Science of Computer Programming 217 (2022) 102781

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Revisiting occurrence typing

Giuseppe Castagna[a,*], Victor Lanvin[a], Mickaël Laurent[a], Kim Nguyen[b]

[a] Institut de Recherche en Informatique Fondamentale (IRIF), CNRS - Université de Paris, France
[b] Laboratoire de Méthodes Formelles (LMF), CNRS - Université Paris-Saclay, France

ARTICLE INFO

Article history:
Received 11 May 2021
Received in revised form 7 February 2022
Accepted 8 February 2022
Available online 11 February 2022

Keywords:
Union types
Intersection types
Type inference
Dynamic languages
Type-cases

ABSTRACT

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 framework that encompasses and generalizes several aspects of current occurrence typing proposals and that can be applied to tackle other problems such as the reconstruction of intersection types for unannotated or partially annotated functions and the optimization of the compilation of gradually typed languages.

1. Introduction

TypeScript [31] and Flow [17] are extensions of JavaScript that allow the programmer to specify in the code type annotations used to statically type-check the program. For instance, the following function definition is valid in both languages

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

Apart from the type annotation (in red) of the function parameter, the above is standard JavaScript code defining a function that checks whether its argument is an integer; if it is so, then it returns the argument's successor (`x+1`), otherwise it calls the method `trim()` of the argument. The annotation specifies that the parameter is either a number or a string (the vertical bar denotes a union type). If this annotation is respected and the function is applied to either an integer or a string, then the application cannot fail because of a type error (`trim()` is a string method of the ECMAScript 5 standard that trims white-spaces 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

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

https://doi.org/10.1016/j.scico.2022.102781

refine—the type of the occurrences of x in the branches of the conditional: since the program tested that x is of type number, then we can safely assume that x is of type number in the "then" branch, and that it is *not* of type number (and thus deduce from the type annotation that it must be of type string) in the "else" branch.

Occurrence typing was first defined and formally studied by Tobin-Hochstadt and Felleisen [42] to statically type-check untyped Scheme programs,[2] and later extended by Tobin-Hochstadt and Felleisen [43] 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 [42] 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 $\texttt{x}+\texttt{1}$ evaluates to an integer and x.trim() to a string, then our function foo has type $(\texttt{number}|\texttt{string})\to(\texttt{number}|\texttt{string})$. But it is clear that a more precise type would be one that states that foo returns a number when it is applied to a number and returns a string when it is applied to a string, so that the type deduced for, say, foo(42) would be number rather than number|string. This is exactly what the *intersection type*

$$(\texttt{number}\to\texttt{number})\ \ \&\ \ (\texttt{string}\to\texttt{string}) \tag{2}$$

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

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

For what concerns negation types, they are pervasive in the occurrence typing approach, even though they are used only at meta-theoretic level,[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., $(\texttt{number}|\texttt{string})\&\neg\texttt{number}$ which is equivalent to the set-theoretic difference $(\texttt{number}|\texttt{string})\setminus\texttt{number}$ 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 two important 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–i.e., any expression whatever form it has—bypassing usual type inference. Second, and most importantly, 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 (3): to put it simply, we are able to infer the type (2) for the unannotated pure JavaScript code of foo (i.e., no type annotation at all), while in TypeScript and Flow (and any other formalism we are aware of) this requires an explicit and full type annotation as the one given in (3).

Finally, the natural target for occurrence typing are languages with dynamic type tests, in particular, dynamic languages. To type such languages occurrence typing is often combined not only, as discussed above, with set-theoretic types, but also with extensible record types (to type objects) and gradual type system (to combine static and dynamic typing) two features that we study in Section 3 as two extensions of our core formalism. Of particular interest is the latter. Greenberg [21] singles out occurrence typing and gradual typing as *the* two "lineages" that partition the research on combining static and dynamic typing: he identifies the former as the "pragmatic, implementation-oriented dynamic-first" lineage and the latter as the "formal, type-theoretic, static-first" lineage. Here we demonstrate that these two "lineages" are not orthogonal or mutually independent, and we combine occurrence and gradual typing showing, in particular, how the former can be used to optimize the compilation of the latter.

[2] According to Sam Tobin-Hochstadt, the terminology *occurrence typing* was first used in a simplistic form by Komondoor et al. [27], although he and Felleisen were not aware of it the at the moment of the writing of [42].

[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.

1.1. Motivating examples

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` in (1) and (3) is rendered as $(x \in \mathsf{Int})\,?\,x+1 : (\mathsf{trim}\ x)$). In particular, 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

$$(x_1 x_2 \in t)\,?\,e_1 : e_2 \tag{4}$$

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 (3). 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 if $x_2 : \mathsf{Int}{\vee}\mathsf{String}$, then the expression[4]

$$\texttt{let}\ x_1 = \texttt{foo}\ \texttt{in}\ (x_1x_2 \in \mathsf{Int})\,?\,((x_1x_2)+x_2) : 42 \tag{5}$$

is well typed with type Int: when typing the branch “then” we know that the test $x_1x_2 \in \mathsf{Int}$ succeeded 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 (5) 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

$$(x_1x_2 \in \mathsf{Int})\,?\,((x_1x_2)+x_2) : ((x_1x_2)\,\texttt{@}\,x_2) \tag{6}$$

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 (6) has type $\mathsf{Int} \vee \mathsf{String}$. Let us see this in detail. The expression in (6) 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, then, is 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 (i.e., the type `any` of TypeScript and Flow) 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 the function 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 (e.g., when the function bound to x_1 is the result of a branching expression), then the test may give us information about the type of the function in the various branches. So for instance if in the expression in (4) x_1 is of type, say, $(s_1 \to t) \vee (s_2 \to \neg t)$, then we can assume for the expression (4) 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

$$(x_1x_2 \in \mathsf{Int})\,?\,(x_1(x_1x_2)+42) : \texttt{not}(x_1(x_1x_2)) \tag{7}$$

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

[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 (12).

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 presented next.

1.2. 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 $e_1 e_2$ 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 $e_1 e_2$ 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 λ-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 $\mathsf{dom}()$ 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 (3), 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 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 2.6.1 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 $e_1 e_2$ 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

$$t_1 \circ s \leq \neg t \tag{8}$$

In other terms, s contains all the legal 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 (8). 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 $t_1 \blacksquare t$. When the test fails, the type of e_2 can be refined in a similar way just by replacing t by $\neg t$: we get the refined type $t_2 \wedge (t_1 \blacksquare \neg t)$. 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 $t_2^+ \overset{\text{def}}{=} t_2 \wedge (t_1 \blacksquare t)$ in the "then" branch (we call it the *positive* branch) and as $t_2^- \overset{\text{def}}{=} t_2 \setminus (t_1 \blacksquare t)$ 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 returned a result in t. Therefore, we can exclude from t_1 all the functions that, when applied to an argument in t_2^+, yield a result not in t. It can be obtained simply by removing from t_1 the functions in $t_2^+ \to \neg t$, that is, we refine the type of e_1 in the "then" branch as $t_1^+ = t_1 \setminus (t_2^+ \to \neg t)$. Note that this also removes functions diverging on t_2^+ arguments. 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 $t_1 \setminus (t_2^+ \to \neg t)$ removes, among others, all functions in t_1 that diverge on t_2^+. Let us see all this on our example (7), 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 $\neg(t_2^+ \to \neg t)$, that is, $\neg(\mathsf{String} \to \neg\mathsf{Int})$. 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 (20) of the definition of the function Constr.

1.3. Technical challenges

In the previous section we outlined the main ideas of our approach to occurrence typing. However, the 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 expressions are given different types in the "then" and "else" branches of type tests. For instance, in (6) the expression x_1x_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

$$(x_1x_2 \in \mathsf{Int})\,?\ldots x_1x_2\ldots : \ldots x_1x_2\ldots \tag{9}$$

It is not possible to specialize the type of the variables in the branches. Nevertheless, we want to be able to deduce that x_1x_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 (9) we extend the current type environment with the hypothesis that the expression x_1x_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 $\mathsf{Refine}()$ 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 [44], 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

$$(e(42)\in\mathsf{Bool})\,?e : \ldots \tag{10}$$

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 (10). Now imagine that the application $e(42)$ reduces to a Boolean value, then the whole expression in (10) 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, the proof of type preservation (see Appendix A.3.2) resorts to *type schemes*, a technique introduced by Frisch et al. [19] to type expressions by sets of types, so that the expression in (10) 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:

$$((f\,x, g\,x)\in\mathsf{Int}\times\mathsf{Bool})\,?\ldots : \ldots \tag{11}$$

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 (11) 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 environments 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

$$(\,(x, y) \in ((\mathsf{Int} \vee \mathsf{Bool}) \times \mathsf{Int})\,)\;?\,e : \ldots \tag{12}$$

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 temporarily removing the type assumption about (x, y) from the type environment and by retyping the expression without that assumption (see rule [ENV$_{\mathcal{A}}$] in Section 2.6.3).

Outline

In Section 2 we formalize the ideas we just presented: 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. To ease the presentation all the proofs are omitted from the main text and can be found in the appendix.

Contributions

The main contributions of our work can be summarized as follows:

- We provide a theoretical framework to refine the type of expressions occurring in type tests, thus removing the limitations of current occurrence typing approaches which require both the tests and the refinement to take place on variables.
- We define a type-theoretic approach alternative to the current flow-based approaches. As such it provides different results and it can be thus profitably combined with flow-based techniques.
- We use our analysis for defining a formal framework that reconstructs intersection types for unannotated or partially-annotated functions, something that, in our ken, no other current system can do.
- We prove the soundness of our system. We define algorithms to infer the types that we prove to be sound and show different completeness results which in practice yield the completeness of any reasonable implementation.
- We show how to extend our approach to records with field addition, update, and deletion operations.
- We show how occurrence typing can be extended to and combined with gradual typing and apply our results to optimize the compilation of the latter.

We end this introduction by stressing the practical implications of our work: a perfunctory inspection may give the wrong impression that the only interest of the heavy formalization that follows is to have generic expressions, rather than just variables, in type cases: this would be a bad trade-off. The important point is, instead, that our formalization is what makes analyses such as those presented in Section 3 possible (e.g., the reconstruction of the type (2) for the unannotated pure JavaScript code of `foo`), which is where the actual added practical value and potential of our work resides.

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:

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

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: $t_1 \wedge t_2 \overset{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \overset{\text{def}}{=} t_1 \wedge \neg t_2$, $\mathbb{1} \overset{\text{def}}{=} \neg\mathbb{0}$. 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 $\rhd \subseteq$**Types**$\times$**Types** 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. [19] and detailed description of the algorithm to decide this relation can be found in [6]. For the reader's convenience we succinctly recall the definition of the subtyping relation in the next subsection but it is possible to skip this subsection at first reading and jump directly to Subsection 2.3, since to understand the rest of the paper it suffices to consider that types are interpreted as sets of *values* (i.e., either constants, λ-abstractions, or pairs of values: see Section 2.3 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. All the above is formalized as follows.

2.2. Subtyping

Subtyping is defined by giving a set-theoretic interpretation of the types of Definition 2.1 into a suitable domain $\mathcal{D}$:

Definition 2.2 *(Interpretation domain [19]).* The *interpretation domain* $\mathcal{D}$ is the set of finite terms d produced inductively by the following grammar

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

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

The elements of $\mathcal{D}$ correspond, intuitively, to (denotations of) the results of the evaluation of expressions. In particular, in a higher-order language, the results of computations can be functions which, in this model, are represented by sets of finite relations of the form $\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}$, where Ω (which is not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This is implemented by using in the second projection the meta-variable ∂ which ranges over $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ (we reserve d to range over $\mathcal{D}$, thus excluding Ω). This constant Ω is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used d instead of ∂, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently from its argument as long as this argument is typable (see Section 4.2 of [19] for details). The restriction to *finite* relations corresponds to the intuition that the denotational semantics of a function is given by the set of its finite approximations, where finiteness is a restriction necessary (for cardinality reasons) to give the semantics to higher-order functions.

We define the interpretation $[\![t]\!]$ of a type t so that it satisfies the following equalities, where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets and $\mathbb{B}$ denotes the function that assigns to each basic type the set of constants of that type, so that for every constant c we have $c \in \mathbb{B}(\boldsymbol{b}_c)$ (we use $\boldsymbol{b}_c$ to denote the basic type of the constant c):

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

[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.

We cannot take the equations above directly as an inductive definition of $[\![]\!]$ because types are not defined inductively but coinductively. However, recall that the contractivity condition of Definition 2.1 ensures that the binary relation $\rhd \subseteq$ **Types**$\times$**Types** defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian which gives an induction principle on **Types** that we use combined with structural induction on $\mathcal{D}$ to give the following definition which validates these equalities.

Definition 2.3 *(Set-theoretic interpretation of types [19]).* We define a binary predicate $(d : t)$ ("the element d belongs to the type t"), where $d \in \mathcal{D}$ and $t \in$ **Types**, by induction on the pair (d, t) ordered lexicographically. The predicate is defined as follows:

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

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

Finally, we define the subtyping preorder and its associated equivalence relation as follows.

Definition 2.4 *(Subtyping relation [19]).* We define the *subtyping* relation $\leq$ and the *subtyping equivalence* relation $\simeq$ as $t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ and $t_1 \simeq t_2 \overset{\text{def}}{\iff} (t_1 \leq t_2)$ and $(t_2 \leq t_1)$.

2.3. Syntax

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

$$\begin{array}{lll}
\textbf{Expr} & e & ::= c \mid x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid \pi_j e \mid (e, e) \mid (e{\in}t)\,?\,e : e\\
\textbf{Values} & v & ::= c \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid (v, v)
\end{array} \tag{13}$$

for $j = 1, 2$. In (13), c ranges over constants (e.g., `true`, `false`, `1`, `2`, ...) which are values of basic types; x ranges over variables; (e, e) denotes 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) has the type t or not; $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ denotes the function of parameter x and body e annotated with the type $\wedge_{i\in I} s_i \to t_i$. 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 expression 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 $i \in I$.

2.4. Dynamic semantics

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

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

where $[\![t]\!]_{\mathcal{V}}$ denotes, intuitively, the set of values that have type t. Formally, $[\![t]\!]_{\mathcal{V}} = \{v \mid \exists t' \in \text{typeof}_{\mathcal{V}}(v).\ t' \leq t\}$ where $\text{typeof}_{\mathcal{V}}(v)$ is inductively defined as: $\text{typeof}_{\mathcal{V}}(c) \overset{\text{def}}{=} \{\boldsymbol{b}_c\}$, $\text{typeof}_{\mathcal{V}}(\lambda^{\wedge_{i\in I} s_i \to t_i} x.e) \overset{\text{def}}{=} \{t \mid t \simeq (\wedge_{i\in I} s_i \to t_i) \wedge (\wedge_{j\in J} s'_j \to t'_j), t \not\leq \mathbb{0}\}$, $\text{typeof}_{\mathcal{V}}((v_1, v_2)) \overset{\text{def}}{=} \text{typeof}_{\mathcal{V}}(v_1) \times \text{typeof}_{\mathcal{V}}(v_2)$.[7]

Contextual reductions are defined by the following evaluation contexts:

$$\mathcal{C}[\,] ::= [\,] \mid \mathcal{C}e \mid v\mathcal{C} \mid (\mathcal{C}, e) \mid (v, \mathcal{C}) \mid \pi_i\mathcal{C} \mid (\mathcal{C}{\in}t)\,?\,e : e$$

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

[7] This definition may look complicated but it is necessary to handle some corner cases for negated arrow types (cf. rule [Abs-] in Section 2.5). For instance, it states that $\lambda^{\text{Int}\to\text{Int}} x.x \in [\![(\text{Int}\to\text{Int}) \wedge \neg(\text{Bool}\to\text{Int})]\!]_{\mathcal{V}}$.

2.5. Static semantics

While the syntax and reduction semantics are, on the whole, pretty standard, for what concerns the type system we will have to introduce several unconventional features that we anticipated in Section 1.3 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 2.3. Just notice subtyping is embedded in the system by the classic [Subs] subsumption rule. Next we focus on the unconventional aspects of our system, from the simplest to the hardest.

The first unconventional aspect is that, as explained in Section 1.3, 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:

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

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:

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

In Section 1.3 we explained that in order for our system to satisfy the property of type preservation, the type system must be able to deduce negated arrow types for functions—e.g. the type $(\mathsf{Int} \to \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ for $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.x$. We demonstrated this with the expression in equation (10), for which type preservation holds only if we are able to deduce for this expression the type $(\mathsf{Int} \to t) \setminus (\mathsf{Int} \to \neg\mathsf{Bool})$, that is, $(\mathsf{Int} \to t) \wedge \neg(\mathsf{Int} \to \neg\mathsf{Bool})$. But the sole rule [Abs+] above does not allow us to deduce negations of arrows for λ-abstractions: the rule [Abs-] makes this possible. This rule ensures that given a function $\lambda^t x.e$ (where t is an intersection type), for every type $t_1 \to t_2$, either $t_1 \to t_2$ can be obtained by subsumption from t or $\neg(t_1 \to t_2)$ can be added to the intersection t. In turn this ensures that, for any function and any type t either the function has type t or it has type $\neg t$ (see Petrucciani [34, Sections 3.3.2 and 3.3.3] for a thorough discussion on this rule). As an aside, note that this kind of deduction is already present in the system by Frisch et al. [19] 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. [19] 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:

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

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 $\Gamma\vdash^{\texttt{Env}}_{e,(\neg)t}\Gamma_i$. The intuition is that when $\Gamma\vdash^{\texttt{Env}}_{e,t}\Gamma_1$ 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, $\Gamma\vdash^{\texttt{Env}}_{e,\neg t}\Gamma_2$ (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 $\Gamma\vdash^{\texttt{Env}}_{e,t}\Gamma'$. 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 $i=0,1$, and undefined otherwise)

$$\begin{array}{rclcrclcrcl}
e{\downarrow}\epsilon &=& e & \quad & (e_1,e_2){\downarrow}l.\varpi &=& e_1{\downarrow}\varpi & \quad & \pi_1 e{\downarrow}f.\varpi &=& e{\downarrow}\varpi\\
e_0\,e_1{\downarrow}i.\varpi &=& e_i{\downarrow}\varpi & & (e_1,e_2){\downarrow}r.\varpi &=& e_2{\downarrow}\varpi & & \pi_2 e{\downarrow}s.\varpi &=& e{\downarrow}\varpi
\end{array}$$

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 $\Gamma\vdash^{\texttt{Env}}_{e,t}\Gamma'$ are then deduced by the following two rules:

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

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 $\vdash^{\texttt{Path}}_{\Gamma,e,t}\varpi:t'$ 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 2.6.2). The last ingredient for our type system is the deduction of the judgements of the form $\vdash^{\texttt{Path}}_{\Gamma,e,t}\varpi:t'$ where ϖ is a path to an expression occurring in e. This is given by the following set of rules.

