# Semantic Subtyping for Non-Strict Languages

**Tommaso Petrucciani**
DIBRIS, Università di Genova, Italy
IRIF, Université Paris Diderot, France

**Giuseppe Castagna**
CNRS, IRIF, Université Paris Diderot, France

**Davide Ancona**
DIBRIS, Università di Genova, Italy

**Elena Zucca**
DIBRIS, Università di Genova, Italy

## Abstract

Semantic subtyping is an approach to define subtyping relations for type systems featuring union and intersection type connectives. It has been studied only for strict languages, and it is unsound for non-strict semantics. In this work, we study how to adapt this approach to non-strict languages: in particular, we define a type system using semantic subtyping for a functional language with a call-by-need semantics. We do so by introducing an explicit representation for divergence in the types, so that the type system distinguishes expressions that are results from those which are computations that might diverge.

**2012 ACM Subject Classification** Software and its engineering → Functional languages

**Keywords and phrases** Semantic subtyping, non-strict semantics, call-by-need, union types, intersection types

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2018.4

**Related Version** https://arxiv.org/abs/1810.05555

## 1 Introduction

Semantic subtyping is a powerful framework which allows language designers to define subtyping relations for rich languages of types – including union and intersection types – that can express precise properties of programs. However, it has been developed for languages with call-by-value semantics and, in its current form, it is unsound for non-strict languages. We show how to design a type system which keeps the advantages of semantic subtyping while being sound for non-strict languages (more specifically, for call-by-need semantics).

### 1.1 Semantic subtyping

Union and intersection types can be used to type several language constructs – from branching and pattern matching to function overloading – very precisely. However, they make it challenging to define a subtyping relation that behaves precisely and intuitively.

*Semantic subtyping* is a technique to do so, studied by Frisch, Castagna, and Benzaken [20] for types given by:

$$t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \qquad \text{where } b ::= \mathsf{Int} \mid \mathsf{Bool} \mid \cdots$$

Types include constructors – basic types $b$, arrows, and products – plus union $t \vee t$, intersection $t \wedge t$, negation (or complementation) $\neg t$, and the bottom and top types $\mathbb{0}$ and $\mathbb{1}$ (actually, $t_1 \wedge t_2$ and $\mathbb{1}$ can be defined respectively as $\neg(\neg t_1 \vee \neg t_2)$ and $\neg\mathbb{0}$). The grammar above is interpreted *coinductively* rather than inductively, thus allowing infinite type expressions


that correspond to recursive types. Subtyping is defined by giving an interpretation $[\![\cdot]\!]$ of types as sets and defining $t_1 \leq t_2$ as the inclusion of the interpretations, that is, $t_1 \leq t_2$ is defined as $[\![t_1]\!] \subseteq [\![t_2]\!]$. Intuitively, we can see $[\![t]\!]$ as the set of values that inhabit $t$ in the language. By interpreting union, intersection, and negation as the corresponding operations on sets and by giving appropriate interpretations to the other constructors, we ensure that subtyping will satisfy all commutative and distributive laws we expect: for example, $(t_1 \times t_2) \vee (t'_1 \times t'_2) \leq (t_1 \vee t'_1) \times (t_2 \vee t'_2)$ or $(t \to t_1) \wedge (t \to t_2) \leq t \to (t_1 \wedge t_2)$.

This relation is used in [20] to type a call-by-value language featuring higher-order functions, data constructors and destructors (pairs), and a typecase construct which models runtime type dispatch and acts as a form of pattern matching. Functions can be recursive and are explicitly typed; their type can be an intersection of arrow types, describing overloaded behaviour. A simple example of an overloaded function is

```
let f x = if (x is Int) then (x + 1) else not(x)
```

which tests whether its argument `x` is of type $\mathsf{Int}$ and in this case returns its successor, its negation otherwise. This function can be given the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$, which signifies that it has both type $\mathsf{Int} \to \mathsf{Int}$ and type $\mathsf{Bool} \to \mathsf{Bool}$: the two types define its two possible behaviours depending on the outcome of the test (and, thus, on the type of the input). This is done in [20] by explicitly annotating the whole function definition. Using notation for typecases from [20]: $\mathsf{let}\ f : (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) = \lambda x.\, (x \in \mathsf{Int})\ ?\ (x + 1) : (\mathsf{not}\ x)$. The type deduced for this function is $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$, but it can also be given the type $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$: the latter type states that the function can be applied to both integers and booleans and that its result is either an integer or a boolean. This latter type is less precise than the intersection, since it loses the correlation between the types of the argument and of the result. Accordingly, the semantic definition of subtyping ensures $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \leq (\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$.

