant function that always returns `true`, $\lambda^{\alpha\to\mathsf{Bool}}x.\mathtt{true}$ (which by subsumption is of type $\alpha \to (\alpha \vee \mathsf{Bool})$) and x_2 to an integer, say, `42` which would reduce to the expression `not(42)`, which is not well typed.

As a final example consider the case in which $x_1 : \alpha \to (\alpha \vee \mathtt{true})$, $x_2 : \alpha \wedge t_\circ$, and $t = \mathsf{Bool}$. This case is somehow the dual of the previous one. For the case "then" we cannot do any further assumption on the type of x_2, since the test may have succeeded because the application returned `true` and this may happen independently from the type of x_2. For the "else" case instead we can safely assume that the check failed due to the α part of the result, and therefore when typing e_2 we can safely assume that x_2 has type $x_2 : \neg\mathsf{Bool} \wedge t_\circ$.

[12] A non-trivial example of an expression of this type is the function $\lambda^{\alpha\to(\alpha\vee\mathsf{Bool})}x.(e\in u)\,?\,x : \mathtt{true}$.

The reason why polymorphism makes a difference is that, intuitively, a polymorphic function type already is an intersection of arrows, insofar as from an observational point of view it is equivalent to the infinite intersection of all its instances. Since we cannot work with infinitely many instances, we will pick up those that give us the information we need for occurrence typing and that are computed starting from the type of x_1 and from the type t as we explain next.

The idea is to single out the two most general type substitutions for which some test may succeed and fail, respectively, and apply these substitutions to refine the types of the corresponding occurrences when typing the "then" and "else" branches.

Consider:

- $x_1 : s \to t$
- $x_2 : u$ with $u \leq s$
- the test $x_1 x_2 \in \tau$ where τ is a closed type.

Then we proceed as follows for the THEN branch:

First, check whether $\exists \sigma$ such that $t\sigma \leq \neg\tau$.

- If such a σ does not exist, then this means that for all possible assignments of polymorphic type variables of $s \to t$, the test may succeed. Therefore the success of the test does not depend on the particular instance of $s \to t$ and, thus, it is not possible to pick some substitution that differentiates the success of the test and that could specialize the type of x_2 in the "then" branch.
- If $\exists \sigma$ such that $t\sigma \leq \neg\tau$, then we know that there is at least one assignment for the type variables of $s \to t$ that ensures that the test cannot but fail. Therefore for the typing of the branch "then" we want to exclude all such substitutions. To put it otherwise we want only to pick the substitutions σ for which the intersection of τ and $t\sigma$ is not empty, that is, that there is a value in τ that may be the result of the application. These are infinitely many substitutions (if τ contains infinitely many values), and since we do not know which one will be used (in the case of success, we just know that at least one of them will be used but not which one), then we have to take all of them. Therefore we approximate them with the substitution that ensures that all values in τ may be a result of the application. That is
  (1) Find whether $\exists \sigma_\circ$ such that $\tau \leq t\sigma_\circ$
  (2) Specialize for the "then" branch, the type of x_1 and of x_2 by applying the substitution $\sigma_\circ$ to them.

For the ELSE branch we proceed as the above but considering the test $x_1 x_2 \in \neg\tau$.

In summary the algorithm is defined as follows:

THEN: $\exists \sigma$ such that $t\sigma \leq \neg\tau$?
  no: no specialization is possible
  yes: find $\sigma_\circ$ such that $\tau \leq t\sigma_\circ$ and refine in the "then" branch the type of x_2 as $u\sigma_\circ$, of x_1 as $s\sigma_\circ \to t\sigma_\circ$, and of $x_1 x_2$ as $t\sigma_\circ$.
ELSE: $\exists \sigma$ such that $t\sigma \leq \tau$?
  no: no specialization is possible
  yes: find $\sigma_\circ$ such that $\neg\tau \leq t\sigma_\circ$ and refine in the "else" branch the type of x_2 as $u\sigma_\circ$, of x_1 as $s\sigma_\circ \to t\sigma_\circ$, and of $x_1 x_2$ as $t\sigma_\circ$.

Notice that on the given examples the algorithm returns the expected results .

All the discussion we did above holds only when the type variables at issue are so-called *monomorphic* type variables. These are variables that are bound somewhere else and all occurrences of which will be all instantiated with the same type. For instance if x_2 is the polymorphic identity

function we want to still use it polymorphically in the branches and specialize its type:

$$\begin{array}{ll}\texttt{let} & x_2 \texttt{ = } \lambda x.x \\ \texttt{in} & ((x_2 x_2 \in \mathsf{Int}))\,?\,x_2(\texttt{true}) : (x_2 x_2)\texttt{false}\end{array}$$

rather than specializing it to either Int or Int $\to$ Int (or, worse since unsound, to their intersection).

In other terms we want to enrich occurrence typing by instantiating types in a particular way, only if we know that those types will be instantiated all in the same way. For that we have to syntactically distinguish polymorphic functions from monomorphic ones, so as to deduce that x_2 in the example above as type $\forall\alpha.\alpha \to \alpha$, rather than $\alpha \to \alpha$.