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

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

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

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

These rules implement the analysis described in Section 1.2 for functions and extend it to products. Let us comment each rule in detail. [PSubs] is just subsumption for the deduction $\vdash^{\texttt{Path}}$. 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 $\vdash^{\texttt{Path}}$ 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. [PAppL] 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 all the rules of our type system (they are summarized for the reader's convenience in Appendix A.1), which satisfies the property of safety, deduced, as customary, from the properties of progress and subject reduction (*cf.* Appendix A.3).

Theorem 2.5 *(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 $e \leadsto^* v$.*

2.6. Algorithmic system

The type 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t}$ for paths.

(i). Multiple types have two distinct origins each requiring a distinct technical solution. The first origin is the presence of structural rules[8] such as [SUBS] and [INTER]. We handle this presence in the classic way: we define an algorithmic system that tracks the minimum type 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 2.6.1. The second 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 (finitely) many negations of arrow types as possible without making the type empty. We do not handle this multiplicity directly in the algorithmic system but only in the proof of its soundness by using and adapting the technique of *type schemes* defined by Frisch et al. [19]. Type schemes are canonical representations of the infinite sets of types of λ-abstractions which can be used to define an algorithmic system that can be easily proved to be sound. The simpler algorithm that we propose in this section implies (i.e., it is less precise than) the one with type schemes (*cf.* Lemma Appendix B.20) and it is thus sound, too. The algorithm of this section is not only simpler but, as we discuss in Section 2.6.4, is also the one that should be used in practice. This is why we preferred to present it here and relegate the presentation of the system with type schemes to Appendix B.2.1.

(ii). For what concerns the use of the auxiliary derivation for the $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$ and $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : t'$ judgments, we present in Section 2.6.2 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 2.6.3.

2.6.1. 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 1.2. Consider the classic rule [APP] for applications. It essentially does three things: (i) it checks that the expression in the function position has a functional type; (ii) it checks that the argument is in the domain of the function, and (iii) it returns the type of the application. In systems without set-theoretic types these operations are quite straightforward: (i) corresponds to checking that the expression has an arrow type, (ii) corresponds to checking that the argument is in the domain of the arrow deduced for the function, and (iii) corresponds to returning the codomain of that 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 $\mathsf{dom}()$ and $\circ$ we informally described in Section 1.2 where we also introduced the operator $\blacksquare$. These three operators are used by our algorithm and formally defined as:

$$\mathsf{dom}(t) = \max\{u \mid t \leq u \to \mathbb{1}\} \tag{14}$$

$$t \circ s = \min\{u \mid t \leq s \to u\} \tag{15}$$

$$t \blacksquare s = \min\{u \mid t \circ (\mathsf{dom}(t) \setminus u) \leq \neg s\} \tag{16}$$

In short, $\mathsf{dom}(t)$ is the largest domain of any single arrow that subsumes t, $t \circ s$ is the smallest codomain of an arrow type that subsumes t and has domain s and $t \blacksquare s$ was explained before.

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:

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

$$\boldsymbol{\pi_1}(t) \;=\; \min\{u \mid t \leq u \times \mathbb{1}\} \qquad \boldsymbol{\pi_2}(t) \;=\; \min\{u \mid t \leq \mathbb{1} \times u\} \tag{17}$$

All the operators above but $\blacksquare$ are already present in the theory of semantic subtyping: the reader can find how to compute them in [19, Section 6.11] (see also [6, §4.4] for a detailed description). Below we just show our new 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:

$$t \blacksquare s = \mathsf{dom}(t) \wedge \bigvee_{i\in I}\left(\bigwedge_{\{P\subseteq P_i \mid s \leq \bigvee_{p\in P}\neg t_p\}}\left(\bigvee_{p\in P}\neg s_p\right)\right) \tag{18}$$

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 (16) is given in the Appendix B.1.

2.6.2. Type environments for occurrence typing

The second ingredient necessary to the definition of our algorithmic systems is the algorithm for the deduction of $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$, 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 deduced for e under the type environment Γ in the algorithmic type system of Section 2.6.3. That is, $\mathsf{typeof}_\Gamma(e) = t$ if and only if $\Gamma \vdash_{\mathcal{A}} e : t$ is provable.

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

$$\mathsf{Constr}_{\Gamma,e,t}(\epsilon) = t \tag{19}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.0) = \neg(\mathsf{Intertype}_{\Gamma,e,t}(\varpi.1) \to \neg\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{20}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.1) = \mathsf{typeof}_\Gamma(e{\downarrow}\varpi.0) \blacksquare \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \tag{21}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.l) = \boldsymbol{\pi_1}(\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{22}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.r) = \boldsymbol{\pi_2}(\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{23}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.f) = \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \times \mathbb{1} \tag{24}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.s) = \mathbb{1} \times \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \tag{25}$$

$$\mathsf{Intertype}_{\Gamma,e,t}(\varpi) = \mathsf{Constr}_{\Gamma,e,t}(\varpi) \wedge \mathsf{typeof}_\Gamma(e{\downarrow}\varpi) \tag{26}$$

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).[9]

Each case of the definition of the Constr function corresponds to the application of a logical rule (*cf.* definition in Footnote 8) in the deduction system for $\vdash^{\texttt{Path}}$: case (19) corresponds to the application of [PEps]; case (20) implements [PAppl] straightforwardly; the implementation of rule [PAppR] is subtler: instead of finding the best t_1 to subtract (by intersection) from the static type of the argument, (21) finds directly the best type for the argument by applying the $\blacksquare$ operator to the static type of the function and the refined type of the application. The remaining (22)–(25) cases are the straightforward implementations of the rules [PPairL], [PPairR], [PFst], and [PSnd], respectively.

The other recursive function, $\mathsf{Intertype}$, 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 $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$. 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 $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$ 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 1.3 we said that we want our analysis to be able to capture all the information available from nested checks. If we

[9] Note that the definition is well-founded. This can be seen by analyzing the rule [Case$_{\mathcal{A}}$] of Section 2.6.3: 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 $\Gamma \vdash e : t_0$ (and this is possible only if we were able to deduce under the hypothesis Γ the type of every occurrence of e).

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 $\vdash^{\texttt{Path}}$ (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 1.3 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 $(xx{\in}\mathbb{1})\,?\,e_1 : e_2$. If $x : \mathbb{1}\to\mathbb{1}$, then when refining the “then” branch, every iteration of Refine yields for x a type strictly more precise than the type deduced in the previous iteration (because of the $\varpi.0$ case).

The solution we adopt in practice is to bound the number of iterations to some number n_o. This is obtained by the following definition of Refine

$$\begin{array}{r}\mathsf{Refine}_{e,t} \overset{\text{def}}{=} (\mathsf{RefineStep}_{e,t})^{n_o}\\ \text{where }\mathsf{RefineStep}_{e,t}(\Gamma)(e') = \end{array}\left\{\begin{array}{ll}\bigwedge_{\{\varpi \mid e\downarrow\varpi\equiv e'\}}\mathsf{Intertype}_{\Gamma,e,t}(\varpi) & \text{if } \exists\varpi.\ e\downarrow\varpi\equiv e'\\ \Gamma(e') & \text{otherwise, if } e'\in\mathsf{dom}(\Gamma)\\ \text{undefined} & \text{otherwise}\end{array}\right.$$

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, but also we had to build the expression that causes the divergence in quite an *ad hoc* way which makes divergence even more unlikely: setting an n_o twice the depth of the syntax tree of the outermost type case should be more than enough to capture all realistic cases. For instance, all examples given in Section 4 can be checked (or found to be ill-typed) with $n_o = 1$.

2.6.3. Algorithmic typing rules

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

$$[\textsc{Efq}_{\mathcal{A}}]\ \frac{}{\Gamma,(e:\mathbb{0})\vdash_{\mathcal{A}} e':\mathbb{0}}\ \text{WITH PRIORITY OVER ALL THE OTHER RULES}\qquad [\textsc{Var}_{\mathcal{A}}]\ \frac{}{\Gamma\vdash_{\mathcal{A}} x:\Gamma(x)}\ x\in\mathsf{dom}(\Gamma)$$

$$[\textsc{Env}_{\mathcal{A}}]\ \frac{\Gamma\setminus\{e\}\vdash_{\mathcal{A}} e:t}{\Gamma\vdash_{\mathcal{A}} e:\Gamma(e)\wedge t}\ \begin{array}{l}e\in\mathsf{dom}(\Gamma)\ \text{AND}\\ e\ \text{NOT A VARIABLE}\end{array}\qquad [\textsc{Const}_{\mathcal{A}}]\ \frac{}{\Gamma\vdash_{\mathcal{A}} c:\boldsymbol{b}_c}\ c\notin\mathsf{dom}(\Gamma)$$

$$[\textsc{Abs}_{\mathcal{A}}]\ \frac{\Gamma,x:s_i\vdash_{\mathcal{A}} e:t_i'\qquad t_i'\leq t_i}{\Gamma\vdash_{\mathcal{A}}\lambda^{\wedge_{i\in I}s_i\to t_i}x.e:\wedge_{i\in I}s_i\to t_i}\ \lambda^{\wedge_{i\in I}s_i\to t_i}x.e\notin\mathsf{dom}(\Gamma)$$

$$[\textsc{App}_{\mathcal{A}}]\ \frac{\Gamma\vdash_{\mathcal{A}} e_1:t_1\qquad \Gamma\vdash_{\mathcal{A}} e_2:t_2\qquad t_1\leq\mathbb{0}\to\mathbb{1}\qquad t_2\leq\mathsf{dom}(t_1)}{\Gamma\vdash_{\mathcal{A}} e_1e_2:t_1\circ t_2}\ e_1e_2\notin\mathsf{dom}(\Gamma)$$

$$[\textsc{Case}_{\mathcal{A}}]\ \frac{\Gamma\vdash_{\mathcal{A}} e:t_0\qquad \mathsf{Refine}_{e,t}(\Gamma)\vdash_{\mathcal{A}} e_1:t_1\qquad \mathsf{Refine}_{e,\neg t}(\Gamma)\vdash_{\mathcal{A}} e_2:t_2}{\Gamma\vdash_{\mathcal{A}}(e{\in}t)\,?\,e_1 : e_2:t_1\vee t_2}\ (e{\in}t)\,?e_1:e_2\notin\mathsf{dom}(\Gamma)$$

$$[\textsc{Proj}_{\mathcal{A}}]\ \frac{\Gamma\vdash_{\mathcal{A}} e:t\qquad t\leq\mathbb{1}\times\mathbb{1}}{\Gamma\vdash_{\mathcal{A}}\pi_i e:\boldsymbol{\pi}_{\mathbf{i}}(t)}\ \pi_i e{\notin}\mathsf{dom}(\Gamma)\qquad [\textsc{Pair}_{\mathcal{A}}]\ \frac{\Gamma\vdash_{\mathcal{A}} e_1:t_1\qquad \Gamma\vdash_{\mathcal{A}} e_2:t_2}{\Gamma\vdash_{\mathcal{A}}(e_1,e_2):t_1\times t_2}\ (e_1,e_2){\notin}\mathsf{dom}(\Gamma)$$

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. 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 subtyping in the elimination rules [$\textsc{App}_{\mathcal{A}}$,$\textsc{Proj}_{\mathcal{A}}$]. In particular, for [$\textsc{App}_{\mathcal{A}}$] 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 (partially) replaced by the rule [$\textsc{Env}_{\mathcal{A}}$] which 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 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. Finally, notice that there is no counterpart for the rule [ABS-] and that therefore it is not possible to deduce negated arrow types for functions. This means that the algorithmic system is not complete as we discuss in details in the next section.

2.6.4. Properties of the algorithmic system

In what follow we will use $\Gamma \vdash^{n_o}_{\mathcal{A}} e : t$ to stress the fact that the judgment $\Gamma \vdash_{\mathcal{A}} e : t$ is provable in the algorithmic system where $\mathsf{Refine}_{e,t}$ is defined as $(\mathsf{RefineStep}_{e,t})^{n_o}$; we will omit the index n_o—thus keeping it implicit—whenever it does not matter in the context.

The algorithmic system above is sound with respect to the deductive one of Section 2.5

Theorem 2.6 *(Soundness). For every Γ, e, t, n_o, if $\Gamma \vdash^{n_o}_{\mathcal{A}} e : t$, then $\Gamma \vdash e : t$.*

The proof of this theorem (see Appendix B.5) is obtained by defining an algorithmic system $\vdash_{\mathcal{A}_{\text{ts}}}$ that uses type schemes, that is, which associates each typable term e with a possibly infinite set of types $\mathbb{t}$ (in particular a λ-expression $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ will be associated to a set of types of the form $\{s \mid \exists s_0 = \bigwedge_{i=1..n} t_i \to s_i \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j).\ \mathbb{0} \not\simeq s_0 \leq s\})$ and proving that, if $\Gamma \vdash_{\mathcal{A}} e : t$ then $\Gamma \vdash_{\mathcal{A}_{\text{ts}}} e : \mathbb{t}$ with $t \in \mathbb{t}$: the soundness of $\vdash_{\mathcal{A}}$ follows from the soundness of $\vdash_{\mathcal{A}_{\text{ts}}}$.

Completeness needs a more detailed explanation. The algorithmic system $\vdash_{\mathcal{A}}$ is not complete w.r.t. the language presented in Section 2.3 because it cannot deduce negated arrow types for functions. However, no practical programming language with structural subtyping would implement the full language of Section 2.3, but rather restrict all expressions of the form $(e{\in}t)\ ?\ e_1 : e_2$ so that the type t tested in them is either non functional (e.g., products, integer, a record type, etc.) or it is $\mathbb{0} \to \mathbb{1}$ (i.e., the expression can just test whether e returns a function or not).[10] There are multiple reasons to impose such a restriction, the most important ones can be summarized as follows:

1. For explicitly-typed languages it may yield conterintutive results, since for instance $\lambda^{\mathsf{Int}\to\mathsf{Int}} x.x \in \mathsf{Bool} \to \mathsf{Bool}$ should fail despite the fact that identity functions maps Booleans to Booleans.
2. For implicitly-typed languages it yields a semantics that depends on the inference algorithm, since $(\lambda y.(\lambda x.y))3 \in 3{\to}3$ may either fail or not according to whether the type deduced for the result of the expression is either $\mathsf{Int}{\to}\mathsf{Int}$ or $3{\to}3$ (which are both valid but incomparable).
3. For gradually-typed languages it would yield a problematic system as we explain in Section 3.3.

Now, if we apply this restriction to the language of Section 2.3, then the algorithmic system of section 2.6.3 is complete. Let say that an expression e is *positive* if it never tests a functional type more precise than $\mathbb{0} \to \mathbb{1}$ (see Appendix B.5 for the formal definition). Then we have:

Theorem 2.7 *(Completeness for positive expressions). For every type environment Γ and* positive *expression e, if $\Gamma \vdash e : t$, then there exist n_o and t' such that $\Gamma \vdash^{n_o}_{\mathcal{A}} e : t'$.*

We can use the algorithmic system $\vdash_{\mathcal{A}_{\text{ts}}}$ defined for the proof of Theorem 2.6 to give a far more precise characterization than the above of the terms for which our algorithm is complete: positivity is a practical but rough approximation. The system $\vdash_{\mathcal{A}_{\text{ts}}}$ copes with negated arrow types, but it still is not complete essentially for two reasons: (i) the recursive nature of rule [PATH] and (ii) the use of nested [PAPPL] that yields a precision that the algorithm loses by using type schemes in defining of Constr (case (20) 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 *(Rank-0 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 $\Gamma \vdash^{n_o}_{\mathcal{A}_{ts}} e : t'$ and $t' \leq t$.*

This last result is only of theoretical interest since, in practice, we expect to have only languages with positive expressions. This is why for our implementation we use the library of CDuce [3] in which type schemes are absent and functions are

[10] Of course, there exist languages in which it is possible to check whether some value has a type that has functional subcomponents—e.g., to test whether an object is of some class that possesses some given methods, but that is a case of nominal rather than structural subtyping, which in our framework corresponds to testing whether a value has some basic type.

typed only by intersections of positive arrows. We present the implementation in Section 4, but before we study some extensions.

3. Extensions

As we recalled in the introduction, the main application of occurrence typing is to type dynamic languages. In this section we explore how to extend our work to encompass three features that are necessary to type these languages.