The work of [20] has been extended to treat more language features, including parametric polymorphism [11, 12, 14], type inference [13], and gradual typing [10] and adapted to SMT solvers [6]. It has been used to type object-oriented languages [1, 16], XML queries [9], NoSQL languages [5], and scientific languages [27]. It is also at the basis of the definition of $\mathbb{C}$Duce, an XML-processing functional programming language with union and intersection types [4]. However, only strict evaluation had been considered, until now.

## 1.2 Semantic subtyping in lazy languages

Our work started as an attempt to design a type system for the Nix Expression Language [17], an untyped, purely functional, and lazily evaluated language for Unix/Linux package management. Since Nix is untyped, some programming idioms it encourages require advanced type system features to be analyzed properly. Notably, the possibility of writing functions that use type tests to have an overloaded-like behaviour made intersection types and semantic subtyping a good fit for the language. However, existing semantic subtyping relations are unsound for non-strict semantics; this was already observed in [20] and no adaptation has been proposed later. Here we describe our solution to define a type system based on semantic subtyping which is sound for a non-strict language. In particular, we consider a call-by-need variant of the language studied in [20].

Current semantic subtyping systems are unsound for non-strict semantics because of the way they deal with the bottom type $\mathbb{0}$, which corresponds to the empty set of values ($[\![\mathbb{0}]\!] = \varnothing$). The intuition is that a reducible expression $e$ can be safely given a type $t$ only if all results (i.e., values) it can return are of type $t$. Accordingly, $\mathbb{0}$ can only be assigned

to expressions that are statically known to diverge (i.e., that never return a result). For example, the ML expression $\mathsf{let\ rec\ f\ x} = \mathsf{f\ x\ in\ f\ ()}$ can be given type $\mathbb{0}$. Let us use $\bar{e}$ to denote any diverging expression that, like this, can be given type $\mathbb{0}$. Consider the following typing derivations, which are valid in current semantic subtyping systems ($\pi_2$ projects the second component of a pair).

$$\dfrac{\dfrac{\vdash (\bar{e}, 3) : \mathbb{0} \times \mathsf{Int}}{\vdash (\bar{e}, 3) : \mathbb{0} \times \mathsf{Bool}}\,[\simeq]}{\vdash \pi_2\,(\bar{e}, 3) : \mathsf{Bool}} \qquad \dfrac{\dfrac{\vdash \lambda x.\,3 : \mathbb{0} \to \mathsf{Int}}{\vdash \lambda x.\,3 : \mathbb{0} \to \mathsf{Bool}}\,[\simeq] \qquad \vdash \bar{e} : \mathbb{0}}{\vdash (\lambda x.\,3)\,\bar{e} : \mathsf{Bool}}$$

Note that both $\pi_2\,(\bar{e}, 3)$ and $(\lambda x.\,3)\,\bar{e}$ diverge in call-by-value semantics (since $\bar{e}$ must be evaluated first), while they both reduce to 3 in call-by-name or call-by-need. The derivations are therefore sound for call-by-value, while they are clearly unsound with non-strict evaluation.

Why are these derivations valid? The crucial steps are those marked with $[\simeq]$, which convert between types that have the same interpretation; $\simeq$ denotes this equivalence relation. With semantic subtyping, $\mathbb{0} \times \mathsf{Int} \simeq \mathbb{0} \times \mathsf{Bool}$ holds because all types of the form $\mathbb{0} \times t$ are equivalent to $\mathbb{0}$ itself: none of these types contains any value (indeed, product types are interpreted as Cartesian products and therefore the product with the empty set is itself empty). It can appear more surprising that $\mathbb{0} \to \mathsf{Int} \simeq \mathbb{0} \to \mathsf{Bool}$ holds. We interpret a type $t_1 \to t_2$ as the set of functions which, on arguments of type $t_1$, either diverge or return results in type $t_2$. Since there is no argument of type $\mathbb{0}$ (because, in call-by-value, arguments are always values), all types of the form $\mathbb{0} \to t$ are equivalent (they all contain every well-typed function).