First, we consider record types and record expressions which, in dynamic languages, are used to implement objects. In particular, we extend our system to cope with typical usage patterns of objects employed in these languages such as adding, modifying, or deleting a field, or dynamically testing its presence to specify different behaviors.

Second, in order to precisely type applications in dynamic languages it is crucial to refine the type of some functions to account for their different behaviors with specific input types. But current approaches are bad at it: they require the programmer to explicitly specify a precise intersection type for these functions and, even with such specifications, some common cases fail to type (in that case the only solution is to hard-code the function and its typing discipline into the language). We show how we can use the work developed in the previous sections to infer precise intersection types for functions. In our system, these functions do not require any type annotation or just an annotation for the function parameters, whereas some of them fail to type in current alternative approaches even when they are given the full intersection type specification.

Finally, to type dynamic languages it is often necessary to make statically-typed parts of a program coexist with dynamically-typed ones. This is the aim of gradually typed systems that we explore in the third extension of this section.

3.1. Record 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 (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. Even more, thanks to the presence of union types it is possible to type heterogeneous lists whose content is described by regular expressions on types as proposed by Hosoya et al. [22]. However, this is not enough to cover records and, in particular, the specific usage patterns in dynamic languages of records, whose field are dynamically tested, deleted, added, and modified. This is why we extend here our work to records, building on the record types as they are defined in CDuce.

The extension we present in this section is not trivial. Although we use the record *types* as they are defined in CDuce we cannot do the same for CDuce record *expressions*. The reasons why we cannot use the record expressions of CDuce and we have to define and study new ones are twofold. On the one hand we want to capture the typing of record field extension and field deletion, two operation widely used in dynamic language; on the other hand we need to have very simple expressions formed by elementary sub-expressions, in order to limit the combinatorics of occurrence typing. For this reason we build our records one field at a time, starting from the empty record and adding, updating, or deleting single fields.

Formally, CDuce record *types* can be embedded in our types by adding the following two type constructors:

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

where ℓ ranges over an infinite set of labels Labels and Undef is a special singleton type whose only value is a constant `undef` which is not in $\mathcal{D}$ (for that it is a constant akin to Ω): as a consequence Undef and $\mathbb{1}$ are distinct types, the interpretation of the former being the constant `undef` while the interpretation of the latter being the set of all the other values. 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 (as for OCaml object types), provided by the following syntactic sugar:[11]

- $\{\ell_1 = t_1, \ldots, \ell_n = t_n\}$ for $\{\ell_1 = t_1 \ldots \ell_n = t_n,\ \_ = \mathsf{Undef}\}$ (closed records).
- $\{\ell_1 = t_1, \ldots, \ell_n = t_n\ ..\}$ 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 cannot use CDuce record expressions as they are, but instead we must adapt them to our analysis. So as anticipated, we consider records that are built starting from the empty record expression $\{\}$ by adding, updating, or removing fields:

[11] 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.

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

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 three operators on record types: $t.\ell$ which returns the type of the field ℓ in the record type t, $t_1 + t_2$ which returns the record type formed by all the fields in t_2 and those in t_1 that are not in t_2, and $t\backslash\ell$ which returns the type t in which the field ℓ is undefined. They are formally defined as follows (see Frisch [18] for more details):

$$t.\ell = \begin{cases} \min\{u \mid t \leq \{\ell = u\ ..\}\} & \text{if } t \leq \{\ell = \mathbb{1}\ ..\} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{27}$$

$$t_1 + t_2 = \min\left\{ u \;\middle|\; \forall \ell \in \mathsf{Labels}.\ \begin{cases} u.\ell \geq t_2.\ell & \text{if } t_2.\ell \leq \neg\mathsf{Undef} \\ u.\ell \geq t_1.\ell \vee (t_2.\ell \setminus \mathsf{Undef}) & \text{otherwise} \end{cases} \right\} \tag{28}$$

$$t\backslash\ell = \min\left\{ u \;\middle|\; \forall \ell' \in \mathsf{Labels}.\ \begin{cases} u.\ell' \geq \mathsf{Undef} & \text{if } \ell' = \ell \\ u.\ell' \geq t.\ell' & \text{otherwise} \end{cases} \right\} \tag{29}$$

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 $\{..\}$ is the largest record type.

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

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

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

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

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

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

Deriving the algorithm from these rules is then straightforward:

$$\begin{array}{ll} \mathsf{Constr}_{\Gamma,e,t}(\varpi.a_\ell) = \{\ell : \mathsf{Intertype}_{\Gamma,e,t}(\varpi)\ ..\} & \mathsf{Constr}_{\Gamma,e,t}(\varpi.r_\ell) = (\mathsf{Intertype}_{\Gamma,e,t}(\varpi))\backslash\ell + \{\ell =?\ \mathbb{1}\} \\ \mathsf{Constr}_{\Gamma,e,t}(\varpi.u_\ell^2) = (\mathsf{Intertype}_{\Gamma,e,t}(\varpi)).\ell & \mathsf{Constr}_{\Gamma,e,t}(\varpi.u_\ell^1) = (\mathsf{Intertype}_{\Gamma,e,t}(\varpi))\backslash\ell + \{\ell =?\ \mathbb{1}\} \end{array}$$

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 $\mathsf{Constr}_{\Gamma,e,t}(\varpi.u_\ell^1)$—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 for the first definition of the function `foo` in (1) the type `(number∨string) → (number∨string)`, while the more precise type

$$\texttt{(number}\to\texttt{number)} \wedge \texttt{(string}\to\texttt{string)} \tag{30}$$

can be deduced by these languages only if they are instructed to do so: the programmer has to explicitly annotate `foo` with the type (30): we did it in (3) using Flow—the TypeScript annotation for it is much heavier. 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 (30) even in the absence the annotation given in (3). 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 in a pseudo TypeScript that uses our syntax for type-cases

```
    function (x : τ     ) {
    return (x ∈     Real) ? ((x ∈     Int) ? x+1 : sqrt(x)) : !x;
  }
```
(31)

where we assume that `Int` is a subtype of `Real`. When τ is `Real∨Bool` 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 `!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 [43] 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 propose a more general approach than the one by Tobin-Hochstadt and Felleisen [43] since we allow the programmer to hint a particular type for the argument and let the system deduce, 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 environment, 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):

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

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

$$[\textsc{Case}]\ \frac{\Gamma \vdash e \triangleright \psi_\circ \qquad \Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma_1 \qquad \Gamma_1 \vdash e \triangleright \psi_1 \qquad \Gamma_1 \vdash e_1 \triangleright \psi_1' \qquad \Gamma \vdash^{\texttt{Env}}_{e,\neg t} \Gamma_2 \qquad \Gamma_2 \vdash e \triangleright \psi_2 \qquad \Gamma_2 \vdash e_2 \triangleright \psi_2'}{\Gamma \vdash\ (e \in t)\ ?e_1 : e_2 \triangleright \psi_\circ \cup \psi_1 \cup \psi_1' \cup \psi_2 \cup \psi_2'}$$

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:

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

All that remains to do is to 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 contained in the domain of Γ restricted to variables. Simply put, this rule first collects all possible types that are deduced for a variable x during the typing of the body of the λ and then uses them to re-type the body 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 (31) 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) ? x+1 : 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:

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

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 set of input types and add the resulting arrows to the type deduced for the whole function (see Section 4 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 partition rather than on the union thereof. Of course, we could use much a 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.

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 add to our deduction system a last further rule:

$$[\textsc{OverApp}]\ \frac{\Gamma \vdash e : \bigvee \bigwedge_{i\in I} t_i \to s_i \qquad \Gamma \vdash x : t \qquad \Gamma \vdash e \triangleright \psi_1 \qquad \Gamma \vdash x \triangleright \psi_2}{\Gamma \vdash e\,x \triangleright \psi_1 \cup \psi_2 \cup \bigcup_{i\in I}\{x \mapsto t \wedge t_i\}}\ (t \wedge t_i \not\simeq \mathbb{0})$$

Whenever a function parameter is the argument of an overloaded function, we record as possible types for this parameter all the domains t_i of the arrows that type the overloaded function, restricted (via intersection) by the static type t of the parameter and provided that the type is not empty ($t \wedge t_i \not\simeq \mathbb{0}$). We show the remarkable power of this rule on some practical examples in Section 4.

3.3. Integrating gradual typing

Gradual typing is an approach proposed by Siek and Taha [37] 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 [41]—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 explore this integration for the formalism of Section 2 for which the integration of gradual typing was first defined by Castagna and Lanvin [7] and successively considerably improved by Castagna et al. [8] (see Lanvin [28] for a comprehensive presentation).

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 JavaScript code of (1) and (3) in the introduction can also be typed by using gradual typing:

```
function foo(x : ?) {
    return (typeof(x) === "number")? x+1 : x.trim();        (32)
}
```

“Standard” or “safe” gradual typing inserts two dynamic checks since it compiles the code above into:

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

where $e\langle t\rangle$ is a type-cast that dynamically checks whether the value returned by e has type t.[12] 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 above in order to statically detect the insertion of useless casts. Using occurrence typing to type the gradually-typed version of `foo` in (32), 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` (but the second cast is still necessary though). 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 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

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

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`$\langle ? \rangle$ 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`$\langle ? \rangle$, and so would the whole function call. Although this behavior is type safe, this violates the gradual guarantee [39] since giving a *more precise* type to the parameter `x` (such as `number`) would make the function succeed, as the cast to $?$ would not be inserted. A solution is to modify the semantics of type-cases, and in particular of `typeof`, to strip off all the casts in values, even nested ones. While this adds a new overhead at runtime, this is preferable to losing the gradual guarantee, and the overhead can be mitigated by having a proper representation of cast values that allows to strip all casts at once.

However, this problem gets much more complex when considering functional values. In fact, as we hinted in Section 2.6, there is no way to modify the semantics of type cases to preserve both the gradual guarantee and the soundness of the system in the presence of arbitrary type cases. For example, consider the function $f = \lambda^{(\mathsf{Int}\to\mathsf{Int})\to\mathsf{Int}} g.(g{\in}(\mathsf{Int} \to \mathsf{Int}))\,?\,g\ 1 : \mathtt{true}$. This function is well-typed since the type of the parameter guarantees that only the first branch can be taken, and thus that only an integer can be returned. However, if we apply this function to $h = (\lambda^{?\to ?}x.\ x)\langle \mathsf{Int} \to \mathsf{Int}\rangle$, the type case strips off the cast around h (to preserve the gradual guarantee), then checks if $\lambda^{?\to ?}x.\ x$ has type $\mathsf{Int} \to \mathsf{Int}$. Since $? \to ?$ is not a subtype of $\mathsf{Int} \to \mathsf{Int}$, the check fails and the application returns `true`, which is unsound. Therefore, to preserve soundness in the presence of gradual types, type cases should not test functional types other than $\mathbb{0} \to \mathbb{1}$, which is the same restriction as the one presented by Siek and Tobin-Hochstadt [38].

While this solves the problem of the gradual guarantee, it is clear that it would be much better if the application `foo(42)` were compiled as is, without introducing the cast `42`$\langle ? \rangle$, thus getting rid of the overhead associated with removing this cast in the type case. 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 (32) the type `(number`$\to$`number)`$\wedge$`((`$?$`\number)`$\to$`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. [8] for the definition of the subtyping relation on these types). For every gradual type τ, define $\tau^{\Uparrow}$ as the (non gradual) 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:

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

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 succeeds it eliminates unnecessary gradual types and, thus, unnecessary casts. Let us see how this works on the function `foo` in (32). First, we deduce the refined hypothesis $\psi(x) = \{$`number`$\wedge ?$, $?$`\number`$\}$. Typing the function using this new hypothesis but without considering the maximal interpretation would yield ($? \to$ `number` $\vee$ `string`) $\wedge$ ((`number` $\wedge\ ?$) $\to$ `number`) $\wedge$ (($?$`\number`) $\to$ `string`). However, as we stated before, this would introduce an unnecessary cast if the function were to be applied to an integer.[13] Hence the need for the second part of Rule [AbsInf+]: the maximal interpretation of `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 (`number` $\to$ `number`) $\wedge$ (($?$`\number`) $\to$ `string`).

4. Implementation

We present in this section preliminary results obtained by our implementation. After giving some technical highlights, we focus on demonstrating the behavior of our typing algorithm on meaningful examples. We also provide an in-depth comparison with the fourteen examples of [43].

[13] Notice that considering `number` $\wedge\ ? \simeq$ `number` is not an option, since it would force us to choose between having the gradual guarantee or having, say, `number` $\wedge$ `string` be more precise than `number` $\wedge\ ?$.

4.1. Implementation details

We have implemented the algorithmic system $\vdash_{\mathcal{A}}$ we presented in Section 2.6.3. Besides the type-checking algorithm defined on the base language, our implementation supports the record types and expressions of Section 3.1 and the refinement of function types described in Section 3.2. Furthermore, our implementation uses for the inference of arrow types the following improved rule:

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

instead of the simpler [AbsInf+] given in Section 3.2. The difference of this new rule with respect to [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_` (see 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.

The implementation is rather crude and consists of 2000 lines of OCaml code, including parsing, type-checking of programs, and pretty printing of types. CDuce is used as a library to provide set-theoretic types and semantic subtyping. The implementation faithfully transcribes in OCaml the algorithmic system $\vdash_{\mathcal{A}}$ as well as all the type operations defined in this work. One optimization that our implementation features (with respect to the formal presentation) is the use of a memoization environment in the code of the $\mathsf{Refine}_{e,t}(\Gamma)$ function, which allows the inference to avoid unnecessary traversals of e. Lastly, while our prototype allows the user to specify a particular value for the n_o parameter we introduced in Section 2.6.2, a value of 1 for n_o is sufficient to check all examples we present in the rest of the section.

4.2. Experiments

We demonstrate the output of our type-checking implementation in Table 1 and Table 2. Table 1 lists some examples, none of which can be typed by current systems. Even though some systems such as Flow and TypeScript can type some of these examples by adding explicit type annotations, the code 6, 7, 9, and 10 in Table 1 and, even more, the `and_` and `xor_` functions given in (33) and (34) later in this section are out of reach of current systems, even when using the right explicit annotations.

It should be noted that for all the examples we present, the time for the type inference process is less than 5 ms, hence we do not report precise timings in the table. These and other examples can be tested in the online toplevel available at https://occtyping.github.io/

In Table 1, the second column gives a code fragment and the third column the type deduced by our implementation as is (we pretty printed it but we did not alter the output). Code 1 is a straightforward function similar to our introductory example `foo` in (1) and (3) where `incr` is the successor function and `lneg` the logical negation for Booleans. Here the programmer annotates the parameter of the function with a coarse type $\mathsf{Int}\vee\mathsf{Bool}$. Our implementation first type-checks the body of the function under this assumption, but doing so it 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 $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$. Note that w.r.t. rule [AbsInf+] of Section 3.2, the rule [AbsInf++] we use in the implementation improves the output of the computed type. Indeed, using rule [AbsInf+] we would have obtained the type $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})\wedge(\mathsf{Bool}\vee\mathsf{Int}\to\mathsf{Bool}\vee\mathsf{Int})$ with a redundant arrow. Here we can see that, since we deduced the first two arrows $(\mathsf{Int}\to\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool})$, and since the union of their domain exactly covers the domain of the third arrow, then 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 $\mathbb{1}$, 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 [43]. 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 $\neg\mathsf{True}$ is equivalent to `Any`$\setminus$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 the CDuce's types that our system uses, is syntactic sugar for $\mathsf{True}\vee\mathsf{False}$) yielding the type $(\mathsf{True}\to\mathsf{False})\wedge(\mathsf{False}\to\mathsf{True})$. 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: the function returns `false` if and only if both arguments are of type $\neg\mathsf{True}$, that is, they are any value

Table 1
Types inferred by the implementation.

	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 → Any → True) ∧ (¬True → 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 → Any → 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 omitted)
		1
		2
		3

8	atom nil type Document = { nodeType=9 ..} and Element = { nodeType=1, childNodes=NodeList ..} and Text = { nodeType=3, isElementContentWhiteSpace=Bool ..} 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 is { nodeType=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
11	let typeof = fun (x:Any) -> if x is Int then "number" else if x is Char then "string" else if x is Bool then "boolean" else "object" let test = fun (x:Any) -> if typeof x is "number" then incr x else if typeof x is "string" then charcode x else if typeof x is "boolean" then int_of_bool x else 0	(Int → "number")∧ (Char → "string")∧ (Bool → "boolean")∧ (¬(Bool∨Int∨Char) → "object") ∧ . . . (two other redundant cases omitted) (Int → Int) ∧ (Char → Int) ∧ (Bool → Int)∧ (¬(Bool∨Int∨Char) → 0) ∧ . . . (two other redundant cases omitted)
12	atom null type Object = Null | { prototype = Object ..} type ObjectWithPropertyL = { l = Any ..} | { prototype = ObjectWithPropertyL ..} let has_property_l = fun (o:Object) -> if o is ObjectWithPropertyL then true else false let has_own_property_l = fun (o:Object) -> if o is { l=Any ..} then true else false let get_property_l = fun (self:Object->Any) o -> if has_own_property_l o is True then o.l else if o is Null then null else self (o.prototype)	(ObjectWithPropertyL → True) ∧ (X1 → False) where X1 = (Nil | {l = ?Empty, prototype = X1 ..}) ({l = Any, prototype = Object ..} → True) ∧ ((Nil | {l = ?Empty, prototype = Object ..}) → False) Object → Any

different from true. 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 $(\mathsf{True} \to \neg\mathsf{True} \to \mathsf{False}) \wedge (\mathsf{True} \to \mathsf{True} \to \mathsf{True})$).

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 statically 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., 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 can directly be accessed after having checked for the appropriate nodeType, whereas in 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 record expressions 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.

Code 9 shows the usefulness of the rule [OverApp]. Consider the definition of the xor_ operator. 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. However, since or_ has type $(\mathsf{True} \to \mathbb{1} \to \mathsf{True}) \wedge (\mathbb{1} \to \mathsf{True} \to \mathsf{True}) \wedge (\neg\mathsf{True} \to \neg\mathsf{True} \to \mathsf{False})$ then the rule [OverApp] applies and True, $\mathbb{1}$, and ¬True become candidate types for x, which allows us to deduce the precise type given in the table. Finally, thanks to rule [OverApp] it is not 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))      (33)
let xor_ = fun (x : Any) -> fun (y : Any) -> and_ (or_ x y) (not_ (and_ x y))  (34)
```

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

As for Code 10 (corresponding to our introductory example (11)), it illustrates the need for iterative refinement of type environments, as defined in Section 2.6.2. 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.

Code 11 implements the typical type-switching pattern used in JavaScript. While languages such as Scheme and Racket hard-code specific type predicates for each type—predicates that our system does not need to hard-code since they can be directly defined (cf. Code 3)—, JavaScript hard-codes a typeof function that takes an expression and returns a string indicating the type of the expression. Code 11 shows that typeof can be encoded and precisely typed in our system. Indeed, constant strings are simply encoded as fixed list of characters (themselves encoded as pairs as usual, with special atom nil representing the empty list). Thanks to our precise tracking of singleton types both in the result type of typeof and in the type case of test, we can deduce for the latter a precise type (the given in Table 1 is equivalent to $(\mathsf{Any} \to \mathsf{Int}) \wedge (\neg(\mathsf{Bool}{\vee}\mathsf{Int}{\vee}\mathsf{Char}) \to 0)$).

Code 12 simulates the behavior of JavaScript property resolution, by looking for a property l either in the object o itself or in the chained list of its prototype objects. In this example, we first model prototype-chaining by defining a type Object that can be either the atom Null or any record with a prototype field which contains (recursively) an Object. To ease the reading, we defined a recursive type ObjectWithPropertyL which is either a record with a field l or a record with a prototype of type ObjectWithPropertyL. We can then define two predicate functions has_property_l and has_own_property_l that test whether an object has a property through its prototype or directly. Lastly, we can define a function get_property_l which directly accesses the field if it is present, or recursively search for it through the prototype chain; the recursive search is implemented by calling the explicitly-typed parameter `self` which, in our syntax, refers to the function itself. Of particular interest is the type deduced for the two predicate functions. Indeed, we can see that `has_own_property_l` is given an overloaded type whose first argument is in each case a recursive record type that describes precisely whether `l` is present at some point in the list or not (recall that in a record type a field such as $\{\ell = ?\mathsf{Empty}\ ..\}$, indicate that field ℓ is surely absent). Notice that in our language a fixpoint combinator can be defined as follows

```
type X = X -> S -> T
let z = fun (((S -> T) -> S -> T ) -> (S -> T)) f ->
      let delta = fun ( X -> (S -> T) ) x ->
         f ( fun (S -> T) v -> ( x x v ))
       in delta delta
```

which applied to any function $\mathtt{f}:(S\to T)\to S\to T$ returns a function $(\mathtt{z}\ \mathtt{f}):S\to T$ such that for every non diverging expression e of type S, the expression $(\mathtt{z}\ \mathtt{f})e$ (which is of type T) reduces to $\mathtt{f}((\mathtt{z}\ \mathtt{f})e)$. It is then clear that definition of `get_property_l` in Code 12, is nothing but syntactic sugar for

```
let get_property_l =
  let aux = fun (self:Object->Any) -> fun (o:Object)->
    if has_own_property_l o is True then o.l
    else if o is Null then null
    else self (o.prototype)
  in z aux
```

where S is `Object` and T is `Any`.

4.3. Comparison

In Table 2, we reproduce in our syntax the 14 archetypal examples of Tobin-Hochstadt and Felleisen [43] (we tried to complete such examples with neutral code when they were incomplete in the original paper). Of these 14 examples, Example 1 to 13 depict combinations of type predicates (such as `is_int`) used either directly or through Boolean predicates (such as the `or_` function previously defined). Note that for all examples for which there was no explicit indication in the original version, we *infer* the type of the function whereas in [43] the same examples are always in a context where the type of identifiers is known or the input type of function is fully annotated. Notice also that for Example 6, the goal of the example is to show that indeed, the function is ill-typed (which our typechecker detects accurately).

The original Example 14 of Tobin-Hochstadt and Felleisen [43] is the only case of their work that our system cannot directly capture. It can be written in our syntax as:

```
  let example14 = fun (input : Int|String) ->
    fun (extra : (Any, Any)) ->
      if and2_(is_int input , is_int(fst extra)) is True then
         add input (fst extra)
      else if is_int(fst extra) is True then
         add (strlen input) (fst extra)
      else 0
```
(35)

where `and2_` is the uncurried version of the `and_` function we defined in (33) and `is_int` is the function defined in the third row of Table 1. Our system rejects the expression above, while the system by Tobin-Hochstadt and Felleisen [43] correctly infers the function always return an integer. The reason why our system rejects it is because the type it deduces for the occurrence of `input` in the 6th line of the code is `Int|String` rather than `String` as required by the application of `strlen`. The general reason for this failure is that, contrary to [43], our system does not implement an analysis of the flow of type information. In particular, since the variable `input` does not occur in the condition of the second `if`, then its type is not refined (as it could be). Indeed, if the first test fails, it is either because `fst extra` is not an integer (i.e., `is_int(fst extra)` is not True) or because `input` is not an integer. Therefore, in our setting, the type information propagated to the second test for the pair of the arguments in the first test is : `(is_int input , is_int(fst extra))` $\in \neg(\mathsf{True},\mathsf{True})$, that is `(input, is_int(fst extra))` $\in (\neg\mathsf{Int},\mathsf{True}) \vee (\mathsf{Int},\mathsf{False})$. Since the second test checks whether `is_int(fst extra)` holds or not, then we could deduce that the following occurrence of `input` is of type $\neg\mathsf{Int}$. But since `input` does not occur in the test, this refinement of the type of `input` is not done. Instead, the type deduced for `input` in the second branch is $(\mathsf{String} \vee \mathsf{Int}) \wedge (\neg\mathsf{Int} \vee \mathsf{Int}) = \mathsf{String} \vee \mathsf{Int}$ which is not precise enough to type the application `strlen input`. It not difficult to patch, alas unsatisfactorly, this example in our system: it suffices to test dummily in the second `if` the whole argument of `and2_`, without really checking its first component:

Table 2
Comparison with the 14 examples of Tobin-Hochstadt and Felleisen [43].

	Code	Inferred type

1	`(* Assumes: add1 : Int -> Int *)`
	`let example1 = fun (x:Any) ->`
	`     if x is Int then add1 x else 0`
		Int → Int

2	`(* Assumes strlen: String -> Int *)`
	` let example2 = fun (x:String|Int) ->`
	`  if x is Int then add1 x else strlen x`
		(Int → Int) ∧ (String → Int)

3	`let example3 = fun (x: Any) ->`
	`    if x is (Any \ False) then (x,x) else false`
		(False → False) ∧ (¬False¬(¬False, ¬False))

4	`(*Uses 'is_int' from Table 1.3 and 'or_' from Table 1.5,`
	`  assumes f : (Int|String) -> Int *)`
	`let is_string = fun (x : Any) ->`
	`  if x is String then true else false`
	
	`let example4 = fun (x : Any) ->`
	`   if or (is_int x) (is_string x) is True then x else 'A'`
		(String → True) ∧ (¬String → False)
		
		(Int → Int) ∧ (String → String)∧
		(¬Int → (String ∨ 'A'))∧
		(¬String → (Int ∨ 'A'))∧
		(¬(String ∨ Int) → 'A')

5	`(*Uses 'and_' from Table 1.6,`
	`  assumes strlen : String -> Int *)`
	`let example5 = fun (x : Any) -> fun (y : Any) ->`
	`   if and_ (is_int x) (is_string y) is True then`
	`    add x (strlen y) else 0`
		(Int → String → Int) ∧ (Int → ¬String → 0)) ∧
		(¬Int → String → 0) ∧ (¬String → 0)

6	`let example6 = fun (x : Int|String) -> fun (y : Any) ->`
	`if and_ (is_int x) (is_string y) is True then`
	`  add x (strlen y) else strlen x`
		Type error for `strlen x`, `x` has type Int ∨ String.

7	`let example7 = fun (x : Any) -> fun (y : Any) ->`
	`if`
	` (if (is_int x) is True then (is_string y) else false)`
	`  is True then`
	`  add x (strlen y) else 0`
		(Int → String → Int) ∧ (Int → ¬String → 0)) ∧
		(¬Int → String → 0) ∧ (¬String → 0) (identical to example 5)

8	`let example8 = fun (x : Any) ->`
	` if or_ (is_int x) (is_string x) is True then true`
	` else false`
		(Int → True) ∧ (String → True) ∧
		(¬(String ∨ Int) → False)

9	`let example9 = fun (x : Any) ->`
	`if`
	`  (if is_int x is True then is_int x else is_string x)`
	`  is True then  f x else 0`
		(Int → Int) ∧ (String → Int) ∧
		(¬(String ∨ Int) → 0)

10	`let example10 = fun (p : (Any,Any)) ->`
	`  if is_int (fst p) is True then add1 (fst p) else 7`
		((Int, Any) → Int) ∧ ((¬(Int, Any) → 7)

11	`let example11 = fun (p : (Any, Any)) ->`
	`  if and_ (is_int (fst p)) (is_int (snd p)) is True`
	`  then g p else no`
		((Int, Int) → Int) ∧ (((Any, ¬Int) ∨ (¬Int, Any)) → no)

12	`let example12 =`
	`    fun (p : (Any, Any)) ->`
	`    if is_int (fst p) is True then true else false`
		((Int, Any)− > True) ∧ ((¬Int, Any)− > False))

13	`let example13 = fun (x : Any) -> fun (y : Any) ->`
	`  if and_ (is_int x) (is_string y) is True then 1`
	`  else if is_int x is True then 2`
	`  else 3`
		(Int → String → 1) ∧ (Int → ¬String → 2))∧
		(¬Int → Any → 3)

14	`let example14_alt = fun (input : Int | String) ->`
	` fun (extra : (Any, Any)) ->`
	`  if and2_((is_int input),(is_int (fst extra))) is True`
	`    then add input (fst extra)`
	`  else if (is_int input,is_int (fst extra)) is (Any,True)`
	`    then add (strlen input) (fst extra)`
	`  else 0`
		(Int → ((Int, Any) → Int) ∧ ((¬Int, Any) → 0))∧
		(String → ((Int, Any) → Int) ∧ ((¬Int, Any) → 0))

```
let example14_alt = fun (input : Int|String) ->
  fun (extra : (Any, Any)) ->
    if and2_(is_int input , is_int(fst extra)) is True then
       add input (fst extra)
    else if (is_int input , is_int(fst extra)) is (Any,True) then
       add (strlen input) (fst extra)
    else 0
```

Even if the type of `is_int input` is not really tested (any result will produce the same effect) its presence in the test triggers the refinement of the type of the last occurrence of `input`, which type checks with the (quite precise) type shown in the entry 14 of Table 2, type that is equivalent to Int ∨ String → ((Int, Any) → Int) ∧ ((¬Int, Any) → 0). Lifting this limitation through a control-flow analysis is part of our future work.

In our system, however, it is possible to express dependencies between different arguments of a function by uncurrying the function and typing its arguments by a union of products. To understand this point, consider this simple example:

```
let sum = fun (x : Int|String) -> fun (y : Int|String) ->
    if x is String then concat x y else add x y
```

The definition above does not type-check in any available system, and rightly does so since nothing ensures that `x` and `y` will be either both strings (so that `concat` does not fail) or both integers (so that `add` does not fail). It is however possible to state this dependency between the type of the two arguments by uncurring the function and using a union type:

```
let sum = fun (x : (Int,Int)|(String,String))
    if fst x is String then concat(fst x)(snd x) else add(fst x)(snd x)
```

this function type-checks in our system (and, of course, in Typed Racket as well) but the corresponding type-annotated version in JavaScript

```
function sum (x : [string,string]|[number,number]) {
  if (typeof x[0] === "string") {
     return x[0].concat(x[1]);
  } else {
     return x[0] + x[1];
  }
}
```

is rejected both by Flow and TypeScript since their type analyses fail to detect the dependency of the types of the two projections.

Although these experiments are still preliminary, they show how the combination of occurrence typing and set-theoretic types, together with the type inference for overloaded function types presented in Section 3.2 goes beyond what languages like TypeScript and Flow do, since they can only infer single arrow types. Our refining of overloaded functions is also future-proof and resilient to extensions: since it “retypes” functions using information gathered by the typing of occurrences in the body, its precision will improve with any improvement of our occurrence typing framework.

5. Related work

Occurrence typing was introduced by Tobin-Hochstadt and Felleisen [42] and further advanced in [43] 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 [43] define λ_{TR}, a core calculus for Typed Racket. In this language types are annotated by two logical propositions that record the type of the input depending on the (Boolean) value of the output. For instance, the type of the `number?` function states that when the output is `true`, then the argument has type `Number`, and when the output is `false`, the argument does not. Such information is used selectively in the “then” and “else” branches of a test. Since Tobin-Hochstadt and Felleisen [43] focus their analysis on a particular set of pure operations, the approach works also in the presence of side-effects. Although the choices made by our and their approach seem poles apart (Boolean output of few pure operations vs. any output of every expression), they share some similar techniques. For instance, our deduction system for $\vdash^{\texttt{Path}}$ plays a similar role as the proof systems and update function of Tobin-Hochstadt and Felleisen [43, Figures 4, 7 & 9]. In that framework, in order to type a variable (judgement “$\Gamma \vdash x : \tau$”) one needs to prove that the logical formula τ_x holds (under the hypotheses of Γ). This atomic formula may not be directly available in Γ but may be proven by a combination of logical deduction rules (Fig. 4 of [43]), or by recursively exploring a path leading to x (Fig. 7 and 9 of [43]) a path being a sequence of **cdr** or **car** applications, much like our f and s components of paths. This idea is also present in our deduction system for $\vdash^{\texttt{Path}}$ with differences pertaining to our type framework and design choices: type restrictions can be encoded using set-theoretic intersections and negations (instead of meta-functions working on the syntax of types) and our richer language of paths components. 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. This is particularly beneficial when typing functions since the logical propositions of Tobin-Hochstadt and Felleisen can record dependencies on expressions other than the input of a function. Consider for instance the following example (due to [24]) in JavaScript `function is-y-a-number(x) { return(typeof(y) === "number") }` which defines a functions that disregards its argument and returns whether the variable `y` is an integer or not.[14] While our approach cannot deduce for this function but the type $\mathbb{1} \to$ Bool, the logical approach of Tobin-Hochstadt and Felleisen can record in

[14] Although such a function may appear nonsensical, Kent [24] argues that it corresponds a programming pattern that may appear in Typed Racked due to the expansion of some sophisticated macro definitions.

the type of `is-y-a-number` the fact that when the function returns `true`, then `y` is a number, and the opposite when it returns `false`. In our approach, the only possibility to track such a dependency is that the variable `y` is the parameter of an outer function to which our analysis could give an overloaded type by splitting the type `Any` of `y` into `Number` and $\neg$`Number`. Under the hypothesis of `y` being of type `Number` the type inferred for `is-y-a-number` will then be $\mathbb{1} \to \mathsf{True}$, and $\mathbb{1} \to \mathsf{False}$ otherwise, thus capturing the wanted dependency. Although the approach of using logical proposition has the undeniable advantage over ours of providing more a flow sensitive analysis, we believe that using semantic subtyping as a foundation as we do has also several merits over the logical proposition approach. 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 formulæ. 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 the key examples of Tobin-Hochstadt and Felleisen [43] (the notable exceptions being Example 9 and 14 in their paper, which use the propagation of type information outside of the branches of a test). While Typed Racket supports structured data types such as pairs and records only unions of such types can be expressed at the level of types, and even for those, subtyping is handled axiomatically. For instance, for pairs, the subtyping rule presented in [43] is unable to deduce that $(\texttt{number} \times (\texttt{number} \cup \texttt{bool})) \cup (\texttt{bool} \times (\texttt{number} \cup \texttt{bool}))$ is a subtype of (and actually equal to) $((\texttt{number} \cup \texttt{bool}) \times \texttt{number}) \cup ((\texttt{number} \cup \texttt{bool}) \times \texttt{bool})$ (and likewise for other type constructors combined with union types). For record types, we also type precisely the deletion of labels, which, as far as we know no other system can do. On the other hand, the propagation of logical properties defined in [43] is a powerful tool, that can be extended to cope with sophisticated language features such as the multi-method dispatch of the Closure language [5].

For what concerns the first work by Tobin-Hochstadt and Felleisen [42] it is interesting to compare it with our work because the comparison shows two rather different approaches to deal with the property of type preservation. Tobin-Hochstadt and Felleisen [42] define a first type system that does not satisfy type-preservation. The reason for that is that this first type system checks all the branches of a type-case expression, independently from whether they are selectable or not; this may result in a well-typed expression to reduce to an expression that is not well-typed because it contains a type-case expression with a branch that, due to the reduction, became both non-selectable and ill-typed (see [42, Section 3.3]). To obviate this problem they introduce a *second* type system that extends the previous one with some auxiliary typing rules that type type-case expressions by skipping the typing of non-selectable branches. They use this second type system only to prove type preservation and obtain, thus, the soundness of their type system. In our work, instead, we prefer to start directly with a system that satisfies type preservation. Our system does not have the problem of the first system of [42] thanks to the presence of the [Efq] rule, that we included for that very purpose, that is, to skip non-selectable branches during typing. The choice of one or the other approach is mostly a matter of taste and, in this specific case, boils down to deciding whether some typing problems must be signaled at compile time by an error or a warning. The approach of Tobin-Hochstadt and Felleisen [42] ensures that every subexpression of a program is well-typed and, if not, it generates a type-error. Our approach allows some subexpressions of a program to be ill-typed, but only if they occur in dead branches of type-cases: in that case any reasonable implementation would flag a warning to signal the presence of the dead branches. The very same reasons that explain the presence in our system of [Efq], explain why from the beginning we included in our system the typing rule [Abs-] that deduces negated arrow types: we wanted a system that satisfied type preservation (albeit, for a parallel reduction: cf. Appendix A.2). We then defined an algorithmic system that is not *complete* with respect to the type-system but from which it inherits its soundness. Of course, we could have proceeded as Tobin-Hochstadt and Felleisen [42] did: start directly with a type-system corresponding to the algorithm (i.e., omit the rule [Abs-]) and later extend this system with the rule to infer negated arrows, the only purpose of this extension being to prove type preservation. We preferred not to, not only because we favor type preserving systems, but also because in this way we were able to characterize different subsystems that are complete with respect to the algorithmic system, thus exploring different language designs and arguing about their usefulness.

Highly related to our work is Andrew M. Kent's PhD. dissertation [24], in particular its Chapter 5 whose title is “A set-theoretic foundation for occurrence typing” where he endows the logical techniques of [43] with the set-theoretic types of semantic subtyping [19]. Kent's work builds on the approach developed for Typed Racket that, as recalled above, consists in enriching the types of the expressions with information to track under which hypotheses an expression returns false or not (it considers every non false value to be “truthy”). This tracking is performed by recording in the type of the expression two logical propositions that hold when the expression evaluates to false or not, respectively. The work in Kent [24, Chapter 5] uses set-theoretic types to express type predicates (a predicate that holds only for a type t has type $p : (t \to \texttt{True}) \wedge (\neg t \to \texttt{False})$) as well as to express in a more compact (and, sometimes, more precise) way the types of several built-in Typed Racket functions. It also uses the properties of set-theoretic types to deduce the logical types (i.e., the propositions that hold when an expressions produces `false` or not) of arguments of function applications. To do that it defines a type operator called *function application inversion*, that determines the largest subset of the domain of a function for which an application yields a result of a given type t, and then uses it for the special cases when the type t is either `False` or $\neg$`False` so as to determine the logical type of the argument. For instance, this operator can be used to deduce that if the application `boolean? x` yields false, then the logical proposition $\texttt{x}{\in}\neg\texttt{Bool}$ holds true. The definition of our *worra* operator that we gave in equation (16) is, in its spirit, the same as Kent's *function application inversion* operator (more precisely, the same as the operator pred Kent defines in Figure 5.7 of his dissertation), even though the two operators were defined independently from each other. The exact definitions however are slightly different, since the algorithm given in Kent [24, Figure 5.2] for *function application inversion* is sound only for functions whose type is an intersection of arrows, whereas our definition of worra, given in (18), is sound and complete for any function, in particular, for functions that have a union type (for which Kent's definition may yield unsound results). Apart from these technical issues, the main difference of Kent's approach with respect to ours is that, since it builds on the logical propositions approach, then it focus the use of set-theoretic types and of the worra (or application inversion) operator to determine when an expression yields a result of type `False` or ¬`False`. We have instead a more holistic approach since, not only our analysis strives to infer type information by analyzing all types of results (and not just `False` or ¬`False`), but also it tries to perform this analysis for all possible expressions (and not just for a restricted set of expressions). For instance, we use the operator worra also to refine the type of the function in an application (see discussion in Section 1.2) while in Kent's approach the analysis of an application `f x` refines the properties of the argument `x` but not of the function `f`; and when such an application is the argument of a type test, such as in `number? (f x)`, then in Kent's approach it is no longer possible to refine the information on the argument `x`. The latter is not is a flaw of the approach but a design choice: as we explain at the end of this section, the approach of Type Racket not only focuses on the inference of two logical propositions according to the truthy or false value of an expression, but also it does it only for a selected set of *pure* expressions of the language, to cope with the possible presence of side effects (and applications do not belong to this set since they can be impure). That said, the very fact of focusing on truthy vs. false results may make Kent's analysis fail even for pure Boolean tests where it would be naively expected to work. For example, consider the polymorphic function that when applied to two integers returns whether they have the same parity and false otherwise: `have_same_parity`: $(\mathsf{Int}\to\mathsf{Int}\to\mathsf{Bool}) \wedge (\neg\mathsf{Int}\to\mathsf{Any}\to\mathsf{False}) \wedge (\mathsf{Any}\to\neg\mathsf{Int}\to\mathsf{False})$. We can imagine to use this function to implicitly test whether two arguments are both integers, as in the body of the following function:

```
let f = fun (x : Any) -> fun (y : Any) ->
  if have_same_parity x y is True then add x y else 0
```

While our approach can correctly deduce for this function the type $\mathsf{Any}\to\mathsf{Any}\to\mathsf{Int}$, Kent's approach fails to type check it since to type the "then" branch requires to deduce that the application `have_same_parity x` returns the constant function `true` only if `x` is an integer. Finally, Kent's approach inherits all the advantages and disadvantages that the logical propositions approach has with respect to ours (e.g., flow sensitive analysis vs. user-defined type predicates) that we already discussed at the beginning of this section.

Another direction of research related to ours is the one on semantic types. In particular, several attempts have been made recently to map types to first order formulæ. In that setting, subtyping between types translates to logical implication between formulæ. Bierman et al. [4] introduce Dminor, a data-oriented language featuring a `SELECT`-like construct over collections. Types are mapped to first order formulæ and an SMT-solver is then used to (try to) prove their satisfiability. The refinement types they present go well beyond what can be expressed with the set-theoretic types we use (as they allow almost any pure expression to occur in types). However, the system forgoes any notion (or just characterization) of completeness and the subtyping algorithm is largely dependent on the subtle behavior of the SMT solver (which may timeout or give an incorrect model that cannot be used as a counter-example to explain the type-error). As with our work, the typing rule for the `if` e `then` e_1 `else` e_2 construct of Dminor refines the type of each branch by remembering that e (resp. $\neg e$) is true in e_1 (resp. e_2) and this information is not propagated to the outer context. A similar approach is taken by Chugh et al. [16], and extended to so-called nested refinement types. In these types, an arrow type may appear in a logical formula (whereas previous work only allowed formulæ on "base types"). This is done in the context of a dynamic language and their approach is extended with polymorphism, dynamic dispatch and record types. A problem that is faced by refinement type systems is the one of propagating in the branches of a test the very precise information learned from the test (usually that some equality between terms holds). A solution that is for instance chosen by Ou et al. [32] and Knowles and Flanagan [26] is to devise a meta-function that recursively explores both a type and an expression and constructs a more precise *dependent* type. In the dependent type, fresh variables are introduced to name sub-expressions and record the new constraints. This process—called in the cited works *selfification*—roughly corresponds to our Constr and Refine functions (see Section 2.6.2). Another approach is the one followed by Rondon et al. [35] which is completely based on a program transformation, namely, it consists in putting the term in *A-normal form* as defined by Sabry and Felleisen [36]. Using a program transformation, every destructor application (function application, projection, ...) is given a name through a let-binding. The problem of tracking precise type information for every sub-expression is therefore reduced to the one of keeping precise typing information for a variable. While this solution seems appealing, it is not completely straightforward in our case. Indeed, to retain the same degree of precision, one would need to identify α-equivalent sub-expressions so that they share the same binding, something that a plain A-normalization does not provide (and which, actually, must not provide, since in that case the transformation may not preserve the reduction semantics).

Among the work on refinement types, some have studied the extensions of a refinement type-system with intersection types. For instance, [1] studies a type system with refinement types, polymorphism and full union and intersection (but no negation). While the goal of their type-system is to verify secure protocol implementations, the core language RCF$^{\forall}_{\wedge\vee}$ they present, as well as the associated type-system is a λ-calculus with pattern-matching, let bindings, and a refining test for equality (as well as protocol-oriented constructs such as channel creation, message passing, and expression forking). While on the surface their types resemble ours, they follow another direction. First, their language is fully annotated (meaning that, for instance, polymorphic terms must be explicitly instantiated and intersection types must also be specified through an annotation). Second, since the subtyping relation they provide is syntactic, it cannot in general take into account the distributivity of logical connectives with respect to type constructors. This limitation is however not a problem since the main goal of their subtyping relation is to propagate a *kinding* information that they use to characterize the level of knowledge an attacker may have about a particular value. Another work adding intersection types to refinement types is [33] in the context of liquid types. This work introduces intersection (but not union nor negations) to liquid types, with a particular focus on intersection of arrow types. This work uses a syntactic subtyping relation to push down intersection of types into the logical formulas of types. Once the formulas have been propagated, they are offloaded to an SMT solver to decide the base case of the subtyping relation. Of particular interest is their type-inference algorithm. Contrary to ours, their inference is based on algorithm $\mathcal{W}$, using the polymorphic type deduced as a template for an intersection. They can therefore infer intersection arrow types that are several distinct instances of the same polymorphic type.

Kent et al. [25] bridge the gap between prior work on occurrence typing and SMT-based (sub-)typing. They introduce the λ_{RTR} core calculus, an extension of λ_{TR} of [43] where the logical formulæ 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). The cost of this expressive power in types is however paid by the programmer, who has to write logical annotations (to help the external provers). Here, types and formulæ remain segregated. Subtyping of “structural” types is checked by syntactic rules (as in [43]) while logical formulæ present in type predicates are verified by the SMT solver.

Chaudhuri et al. [15] 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 we showed in (3)) 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 look 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 seems 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. More generally, we believe that what sets our work apart in the palimpsest of the research on occurrence typing is that we have a type-theoretic foundational approach striving as much as possible to explain occurrence typing by extending prior (unrelated but standard) work while keeping prior results. In that respect, we think that our approach is not satisfactory, yet, because it uses non standard type-environments that map expressions rather than variables to types: but all the rest is standard type-theory. And even on the latter aspect it must be recognized that the necessity of tracking types not only for variables but also for more structured expressions is something that shows up, in different forms, in several other approaches. For instance, in the approach defined for Typed Racket [43] the type-system associates to an expression a quadruple formed by its type, two logical propositions, and an object which is a pointer to the environment for the type hypothesis about the expression and, as such, it plays the role of our extended type environments. Likewise, the *selfification* of [32] and [26], propagates the precise type constraints learned during a test. One difference with our approach is that with refinement types the information can be kept at the level of types, since dependent types contain terms and can introduce variables, while in our approach the mapping is kept separate in a type environment. In summary the tracking of types for structured expressions seems an aspect common to different approaches to occurrence types, nevertheless we are confident that even this last non-standard aspect of our system can be removed and that occurrence typing can be explained in a pure standard type-theoretic setting.

On the practical side, while languages such as Flow and Typed Racket are the golden standard of occurrence typing, it may be worth citing that there exist other programming languages that implement some much more simplistic forms of occurrence typing. Languages such as Kotlin [23] and Dart [20] enforce null safety by performing occurrence typing whenever the tested expression is a variable. CDuce [14] implements a slightly more sophisticated form of this simplistic occurrence typing since it is able to refine in the branches of a test the type of all variables that occur in the tested expression as long as they are subexpressions of non-functional values: so for instance for an expression of the form $((x,(f\,z,y)){\in}(\mathsf{Int}{\times}(\mathsf{Int}{\times}\mathsf{Int})))\,?\,e_1 : e_2$ CDuce is able to specialize in e_1 the types of x and y (to Int) but not those of f or z (since they occur in an application). Likewise, Kotlin also supports dynamically testing the type of an object (using the `is` operator similar to Java’s `instanceOf`) and refining the type of the tested variable in the corresponding branch of a test, without having to resort to a manual down-cast. As expected, Kotlin can only refine the type of variables it can statically determine to be immutable, namely local variables introduced by an immutable `val` binding and mutable references introduced by a `var` binding, provided the reference is not modified between the type test and its occurrences in the branch.

This work already has a follow-up, which was recently presented at the POPL conference [9]. Both this work and the system in [9] use the characteristics of semantic subtyping to improve occurrence typing. Both works obtain this improvement by precisely tracking the type of each occurrence of an expression. However, they use rather different techniques to track the occurrences of an expression and associate them with types. In this work, we do it by enriching type environments so that they map occurrences of expressions (expressed in terms of paths) to types. In [9], instead, the different occurrences of the same expression are tracked by using explicit bindings. In practice, in [9] every expression is transformed into an intermediate representation—dubbed maximal-sharing canonical form (MSC-form)—that consists of a list of bindings from variables to expressions whose proper subexpressions are all variables. This form is called *maximal sharing* because all occurrences of a given expression are mapped by the same binding. In other terms, for each subexpression, there is a unique variable and a unique binding that tracks it. The advantages of using bindings instead of enhanced type environments and paths are twofold. First, the definition of the type system is standard: type environments map variables to types, and occurrence typing is expressed by combining the typing rules for type-case expressions with the standard union-elimination rule by MacQueen et al. [30]. Second, MSC-forms relate via a binding all occurrences of a given expression; so, in particular, they may relate occurrences that are inside a type-case with occurrences that are outside it. This allows the system of [9] to capture and analyze the flows of information between different expressions, a kind of analysis that makes the strength of the approaches heralded by Flow and Typed Racket and which constitutes one of the main limitations of the approach presented here.

We end this presentation of related work with a discussion on side effects. Although in our system we did not take into account side-effects—and actually our system works because all the expressions of our language are pure—it is interesting to see how the different approaches of occurrence typing position themselves with respect to the problem of handling side effects, since this helps to better place our work in the taxonomy of the current literature. As Sam Tobin-Hochstadt insightfully noticed, one can distinguish the approaches that use types to reason about the dynamic behavior of programs according to the set of expressions that are taken into account by the analysis. In the case of occurrence typing, this set is often determined by the way impure expressions are handled. On the one end of the spectrum lies our approach: our analysis takes into account *all* expressions but, in its current formulation, it works only for pure languages. On the other end of the spectrum we find the approach of Typed Racket whose analysis reasons about a limited and predetermined set of *pure* operations: all data structure accessors. Somewhere in-between lies the approach of the Flow language which, as hinted above, implements a complex effect systems to determine pure expressions. While the system presented here does not work for impure languages, we argue that its foundational nature predisposes it to be adapted to handle impure expressions as well, by adopting existing solutions or proposing new ones. For instance, it is not hard to modify our system so that it takes into account only a set of predetermined pure expressions, as done by Typed Racket: it suffices to modify the definition of $\Gamma \vdash^{\texttt{Env}}_{e,(\neg)t} \Gamma'$ (cf. Section 2.5) so that Γ' extends Γ with type hypotheses for all expressions occurring in e that are also in the set of predetermined pure expressions (instead of extending it for all subexpressions of e, *tout court*). However, such a solution would be marginally interesting since by excluding from the analysis all applications we would lose most of the advantages of our approach with respect to the one with logical propositions. Thus a more interesting solution would be to use some external static analysis tools—e.g., to graft the effect system of Chaudhuri et al. [15] on ours—to detect impure expressions. The idea would be to mark different occurrences of a same impure expression using different marks. These marks would essentially be used to verify the presence of type hypotheses for a given expression in a type environment Γ; the idea being that expressions with different marks are to be considered as different expressions and, therefore, would not share the same type hypothesis. For instance, consider the test $(f\,x \in \mathsf{Int})\,?\,\ldots : \ldots$: if $f\,x$ were flagged as impure, then an occurrence of $f\,x$ in the "then" branch would not be supposed to be of type Int since it would be typed in an environment Γ containing a binding for an $f\,x$ expression having a mark different from the one in the "then" branch: the regular typing rules would apply for $f\,x$ in that case. This would certainly improve our analysis, but we believe that ultimately our system should not resort to external static analysis tools to detect impure expressions but, rather, it has to integrate this analysis with the typing one, so as to mark *only* those impure expressions whose side-effects may affect the semantics of some type-cases. For instance, consider a JavaScript object `obj` that we modify as follows: `obj["key"] = 3`. If the field `"key"` is already present in `obj` with type Int and we do not test it more than about this type, then it is not necessary to mark different occurrences of `obj` with different marks, since the result of the type-case will not be changed by the assignment; the same holds true if the field is absent but type-cases do not discriminate on its presence. Otherwise, some occurrences of `obj` must use different marks: the analysis will determine which ones. We leave this study for future work.

6. Future work and conclusion

In this work we presented the core of our analysis of occurrence typing, extended it to record types and proposed a couple of novel applications of the theory, namely the reconstruction of intersection types for unannotated 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 [43] as plain functions and have the inference procedure deduce automatically the correct overloaded function type. More generally, our approach surpasses current ones in that it can deduce precise (overloaded) types for functions that in all other approaches either require the programmer to specify the full precise type (e.g., the function `foo` we defined in (1) and (3) in our introduction) or cannot be typed at all (the `and_` and `xor_` functions given in (33) and (34) are the most eloquent examples).

There is still a lot of work to do to fill the gap with real-world programming languages. For example, our analysis cannot handle flow of information, as we discussed for the function `example14` in Section 4. 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 [43] 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 extensions 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 pure languages and we already discussed at length at the end of the previous section our plans to extend it to cope with side-effects. However, the ultimate solution of integrating type and effect analysis in a unique tool 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 $x_1x_2 \in t^\circ$ 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 $t\sigma \leq \neg t^\circ$. 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 $t^\circ \leq t\sigma_\circ$ and we refine for the positive branch the types of x_1, of x_2, and of x_1x_2 by applying $\sigma_\circ$ to their types. While the idea is clear, the technical details are quite involved, especially if we also want functions with intersection types and/or gradual typing. Nevertheless, our approach has an edge on systems that do not account for polymorphism. This needs a whole gamut of non trivial research that we plan to develop in the near future building on the work on polymorphic types for semantic subtyping [13] and the research on the definition of polymorphic languages with set-theoretic types by Castagna et al. [10,11,12] and Petrucciani [34].

CRediT authorship contribution statement

All authors equally contributed to the work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors thank Paul-André Melliès for his help on type ranking and Sam Tobin-Hochstadt and the other reviewers for their feedback and useful insight. This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045-DIGICOSME) operated by ANR as part of the program «Investissement d’Avenir» Idex Paris-Saclay (ANR-11-IDEX-0003-02), by the «Chaire Langages Dynamiques pour les Données» of the Paris-Saclay foundation, and by a Google PhD fellowship for the second author.

Appendix A. Proof of type soundness

We give in this section the complete formalization of the declarative type system as well as the proof of its type safety.

A.1. The declarative type system

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

$$[\textsc{Const}]\ \frac{}{\Gamma\vdash c:\boldsymbol{b}_c}\qquad [\textsc{App}]\ \frac{\Gamma\vdash e_1:t_1\to t_2\quad \Gamma\vdash e_2:t_1}{\Gamma\vdash e_1e_2:t_2}$$

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

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

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

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

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

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

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

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

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

A.2. Parallel semantics

One technical difficulty in the proof of the subject reduction property is that, when reducing an expression e into v in a type case, the expression e disappears (e is not a sub-expression of the test anymore) and, thus, we can no longer refine the expression e in the "then" and "else" branches (which might contain occurrences of e). To circumvent this issue, we introduce a notion of parallel reduction which essentially reduces all occurrences of a sub-expression appearing in a type cases also in the "then" and "else" branch at the same time.

The idea is to label each step of reduction done by a context rule with the inner *notion of reduction* (defined below) that caused the context to reduce. In case of a reduction of the expression tested in the type case, that same reduction is applied in parallel to both branches. The semantics based on parallel reduction is given below where expressions and values are defined as in Section 2.3. The contexts, however, are not exactly those in Section 2.4 since there are two differences: (i) we remove the test expression context, since this requires a specific rule (rule $[\tau\kappa]$) that performs the parallel reduction and (*ii*) context holes are present only at top-level since the parallel reduction will handle the nesting of contexts by applying the rule $[\kappa]$ below multiple times. This yields the following definition:

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

For convenience, we denote $e \overset{e\mapsto e'}{\rightsquigarrow} e'$ by $e \overset{Id}{\rightsquigarrow} e'$ and by $e \overset{-}{\rightsquigarrow} e'$ a step of reduction of the parallel semantics, regardless of the value on the top of the arrow.

Notions of reduction:

$$[\beta]\ \frac{}{(\lambda^t x.e)v \overset{Id}{\rightsquigarrow} e\{x \mapsto v\}} \qquad [\pi]\ \frac{}{\pi_i(v_1,v_2) \overset{Id}{\rightsquigarrow} v_i}$$

$$[\tau_1]\ \frac{}{(v{\in}t)\ ?\ e_1 : e_2 \overset{Id}{\rightsquigarrow} e_1}\ v \in [\![t]\!]_{\mathcal{V}} \qquad [\tau_2]\ \frac{}{(v{\in}t)\ ?\ e_1 : e_2 \overset{Id}{\rightsquigarrow} e_2}\ v \notin [\![t]\!]_{\mathcal{V}}$$

Context reductions:

$$[\kappa]\ \frac{e \overset{e_r\mapsto e_r'}{\rightsquigarrow} e'}{C[e] \overset{e_r\mapsto e_r'}{\rightsquigarrow} C[e']}$$

$$[\tau\kappa]\ \frac{e \overset{e_r\mapsto e_r'}{\rightsquigarrow} e'}{(e{\in}t)\ ?\ e_1 : e_2 \overset{Id}{\rightsquigarrow} (e\{e_r \mapsto e_r'\}{\in}t)\ ?\ e_1\{e_r \mapsto e_r'\} : e_2\{e_r \mapsto e_r'\}}$$

where

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

with

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

$$[\textsc{Pair}]\ \frac{\vdash_{\mathcal{V}} v_1 : t_1 \qquad \vdash_{\mathcal{V}} v_2 : t_2}{\vdash_{\mathcal{V}} (v_1,v_2) : t_1 \times t_2} \qquad [\textsc{Abs}]\ \frac{t = (\wedge_{i\in I} s_i \to t_i) \wedge (\wedge_{j\in J} \neg(s_j' \to t_j')) \qquad t \not\leq \mathbb{0}}{\vdash_{\mathcal{V}} \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : t}$$

Here is a couple of examples of reduction using the parallel semantics:

$$[\tau\kappa]\ \dfrac{[\kappa]\ \dfrac{[\beta]\ \dfrac{}{(\lambda x.\ x+1)\ 1 \overset{Id}{\rightsquigarrow} 2}}{((\lambda x.\ x+1)\ 1, \mathsf{true}) \overset{(\lambda x.\ x+1)1\ \mapsto\ 2}{\rightsquigarrow} (2, \mathsf{true})}}{(((\lambda x.\ x+1)\ 1, \mathsf{true}){\in}\mathsf{Int} \times \mathsf{Bool})\ ?\ (\lambda x.\ x+1)\ 1 : 0 \overset{Id}{\rightsquigarrow} ((2, \mathsf{true}){\in}\mathsf{Int} \times \mathsf{Bool})\ ?\ 2 : 0}$$

and

$$[\tau_1]\ \frac{}{((2, \mathsf{true}){\in}\mathsf{Int} \times \mathsf{Bool})\ ?\ 2 : 0 \overset{Id}{\rightsquigarrow} 2}\ (2, \mathsf{true}) \in [\![\textsc{Int} \times \textsc{Bool}]\!]_{\mathcal{V}}$$

Notice that the rule $[\kappa]$ applies a substitution from an expression to an expressions (rather than from a variable to an expressions). This is formally defined as follows:

Definition Appendix A.1 *(Expression substitutions).* Expression substitutions, ranged over by ρ, map an expression into another expression. The application of an expressions substitution ρ to an expression e, noted $e\rho$ is the capture avoiding replacement defined as follows:

- If $e' \equiv_\alpha e''$, then $e''\{e' \mapsto e\} = e$.
- If $e' \not\equiv_\alpha e''$, then $e''\{e' \mapsto e\}$ is inductively defined as

$$
\begin{aligned}
c\{e' \mapsto e\} &= c \\
x\{e' \mapsto e\} &= x \\
(e_1 e_2)\{e' \mapsto e\} &= (e_1\{e' \mapsto e\})(e_2\{e' \mapsto e\}) \\
(\lambda^{\wedge_{i\in I} s_i \to t_i} x.e)\{e' \mapsto e\} &= \lambda^{\wedge_{i\in I} s_i \to t_i} x.(e\{e' \mapsto e\}) && \text{if } x \notin \mathsf{fv}(e) \cup \mathsf{fv}(e') \\
(\pi_i e)\{e' \mapsto e\} &= \pi_i(e\{e' \mapsto e\}) \\
(e_1, e_2)\{e' \mapsto e\} &= (e_1\{e' \mapsto e\}, e_2\{e' \mapsto e\}) \\
((e_1 \in t) \mathrel{?} e_2 : e_3)\{e' \mapsto e\} &= (e_1\{e' \mapsto e\} \in t) \mathrel{?} e_2\{e' \mapsto e\} : e_3\{e' \mapsto e\}
\end{aligned}
$$

Notice that the expression substitutions 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 operate up to alpha-renaming, too.

Finally notice that according to the definition above the rule $[\tau\kappa]$ could be equivalently written as follows:

$$
[\tau\kappa]\ \frac{e \overset{\rho}{\rightsquigarrow} e'}{(e \in t) \mathrel{?} e_1 : e_2 \overset{Id}{\rightsquigarrow} ((e \in t) \mathrel{?} e_1 : e_2)\rho}
$$

All the proofs below will use the parallel semantics instead of the standard semantics (of Section 2.4). However, the safety of the type system for the standard semantics can be deduced from the safety of the type system for the parallel semantics, using the following lemma:

Lemma Appendix A.2. $\forall e, v.\ e \mathrel{\bar{\rightsquigarrow}}^* v \Rightarrow e \rightsquigarrow^* v$.

Proof. This is a known result for the λ−calculus (even extended with conditional, and basic types), obtained using the Tait and Martin-Löf technique ([2]). See for instance [40] and [29]. The additional substitutions made by the rule $[\tau\kappa]$ will be performed later with the standard semantics. □

A.3. Proofs for the declarative type system

In this section, 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 judgment also use a well-formed environment. It is a consequence of the fact that rule [Case] requires e to be typeable and that it only refines subexpressions of e.

A.3.1. Environments

Definition Appendix A.3 *(Well-formed environment).* We say that an environment Γ is *well-formed* if and only if $\forall e \in \mathsf{dom}(\Gamma)$ such that 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 than the one recorded in Γ).

Definition Appendix A.4 *(Bottom environment).* Let Γ be an environment.
Γ is bottom (noted $\Gamma = \bot$) if and only if $\exists e \in \mathsf{dom}(\Gamma).\ \Gamma(e) \simeq \mathbb{0}$.

Definition Appendix A.5 *(Pre)order on environments.* Let Γ and Γ' be two environments. We write $\Gamma' \leq \Gamma$ if and only if:

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

This relation is a preorder (proof below).

Definition Appendix A.6 *(Application of a substitution to an environment).* Let Γ be an environment and ρ a substitution from expressions to expressions. The environment $\Gamma\rho$ is defined by:

$$\begin{aligned}
&\mathsf{dom}(\Gamma\rho) = \mathsf{dom}(\Gamma)\rho\\
&\forall e \in \mathsf{dom}(\Gamma\rho),\, (\Gamma\rho)(e) = \bigwedge_{\{e' \in \mathsf{dom}(\Gamma)\ \mid\ e'\rho \equiv e\}} \Gamma(e')
\end{aligned}$$

Definition Appendix A.7 *(Ordinary environments).* We say that an environment Γ is ordinary if and only if its domain only contains variables.

A.3.2. Subject reduction

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

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

Proof. See theorem 5.5, lemmas 6.19, 6.22, 6.23 of [19]. □

Lemma Appendix A.8 *(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 $[\tau\kappa]$ are up to alpha-renaming. □

Lemma Appendix A.9 *(Soundness and completeness of value typing). Let v be a value, t a type, and Γ an environment.*

- *If $\Gamma \vdash v : t$ and $\Gamma \neq \bot$, then $v \in [\![t]\!]_{\mathcal{V}}$.*
- *If $v \in [\![t]\!]_{\mathcal{V}}$ and v is well-typed in Γ, then $\Gamma \vdash v : t$.*

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

Lemma Appendix A.10 *(Monotonicity). Let Γ and Γ' be two environments such that $\Gamma' \leq \Gamma$. Then, we have:*

$$\begin{aligned}
\forall e, t.\ &\Gamma \vdash e : t \Rightarrow \Gamma' \vdash e : t\\
\forall e, t, \Gamma_1.\ &\Gamma \vdash^{\mathit{Env}}_{e,t} \Gamma_1 \Rightarrow \exists {\Gamma_1}' \leq \Gamma_1.\ \Gamma' \vdash^{\mathit{Env}}_{e,t} {\Gamma_1}'\\
\forall e, t, \varpi, t'.\ &\vdash^{\mathit{Path}}_{\Gamma,e,t} \varpi : t' \Rightarrow \vdash^{\mathit{Path}}_{\Gamma',e,t} \varpi : t'
\end{aligned}$$

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 Appendix A.11 *(Preorder relation). The relation $\leq$ on environments is a preorder.*

Lemma Appendix A.12 *(Value refinement 1). If we have $\vdash^{\mathit{Path}}_{\Gamma,e,t} \varpi.x : t'$ 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 $\vdash^{\mathit{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.*

Proof. We proceed by induction on the derivation of $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi.x : t'$.

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 us 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi.1 : t_1$, 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : t_2$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi.0 : t_1 \to t_2$, 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : t_2'$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : t_1 \times t_2$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : t'$ and using the rules [PInter] and [PTypeof], we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : \mathbb{0}$.

[PSnd] Similar to the previous case. □

Corollary Appendix A.13 *(Value refinement 2). For any derivable judgement of the form $\Gamma \vdash^{Env}_{e,t} \Gamma'$ (with e well-typed in Γ), we can construct a derivation of $\Gamma \vdash^{Env}_{e,t} \Gamma''$ 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 Appendix A.14 *(Value testing). For any derivable judgement of the form $\Gamma \vdash^{Env}_{v,t} \Gamma'$ (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 $\vdash^{\texttt{Path}}_{\Gamma,v,t} \varpi : t'$ 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 Appendix A.15 *(Substitution). Let Γ be an environment. Let e_a and e_b be two expressions.*

Let us 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)*
- *$v\,v$ (application of two values)*
- *(v, v) (product of two values)*

Let us 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 be 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:

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

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

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

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 $e \in \mathsf{dom}(\Gamma)$, then we have $e\rho \in \mathsf{dom}(\Gamma\rho)$ 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 $e \in \mathsf{dom}(\Gamma)$ 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 $(e_1 e_2)\rho = (e_1\rho)(e_2\rho)$ (because $e_1 e_2 \neq e_a$). Thus we can directly conclude by using the induction hypothesis.

[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 $\Gamma \vdash^{\texttt{Env}}_{e,t_{if}} \Gamma_1$. We get $\Gamma\rho \vdash e\rho : t_0$, $\Gamma\rho \vdash^{\texttt{Env}}_{e\rho,t_{if}} \Gamma_1\rho$ 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 $\Gamma \vdash^{\texttt{Env}}_{e,\neg t_{if}} \Gamma_2$ 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 $\Gamma\rho \vdash^{\texttt{Env}}_{e\rho,t} \Gamma'\rho$ and $\forall t''.\ \Gamma' \vdash e_a : t'' \Rightarrow \Gamma'\rho \vdash e_b : t''$.
First, let's show that we can derive $\Gamma\rho \vdash^{\texttt{Env}}_{e\rho,t} \Gamma''\rho$ with $\Gamma'' = \Gamma', (e{\downarrow}\varpi : t')$.
There are two cases:

- $e{\downarrow}\varpi$ is a strict sub-expression of e_a.
  In this case, it means that among its three possible forms, e_a is of the form $v\,v$ 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 occurrence of e_a, so it is easy to conclude.
- $e{\downarrow}\varpi$ is not a strict sub-expression of e_a.
  In this case, we know that $e\rho{\downarrow}\varpi$ is defined.
  Thus we can apply the induction hypothesis on $\vdash^{\texttt{Path}}_{\Gamma',e,t} \varpi : t'$. It gives $\vdash^{\texttt{Path}}_{\Gamma'\rho,e\rho,t} \varpi : t'$. If $e\rho{\downarrow}\varpi \in \mathsf{dom}(\Gamma'\rho)$, and $(\Gamma'\rho)(e\rho{\downarrow}\varpi) = t'' \not\geq t'$, then we can derive $\vdash^{\texttt{Path}}_{\Gamma'\rho,e\rho,t} \varpi : t' \wedge t''$ just by using the rules [PInter], [PTypeof] and [Env].
  Using this last judgement together with $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$, we can derive with the rule [Path] the wanted $\Gamma\rho \vdash^{\texttt{Env}}_{e\rho,t} \Gamma''\rho$.

Now, let's show that $\forall t'.\ \Gamma'' \vdash e_a : t' \Rightarrow \Gamma''\rho \vdash e_b : t'$.
Let t' be 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 us suppose $\Gamma'' \neq \bot$.
Let us 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 sub-expression of e_a, it must be a value (given the possible forms of e_a). Moreover, as $\Gamma'' \neq \bot$, we have $\forall v \in \mathsf{dom}(\Gamma'').\ v \in [\![\Gamma''(v)]\!]_{\mathcal{V}}$ (recall the assumptions at the beginning of the proof) and thus $\forall v \in \mathsf{dom}(\Gamma'').\ \Gamma' \vdash v : \Gamma''(v)$. 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 $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma'')$, 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 us 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 sub-expressions 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⋯] All the remaining rules are trivial. □

Theorem Appendix A.16 *(Subject reduction). Let Γ be 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 be 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 [19]). Let us 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 $\mathsf{dom}(\Gamma)$). Let us 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 $e \overset{Id}{\rightsquigarrow} v_i$. 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 $\Gamma \not\equiv \bot$ and Γ is ordinary so there is no pair in $\mathsf{dom}(\Gamma)$). 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 $\Gamma \vdash e_0 : t_0$, $\Gamma \vdash^{\texttt{Env}}_{e_0,t_{if}} \Gamma'$ 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)

  – $v\,v$ (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 Appendix A.17 *(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 [19]. □

Theorem Appendix A.18 *(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 $[\kappa]$.

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 $[\beta]$ 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 $[\tau\kappa]$. Otherwise, by using the induction hypothesis we get that e_0 is a value. In consequence, e is reducible (the reduction rule $[\tau_i]$ 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 $[\kappa]$. 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 $[\pi]$ 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 $[\kappa]$. Otherwise, by using the induction hypothesis we get that both e_1 and e_2 are values. In consequence, e is also a value. □

Appendix B. Typing algorithm: operators, type schemes, proofs of soundness and completeness

We give in this section a typing algorithm that uses types schemes and is more general than the one presented in the main body of the paper (that is, one whose completeness is not limited to positive expressions). We start by defining how to compute the "$\blacksquare$" operator and then we define type schemes and the algorithm. We prove that this algorithm (as well as the one in the main body of the paper) is sound w.r.t. to the declarative type system and that, under certain restrictions it is also complete.

B.1. Operator $\blacksquare$

In this section, we will use the algorithmic definition of $\blacksquare$ 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 Appendix B.1 *(Correctness of $\blacksquare$).* $\forall t, s.\ t \circ (\mathsf{dom}(t) \setminus (t \blacksquare s)) \leq \neg s$

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 $\blacksquare$).
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:

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

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 Appendix B.2 *($\blacksquare$ Alternative definition). The following algorithmic definition for $\blacksquare$ is equivalent to the previous one:*

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

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)\\
&\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)\qquad\square
\end{aligned}$$

Lemma Appendix B.3 *(Optimality of $\blacksquare$). Let t, s, two types. For any u such that $t\circ(\mathsf{dom}(t)\setminus u)\leq\neg s$, 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, $s \wedge \bigwedge_{p\in P} t_p \not\simeq \mathbb{0}$) and such that $a_{i,P} \not\simeq \mathbb{0}$. For convenience, let $a = a_{i,P}$. We just have to show that $a \leq u$.

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

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:

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

As $P_i \setminus Q = P$, we immediately have $s \wedge \bigwedge_{p\in P_i\setminus Q} t_p \not\simeq \mathbb{0}$.

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:

$$(a \setminus u) \wedge \bigwedge_{q\in Q} \neg s_q \simeq (a \wedge \bigwedge_{q\in Q} \neg s_q) \setminus u \simeq a \setminus u \not\simeq \mathbb{0}$$

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 immediately obtain the remaining inequality. □

Theorem Appendix B.4 *(Characterization of ▪).* $\forall t, s.\ t \blacksquare s = \min\{u \mid t \circ (\mathsf{dom}(t) \setminus u) \leq \neg s\}$.

Proof. Immediate consequence of the previous results. □

B.2. Type schemes

We introduce for the proofs the notion of *type schemes* and we define a more powerful algorithmic type system that uses them. It allows us to have a stronger (but still partial) completeness theorem.

The proofs for the algorithmic type system presented in 2.6.3 can be derived from the proofs of this section (see Section B.5).

B.2.1. Type schemes

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

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

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

Definition Appendix B.5 *(Interpretation of type schemes).* We define the function $\{\!\!\{\_\}\!\!\}$ that maps type schemes into sets of types.

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

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 Appendix B.6 *([19]). 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 2.4 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 $v \in t \overset{\text{def}}{\iff} \mathsf{typeof}(v) \leq t$.

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 $\vdash^{\texttt{Path}}$). For our algorithmic system (see [$\textsc{Env}_{\mathcal{A}}$] in Section 2.6.3) all we need to define is the intersection of a type scheme with a type:

Lemma Appendix B.7 *([19]). Let $\mathbb{t}$ be a type scheme and t a type. We can compute a type scheme, written $t \owedge \mathbb{t}$, such that $\{\!\!\{t \owedge \mathbb{t}\}\!\!\} = \{s \mid \exists t' \in \{\!\!\{\mathbb{t}\}\!\!\}.\ t \wedge t' \leq s\}$*

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 Appendix B.8 *(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.

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

Type schemes are already present in the theory of semantic subtyping presented in [19, Section 6.11]. In particular, it explains how the operators such as $\circ$, $\boldsymbol{\pi_1}(t)$ and $\boldsymbol{\pi_2}(t)$ can be extended to type schemes (see also [6, §4.4] for a detailed description).

B.3. Algorithmic type system with type schemes

We present here a refinement of the algorithmic type system presented in 2.6.3 that associates to an expression a type scheme instead of a regular type. This allows to type expressions more precisely and thus to have a more powerful (but still partial) completeness theorem in regards to the declarative type system.

The results about this new type system will be used in Appendix B.5 in order to obtain a soundness and completeness theorem for the algorithmic type system presented in 2.6.3.

$$
[\textsc{Efq}_{\mathcal{A}_{\textsc{ts}}}]\ \frac{}{\Gamma, (e:\mathbb{0}) \vdash_{\mathcal{A}_{\text{ts}}} e' : \mathbb{0}}\ \begin{array}{c}\text{WITH PRIORITY OVER}\\ \text{ALL THE OTHER RULES}\end{array}
\qquad
[\textsc{Var}_{\mathcal{A}_{\textsc{ts}}}]\ \frac{}{\Gamma \vdash_{\mathcal{A}_{\text{ts}}} x : \Gamma(x)}\ x \in \mathrm{dom}(\Gamma)
$$

$$
[\textsc{Env}_{\mathcal{A}_{\textsc{ts}}}]\ \frac{\Gamma \setminus \{e\} \vdash_{\mathcal{A}_{\text{ts}}} e : \mathbb{t}}{\Gamma \vdash_{\mathcal{A}_{\text{ts}}} e : \Gamma(e) \owedge \mathbb{t}}\ e \in \mathrm{dom}(\Gamma) \text{ AND } e \text{ NOT A VARIABLE}
\qquad
[\textsc{Const}_{\mathcal{A}_{\textsc{ts}}}]\ \frac{}{\Gamma \vdash_{\mathcal{A}_{\text{ts}}} c : \boldsymbol{b}_c}\ c \notin \mathrm{dom}(\Gamma)
$$

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

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

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

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

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

$$\mathsf{Constr}_{\Gamma,e,t}(\epsilon) = t \tag{B.1}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.0) = \neg(\mathsf{Intertype}_{\Gamma,e,t}(\varpi.1) \to \neg\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{B.2}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.1) = \mathsf{Repr}(\mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi.0)) \blacksquare \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \tag{B.3}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.l) = \boldsymbol{\pi}_{\mathbf{1}}(\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{B.4}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.r) = \boldsymbol{\pi}_{\mathbf{2}}(\mathsf{Intertype}_{\Gamma,e,t}(\varpi)) \tag{B.5}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.f) = \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \times \mathbb{1} \tag{B.6}$$

$$\mathsf{Constr}_{\Gamma,e,t}(\varpi.s) = \mathbb{1} \times \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \tag{B.7}$$

$$\mathsf{Intertype}_{\Gamma,e,t}(\varpi) = \mathsf{Repr}(\mathsf{Constr}_{\Gamma,e,t}(\varpi) \owedge \mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi)) \tag{B.8}$$

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

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

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

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

B.4. Proofs for the algorithmic type system with type schemes

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

Note that, now that we have type schemes, use a different but more convenient definition for $\mathsf{typeof}_{\Gamma}(e)$ that the one in Section 2.6.2:

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

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.

B.4.1. Soundness

Theorem Appendix B.9 *(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:

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

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

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

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 [Var$_{\mathcal{A}_{\text{TS}}}$]. 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 [Env$_{\mathcal{A}_{\text{TS}}}$]. 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 [Const$_{\mathcal{A}_{\text{TS}}}$]. 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 [Abs$_{\mathcal{A}_{\text{TS}}}$]. 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 [App$_{\mathcal{A}_{\text{TS}}}$]. 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 [Proj$_{\mathcal{A}_{\text{TS}}}$]. 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 [Pair$_{\mathcal{A}_{\text{TS}}}$]. We conclude easily with the induction hypothesis and the rules [Subs] and [Pair].

$e = (e_0 \in t)\ \mathbf{?}\ e_1\ \mathbf{:}\ e_2$ The last rule is [Case$_{\mathcal{A}_{\text{TS}}}$]. 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 $\mathsf{Intertype}_{\Gamma,e,t}(\varpi) = \mathsf{Repr}(\mathsf{Constr}_{\Gamma,e,t}(\varpi) \owedge \mathsf{typeof}_{\Gamma}(e{\downarrow}\varpi))$.

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

Now we have to derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi : \mathsf{Constr}_{\Gamma,e,t}(\varpi)$ (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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi'.0 : f$ and $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : s$.
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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi'.0 : f$, we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi'.0 : t_1 \to t_2$. 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi'.0 : t_1 \to t_2$ and $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : s$.

$\varpi = \varpi'.0$ By using the inner induction hypothesis and the previous case we've just proved, we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$ and $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi'.1 : \mathsf{Intertype}_{\Gamma,e,t}(\varpi'.1)$. 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$, and thus using the rule [PSubs] we can derive $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : t_1 \times \mathbb{1}$. 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 $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$, so we can conclude by applying [PFst].

$\varpi = \varpi'.s$ The inner induction hypothesis gives $\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi' : \mathsf{Intertype}_{\Gamma,e,t}(\varpi')$, 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 $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$ is derivable.

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

- the initial [Base] rule has been removed in order to be able to do the junction,
- all the Γ' at the left of $\vdash^{\texttt{Env}}_{e,t}$ 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 $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma'$.

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

Let's recall a monotonicity property: for any Γ_1 and Γ_2 such that $\Gamma_2 \leq \Gamma_1$, we have $\forall t'.\ \vdash^{\texttt{Path}}_{\Gamma_1,e,t} \varpi : t' \Rightarrow \vdash^{\texttt{Path}}_{\Gamma_2,e,t} \varpi : t'$. Moreover, when we also have $e{\downarrow}\varpi \in \mathsf{dom}(\Gamma_2)$, we can derive $\vdash^{\texttt{Path}}_{\Gamma_2,e,t} \varpi : t' \wedge \Gamma_2(e{\downarrow}\varpi)$ (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 $\Gamma_{\varpi} \leq \Gamma$):

$$\begin{array}{lll} \text{If } e{\downarrow}\varpi \in \mathsf{dom}(\Gamma_\varpi) & \vdash^{\texttt{Path}}_{\Gamma_\varpi,e,t} \varpi : \mathsf{Intertype}_{\Gamma,e,t}(\varpi) \wedge \Gamma_\varpi(e{\downarrow}\varpi) & \Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma_\varpi \\ \text{Otherwise} & \vdash^{\texttt{Path}}_{\Gamma_\varpi,e,t} \varpi : \mathsf{Intertype}_{\Gamma,e,t}(\varpi) & \Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma_\varpi \end{array}$$

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

B.4.2. Completeness

Definition Appendix B.10 *(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 Appendix B.11 *(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:

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

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

Definition Appendix B.12 *(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 $\{\!\!\{\mathbb{t}_1\}\!\!\} \subseteq \{\!\!\{\mathbb{t}_2\}\!\!\}$.

Lemma Appendix B.13. *When well-defined, the following inequalities hold:*

$$\forall t, \mathbb{t}.\ \mathsf{Repr}(t \owedge \mathbb{t}) \leq t \wedge \mathsf{Repr}(\mathbb{t})$$

$$\forall t_1, t_2, \mathbb{t}_1, \mathbb{t}_2.\ t_1 \leq t_2 \textit{ and } \mathbb{t}_1 \leq \mathbb{t}_2 \textit{ and } \mathsf{Repr}(\mathbb{t}_1) \leq \mathsf{Repr}(\mathbb{t}_2) \Rightarrow \mathsf{Repr}(t_1 \owedge \mathbb{t}_1) \leq \mathsf{Repr}(t_2 \owedge \mathbb{t}_2)$$

$$\forall \mathbb{t}_1, \mathbb{t}_2.\ \mathsf{Repr}(\mathbb{t}_1 \circ \mathbb{t}_2) \leq \mathsf{Repr}(\mathbb{t}_1) \circ \mathsf{Repr}(\mathbb{t}_2)$$

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

Lemma Appendix B.14 *(Monotonicity of the algorithm). Let Γ, Γ' and e such that $\Gamma' \leq^e_{\mathcal{A}} \Gamma$ and $\mathsf{typeof}_\Gamma(e) \neq \Omega$. We have:*

$$\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_\Gamma(e) \textit{ and } \mathsf{Repr}(\mathsf{typeof}_{\Gamma'}(e)) \leq \mathsf{Repr}(\mathsf{typeof}_\Gamma(e))$$

$$\forall t, \varpi.\ \mathsf{Intertype}_{\Gamma',e,t}(\varpi) \leq \mathsf{Intertype}_{\Gamma,e,t}(\varpi)$$

$$\forall t.\ \mathsf{Refine}_{e,t}(\Gamma') \leq^e_{\mathcal{A}} \mathsf{Refine}_{e,t}(\Gamma)$$

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 $[\textsc{Abs}_{\mathcal{A}_{\text{TS}}}]$.)

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

If $e = x$ is a variable, then the last rule used in $\mathsf{typeof}_\Gamma(e)$ and $\mathsf{typeof}_{\Gamma'}(e)$ is $[\textsc{Var}_{\mathcal{A}_{\text{TS}}}]$. As $\Gamma' \leq^e_{\mathcal{A}} \Gamma$, we have $\Gamma'(e) \leq \Gamma(e)$ and thus we can conclude with the rule $[\textsc{Var}_{\mathcal{A}_{\text{TS}}}]$. 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 $[\textsc{Env}_{\mathcal{A}_{\text{TS}}}]$. As $\Gamma' \leq^e_{\mathcal{A}} \Gamma$, 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 $\Gamma' \setminus \{e\} \leq^e_{\mathcal{A}} \Gamma \setminus \{e\}$).

- If we have $e \in \mathsf{dom}(\Gamma')$, we have according to the rule $[\textsc{Env}_{\mathcal{A}_{\text{TS}}}]$ $\mathsf{typeof}_{\Gamma'}(e) \leq \mathsf{typeof}_{\Gamma' \setminus \{e\}}(e) \leq \mathsf{typeof}_{\Gamma \setminus \{e\}}(e)$.
  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 $[\textsc{Env}_{\mathcal{A}_{\text{TS}}}]$ for $\mathsf{typeof}_{\Gamma'}(e)$. As $\Gamma' \setminus \{e\} \leq^e_{\mathcal{A}} \Gamma \setminus \{e\} = \Gamma$, 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 $[\textsc{Const}_{\mathcal{A}_{\text{TS}}}]$. 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$_{\mathcal{A}_{\text{TS}}}$]. We have $\forall i\in I.\ \Gamma',(x:s_i)\leq_{\mathcal{A}}^{e'}\Gamma,(x:s_i)$ (quite straightforward) so by applying the induction hypothesis we have $\forall i\in I.\ \textsf{typeof}_{\Gamma',(x:s_i)}(e')\leq\textsf{typeof}_{\Gamma,(x:s_i)}(e')$.

$e=e_1e_2$ The last rule is [App$_{\mathcal{A}_{\text{TS}}}$]. We can conclude immediately by using the induction hypothesis and noticing that $\circ$ is monotonic for both of its arguments.

$e=\pi_ie'$ The last rule is [Proj$_{\mathcal{A}_{\text{TS}}}$]. 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$_{\mathcal{A}_{\text{TS}}}$]. We can conclude immediately by using the induction hypothesis.

$e=(e_0{\in}t)\ \textbf{?}\ e_1:e_2$ The last rule is [Case$_{\mathcal{A}_{\text{TS}}}$]. By using the induction hypothesis we get $\textsf{Refine}_{e_0,t}(\Gamma')\leq_{\mathcal{A}}^{e_0}\textsf{Refine}_{e_0,t}(\Gamma)$. We also have $\Gamma'\leq_{\mathcal{A}}^{e_1}\Gamma$ (as e_1 is a subexpression of e).

From those two properties, let's show that we can deduce $\textsf{Refine}_{e_0,t}(\Gamma')\leq_{\mathcal{A}}^{e_1}\textsf{Refine}_{e_0,t}(\Gamma)$:

Let $e'\in\textsf{dom}(\textsf{Refine}_{e_0,t}(\Gamma))$ a subexpression of e_1.

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

So we have $\textsf{Refine}_{e_0,t}(\Gamma')\leq_{\mathcal{A}}^{e_1}\textsf{Refine}_{e_0,t}(\Gamma)$. Consequently, we can apply the induction hypothesis again to get $\textsf{typeof}_{\textsf{Refine}_{e_0,t}(\Gamma')}(e_1)\leq\textsf{typeof}_{\textsf{Refine}_{e_0,t}(\Gamma)}(e_1)$.

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 $\forall t_1,t_2,\mathbb{t}_1,\mathbb{t}_2.\ t_1\leq t_2$ and $\mathbb{t}_1\leq\mathbb{t}_2$ and $\textsf{Repr}(\mathbb{t}_1)\leq\textsf{Repr}(\mathbb{t}_2)\Rightarrow\textsf{Repr}(t_1\owedge\mathbb{t}_1)\leq\textsf{Repr}(t_2\owedge\mathbb{t}_2)$ (lemma above).

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

$$\textsf{Constr}_{\Gamma',e,t}(\varpi)\leq\textsf{Constr}_{\Gamma,e,t}(\varpi)$$

$$\textsf{typeof}_{\Gamma'}(e{\downarrow}\varpi)\leq\textsf{typeof}_{\Gamma}(e{\downarrow}\varpi)$$

$$\textsf{Repr}(\textsf{typeof}_{\Gamma'}(e{\downarrow}\varpi))\leq\textsf{Repr}(\textsf{typeof}_{\Gamma}(e{\downarrow}\varpi))$$

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 $\textsf{Constr}_{\Gamma',e,t}(\varpi)\leq\textsf{Constr}_{\Gamma,e,t}(\varpi)$. 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 $(t_1\blacksquare t')\wedge\textsf{dom}(t_2)\leq t_2\blacksquare t'$. By the absurd, let's suppose it is not true. Let's note $t''=(t_1\blacksquare t')\wedge\textsf{dom}(t_2)$. Then we have $t''\leq\textsf{dom}(t_2)\leq\textsf{dom}(t_1)$ 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 $t_1=\textsf{Repr}(\textsf{typeof}_{\Gamma'}(e{\downarrow}\varpi'.0))$ and $t_2=\textsf{Repr}(\textsf{typeof}_{\Gamma}(e{\downarrow}\varpi'.0))$ and $t'=\textsf{Intertype}_{\Gamma,e,t}(\varpi')$. As e is well-typed, and using the inner induction hypothesis, we have $\textsf{Repr}(\textsf{typeof}_{\Gamma'}(e{\downarrow}\varpi'.1))\leq\textsf{Repr}(\textsf{typeof}_{\Gamma}(e{\downarrow}\varpi'.1))\leq\textsf{dom}(t_2)$. Thus, using this property, we get:

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

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

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

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 $\bigwedge_{\{\varpi\ \mid\ e\downarrow\varpi\equiv e'\}}\textsf{Intertype}_{\Gamma',e,t}(\varpi)\leq\bigwedge_{\{\varpi\ \mid\ e\downarrow\varpi\equiv e'\}}\textsf{Intertype}_{\Gamma,e,t}(\varpi)$.

The rest follows. □

Definition Appendix B.15 *(Positive derivation).* A derivation of the declarative type system is said positive iff it does not contain any rule [Abs-].

Theorem Appendix B.16 *(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:

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

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

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$_{\mathcal{A}_{\text{TS}}}$]. 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$_{\mathcal{A}_{\text{TS}}}$]. 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$_{\mathcal{A}_{\text{TS}}}$], 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$_{\mathcal{A}_{\text{TS}}}$]. Thus this case is trivial.

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

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

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

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

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

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

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

[Path] We have $\Gamma' = \Gamma_1, (e{\downarrow}\varpi : t')$. By applying the induction hypothesis on the premise $\Gamma \vdash^{\texttt{Env}}_{e,t} \Gamma_1$, we have $\mathsf{RefineStep}^n_{e,t}(\Gamma) = \Gamma_2$ with $\Gamma_2 \leq_{\mathcal{A}} \Gamma_1$ for a certain n.

We now proceed by induction on the derivation $\vdash^{\texttt{Path}}_{\Gamma_1,e,t} \varpi : t'$ to show that we can obtain $\mathsf{Intertype}_{\Gamma'',e,t}(\varpi) \leq t'$ with $\Gamma'' = \mathsf{RefineStep}^{n'}_{e,t}(\Gamma_2)$ 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:

$$\mathsf{Intertype}_{\Gamma''_1,e,t}(\varpi) \leq t_1$$

$$\mathsf{Intertype}_{\Gamma''_2,e,t}(\varpi) \leq t_2$$

$$\mathsf{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_{\mathcal{A}} \Gamma''_1$$

$$\mathsf{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_{\mathcal{A}} \Gamma''_2$$

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

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

[PEPS] Trivial.

[PAPPR] By using the induction hypothesis we get:

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

By taking $n' = \max(n_1, n_2) + 1$, we can have $\Gamma'' = \mathsf{RefineStep}^{n'}_{e,t}(\Gamma_2)$ with $\Gamma'' \leq_{\mathcal{A}} \mathsf{RefineStep}_{e,t}(\Gamma_1'')$ and $\Gamma'' \leq_{\mathcal{A}} \mathsf{RefineStep}_{e,t}(\Gamma_2'')$.
In consequence, we have $\mathsf{Repr}(\mathsf{typeof}_{\Gamma''}(e{\downarrow}\varpi.0)) \leq \mathsf{Intertype}_{\Gamma_1'',e,t}(\varpi.0) \leq t_1 \to t_2$ (by definition of $\mathsf{RefineStep}_{e,t}$). We also have, by monotonicity, $\mathsf{Intertype}_{\Gamma'',e,t}(\varpi) \leq t_2'$.
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:

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

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

[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. □

From this result, we will now prove a stronger but more complex completeness theorem. We were not able to prove full completeness, just a partial form of it. Indeed, the use of nested [PAPPL] yields a precision that the algorithm loses by applying Repr() in the definition of Constr. Completeness is recovered by forbidding nested negated arrows on the left-hand side of negated arrows.

Definition Appendix B.17 *(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 ($\vdash^{\texttt{Path}}_{\Gamma,e,t} \varpi.1 : t_1$).

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.

Lemma Appendix B.18. *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$_{\mathcal{A}_{\text{TS}}}$], [Env$_{\mathcal{A}_{\text{TS}}}$] nor [App$_{\mathcal{A}_{\text{TS}}}$] can produce a type containing a constructor $[\cdots]$. □

Theorem Appendix B.19 *(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:

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

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

Proof. This proof is done by induction. It 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 [Abs$_{\mathcal{A}_{\text{TS}}}$]. 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 $\Gamma \vdash^{\mathtt{Env}}_{e,t} \Gamma'$.

[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 $\Gamma \vdash^{\mathtt{Env}}_{e,t} \Gamma_1$, 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 $\vdash^{\mathtt{Path}}_{\Gamma_1,e,t} \varpi : t'$ that $\mathsf{Intertype}_{\Gamma'',e,t}(\varpi) \owedge \mathsf{typeof}_{\Gamma''}(e{\downarrow}\varpi) \leq t'$.

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 $\Gamma' = \bot \Rightarrow \Gamma_{n_o} = \bot$).

[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:

$$\mathsf{Intertype}_{\Gamma''_1,e,t}(\varpi.0) \owedge \mathsf{typeof}_{\Gamma''_1}(e{\downarrow}\varpi.0) \leq t_1 \to t_2$$

$$\mathsf{Intertype}_{\Gamma''_2,e,t}(\varpi) \owedge \mathsf{typeof}_{\Gamma''_2}(e{\downarrow}\varpi) \leq t'_2$$

$$t_2 \wedge t'_2 \simeq \mathbb{0}$$

$$\mathsf{RefineStep}^{n_1}_{e,t}(\Gamma_1) \leq_{\mathcal{A}} \Gamma''_1$$

$$\mathsf{RefineStep}^{n_2}_{e,t}(\Gamma_2) \leq_{\mathcal{A}} \Gamma''_2$$

Moreover, as $e{\downarrow}\varpi$ is an application, we can use the lemma above to deduce $\mathsf{Intertype}_{\Gamma''_2,e,t}(\varpi) \owedge \mathsf{typeof}_{\Gamma''_2}(e{\downarrow}\varpi) = \mathsf{Intertype}_{\Gamma''_2,e,t}(\varpi)$ (see definition of $\mathsf{Intertype}$).

Thus we have $\mathsf{Intertype}_{\Gamma''_2,e,t}(\varpi) \leq t'_2$.

We also have $\mathsf{Intertype}_{\Gamma''_1,e,t}(\varpi.0) \leq \mathsf{Repr}(\mathsf{Intertype}_{\Gamma''_1,e,t}(\varpi.0) \owedge \mathsf{typeof}_{\Gamma''_1}(e{\downarrow}\varpi.0)) \leq t_1 \to t_2$.

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:

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

Moreover, as $e{\downarrow}\varpi$ is an application, we can use the lemma above to deduce $\mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi) \owedge \mathsf{typeof}_{\Gamma_2''}(e{\downarrow}\varpi) = \mathsf{Intertype}_{\Gamma_2'',e,t}(\varpi)$ (see definition of $\mathsf{Intertype}$).
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.5. Proofs for the algorithmic type system without type schemes

In this section, we consider the algorithmic type system without type schemes, as defined in 2.6.3.

B.5.1. Soundness

Lemma Appendix B.20. *For every Γ, e, t, n_o, if $\Gamma \vdash_{\mathcal{A}} e : t$, then there exists $\mathbb{t} \leq t$ such that $\Gamma \vdash_{\mathcal{A}_{ts}} e : \mathbb{t}$.*

Proof. Straightforward induction over the structure of e. □

Theorem Appendix B.21 *(Soundness of the algorithmic type system without type schemes). For every Γ, e, t, n_o, if $\Gamma \vdash_{\mathcal{A}} e : t$, then $\Gamma \vdash e : t$.*

Proof. Trivial by using the Theorem Appendix B.9 and the previous lemma. □

B.5.2. Completeness

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

Lemma Appendix B.22. *If we restrict the language to positive expressions e_+, then we have the following property:*
$\forall \Gamma, e_+, \mathbb{t}.\ \Gamma \vdash_{\mathcal{A}_{ts}} e_+ : \mathbb{t} \Rightarrow \Gamma \vdash_{\mathcal{A}} e_+ : \mathsf{Repr}(\mathbb{t})$

Proof. We can prove it by induction over the structure of e_+.

The main idea of this proof is that, as e_+ is a positive expression, the rule [Abs-] is not needed anymore because the negative part of functional types (i.e. the N_i part of their DNF) becomes useless:

- 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$_{\mathcal{A}_{\text{TS}}}$]). □

Theorem Appendix B.23 *(Completeness of the algorithmic type system for positive expressions). For every type environment Γ and positive expression e_+, if $\Gamma \vdash e_+ : t$, then there exist n_o and t' such that $\Gamma \vdash_{\mathcal{A}} e_+ : t'$.*

Proof. Trivial by using the Theorem Appendix B.16 and the previous lemma. □

References

[1] Michael Backes, Cătălin Hriţcu, Matteo Maffei, Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations, J. Comput. Secur. 22 (2) (2014) 301–353, https://doi.org/10.3233/JCS-130493.

[2] Hendrik P. Barendregt, The Lambda Calculus Its Syntax and Semantics, vol. 103, revised ed., North Holland, 1984.

[3] Véronique Benzaken, Giuseppe Castagna, Alain Frisch, CDuce: an XML-centric general-purpose language, in: ICFP '03, 8th ACM International Conference on Functional Programming, Uppsala, Sweden, ACM Press, 2003, pp. 51–63.

[4] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, David Langworthy, Semantic subtyping with an SMT solver, SIGPLAN Not. 45 (9) (Sept. 2010) 105–116, https://doi.org/10.1145/1932681.1863560.

[5] Ambrose Bonnaire-Sergeant, Rowan Davies, Sam Tobin-Hochstadt, Practical optional types for clojure, in: Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, in: Lecture Notes in Computer Science, vol. 9632, Springer, 2016, pp. 68–94.

[6] Giuseppe Castagna, Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers), Log. Methods Comput. Sci. 16 (1) (2020) 15:1–15:58, https://doi.org/10.23638/LMCS-16(1:15)2020.

[7] Giuseppe Castagna, Victor Lanvin, Gradual typing with union and intersection types, Proc. ACM Program. Lang. 1 (ICFP) (Aug. 2017), https://doi.org/10.1145/3110285, Article 41, 28 pp.

[8] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, Jeremy G. Siek, Gradual typing: a new perspective, in: POPL '19 46th ACM Symposium on Principles of Programming Languages, Proc. ACM Program. Lang. 3 (Jan. 2019), https://doi.org/10.1145/3290329, Article 16, 32 pages.

[9] Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, Matthew Lutze, On type-cases, union elimination, and occurrence typing, Proc. ACM Program. Lang. 6 (POPL) (Jan. 2022), https://doi.org/10.1145/3498674, 31 pages.

[10] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Pietro Abate, Polymorphic functions with set-theoretic types. part 2: local type inference and type reconstruction, in: Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15), 2015, pp. 289–302.

[11] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, Luca Padovani, Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation, in: Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '14), 2014, pp. 5–17.

[12] Giuseppe Castagna, Tommaso Petrucciani, Kim Nguyen, Set-theoretic types for polymorphic variants, in: ICFP '16, 21st ACM SIGPLAN International Conference on Functional Programming, 2016, pp. 378–391.

[13] Giuseppe Castagna, Zhiwu Xu, Set-theoretic foundation of parametric polymorphism and subtyping, in: ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming, 2011, pp. 94–106.

[14] CDuce, The CDuce Compiler. CDuce, https://www.cduce.org.

[15] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, Gabriel Levi, Fast and precise type checking for JavaScript, Proc. ACM Program. Lang. 1 (OOPSLA) (Oct. 2017), https://doi.org/10.1145/3133872, Article 48, 30 pages.

[16] Ravi Chugh, Patrick M. Rondon, Ranjit Jhala, Nested refinements: a logic for duck typing, in: Proceedings of the 39th Annual ACM SIGPLAN Symposium on Principles of Programming Languages, Philadelphia, PA, USA, POPL '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 231–244.

[17] Facebook. Flow. Facebook, https://flow.org/.

[18] Alain Frisch, Théorie, conception et réalisation d'un langage de programmation adapté à XML, Ph.D. Dissertation, Université Paris 7 Denis Diderot, 2004, http://www.cduce.org/papers/frisch_phd.pdf.

[19] Alain Frisch, Giuseppe Castagna, Véronique Benzaken, Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types, J. ACM 55 (4) (Sept. 2008) 19:1–19:64, https://doi.org/10.1145/1391289.1391293.

[20] Google, Dart Programming Language, Specification Google, https://dart.dev/guides/language/spec.

[21] Michael Greenberg, The dynamic practice and static theory of gradual typing, in: Benjamin S. Lerner, Rastislav Bodík, Shriram Krishnamurthi (Eds.), 3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA, in: LIPIcs, vol. 136, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 6:1–6:20.

[22] Haruo Hosoya, Jérôme Vouillon, Benjamin C. Pierce, Regular expression types for XML, in: Proceedings of the International Conference on Functional Programming (ICFP), SIGPLAN Not. 35 (9) (2000).

[23] JetBrains, Kotlin documentation, Available at http://kotlinlang.org/docs/reference, 2018.

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

[25] Andrew M. Kent, David Kempe, Sam Tobin-Hochstadt, Occurrence typing modulo theories, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, CA, USA, PLDI '16, ACM, New York, NY, USA, 2016, pp. 296–309.

[26] Kenneth Knowles, Cormac Flanagan, Compositional reasoning and decidable checking for dependent contract types, in: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification (Savannah, GA, USA) (PLPV '09), Association for Computing Machinery, New York, NY, USA, 2009, pp. 27–38.

[27] Raghavan Komondoor, Ganesan Ramalingam, Satish Chandra, John Field, Dependent types for program understanding, in: Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, in: Lecture Notes in Computer Science, vol. 3440, Springer, 2005, pp. 157–173.

[28] Victor Lanvin, A Semantic Foundation for Gradual Set-Theoretic Types, Ph.D. Dissertation, Université de Paris, 2021.

[29] Jean-Jacques Lévy, Redexes are stable in the λ-calculus, Math. Struct. Comput. Sci. 27 (5) (2017) 738–750, https://doi.org/10.1017/S0960129515000353.

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

[31] Microsoft. TypeScript Microsoft, https://www.typescriptlang.org/.

[32] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, David Walker, Dynamic typing with dependent types, in: Jean-Jacques Levy, Ernst W. Mayr, John C. Mitchell (Eds.), Exploring New Frontiers of Theoretical Informatics, Springer US, Boston, MA, 2004, pp. 437–450.

[33] Mário Pereira, Sandra Alves, Mário Florido, Liquid intersection types, Electron. Proc. Theor. Comput. Sci. 177 (Mar 2015) 24–42, https://doi.org/10.4204/eptcs.177.3.

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

[35] Patrick Maxim Rondon, Ming Kawaguchi, Ranjit Jhala, Liquid types, in: Rajiv Gupta, Saman P. Amarasinghe (Eds.), Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June7-13, 2008, ACM, 2008, pp. 159–169.

[36] Amr Sabry, Matthias Felleisen, Reasoning about programs in continuation-passing style, in: Proceedings of the 1992 ACM Conference on LISP and Functional Programming, San Francisco, California, USA, LFP '92, Association for Computing Machinery, New York, NY, USA, 1992, pp. 288–298.

[37] Jeremy G. Siek, Walid Taha, Gradual typing for functional languages, in: Scheme and Functional Programming Workshop, vol. 6, 2006, pp. 81–92.

[38] Jeremy G. Siek, Sam Tobin-Hochstadt, The recursive union of some gradual types, in: A List of Successes That Can Change the World, Springer, 2016, pp. 388–410.

[39] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, John Tang Boyland, Refined criteria for gradual typing, in: 1st Summit on Advances in Programming Languages (SNAPL 2015), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[40] Masako Takahashi, Parallel reductions in λ-calculus, J. Symb. Comput. 7 (2) (1989) 113–123, https://doi.org/10.1016/S0747-7171(89)80045-8.

[41] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen, Is sound gradual typing dead?, in: Proceedings of the 43rd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (St Petersburg, FL, USA) POPL '16, ACM, , 2016, pp. 456–468.

[42] Sam Tobin-Hochstadt, Matthias Felleisen, The design and implementation of typed scheme, in: Proceedings of the 35th Annual ACM SIGPLAN Symposium on Principles of Programming Languages, San Francisco, California, USA, POPL '08, ACM, New York, NY, USA, 2008, pp. 395–406.

[43] Sam Tobin-Hochstadt, Matthias Felleisen, Logical types for untyped languages, in: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (Baltimore, Maryland, USA, ICFP '10, ACM, New York, NY, USA, 2010, pp. 117–128.

[44] Andrew K. Wright, Matthias Felleisen, A syntactic approach to type soundness, Inf. Comput. 115 (1) (1994) 38–94, https://doi.org/10.1006/inco.1994.1093.