## 1.3 Our approach

The intuition behind our solution is that, with non-strict semantics, it is not appropriate to see a type as the set of the values that have that type. In a call-by-value language, operations like application or projection occur on values: thus, we can identify two types (and, in some sense, the expressions they type) if they contain (and their expressions may produce) the same values. In non-strict languages, though, operations also occur on partially evaluated results: these, like $(\bar{e}, 3)$ in our example, can contain diverging sub-expressions below their top-level constructor.

As a result, it is unsound, for example, to type $(\bar{e}, 3)$ as $\mathbb{0} \times \mathsf{Int}$, since we have that $\mathbb{0} \times \mathsf{Int}$ and $\mathbb{0} \times \mathsf{Bool}$ are equivalent. It is also unsound to have subtyping rules for functions which assume implicitly that every argument will eventually be a value.

One approach to solve this problem would be to change the interpretation of $\mathbb{0}$ so that it is non-empty. However, the existence of types with an empty interpretation is important for the internal machinery of semantic subtyping. Notably, the decision procedure for subtyping relies on them (checking whether $t_1 \leq t_2$ holds is reduced to checking whether the type $t_1 \wedge \neg t_2$ is empty). Therefore, we keep the interpretation $[\![\mathbb{0}]\!] = \varnothing$, but we change the type system so that this type is *never* derivable, not even for diverging expressions. We keep it as a purely "internal" type useful to describe subtyping, but never used to type expressions.

We introduce instead a separate type $\bot$ as the type of diverging expressions. This type is non-empty but disjoint from the types of constants, functions, and pairs: $[\![\bot]\!]$ is a singleton whose unique element represents divergence. Introducing the type $\bot$ means that we track termination in types. In particular, we distinguish two classes of types: those that are disjoint from $\bot$ (for example, $\mathsf{Int}$, $\mathsf{Int} \to \mathsf{Bool}$, or $\mathsf{Int} \times \mathsf{Bool}$) and those that include $\bot$ (since the interpretation of $\bot$ is a singleton, no type can contain a proper subset of it). Intuitively, the

former correspond to computations that are guaranteed to terminate: for example, $\mathsf{Int}$ is the type of terminating expressions producing an integer result. Conversely, the types of diverging expressions must always contain $\bot$ and, as a result, they can always be written in the form $t \vee \bot$, for some type $t$. Subtyping verifies $t \leq t \vee \bot$ for any $t$: this ensures that a terminating expression can always be used when a possibly diverging one is expected. This subdivision of types suggests that $\bot$ is used to approximate the set of diverging well-typed expressions: an expression whose type contains $\bot$ is an expression that *may* diverge. Actually, the type system we propose performs a rather gross approximation. We derive "terminating types" (i.e., subtypes of $\neg\bot$) only for expressions that are already results and cannot be reduced: constants, functions, or pairs. Applications and projections, instead, are always typed by assuming that they might diverge. The typing rules are written to handle and propagate the $\bot$ type. For example, we type applications using the following rule.

$$\frac{\Gamma \vdash e_1 : (t' \to t) \vee \bot \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t \vee \bot}$$

This rule allows the expression $e_1$ to be possibly diverging: we require it to have the type $(t' \to t) \vee \bot$ instead of the usual $t' \to t$ (but an expression with the latter type can always be subsumed to have the former type). We type the whole application as $t \vee \bot$ to signify that it can diverge even if the codomain $t$ does not include $\bot$, since $e_1$ can diverge.

This system avoids the problems we have seen with semantic subtyping: no expression can be assigned the empty type, which was the type on which subtyping had incorrect behaviour. The new type $\bot$ does not cause the same problems because $[\![\bot]\!]$ is non-empty. For example, the type of expressions like $(\bar{e}, 3)$ – where $\bar{e}$ is diverging – is now $\bot \times \mathsf{Int}$. This type is not equivalent to $\bot \times \mathsf{Bool}$: indeed, the two interpretations are different because the interpretation of types includes an element ($[\![\bot]\!]$) to represent divergence.

Typing all applications as possibly diverging – even very simple ones like $(\lambda x.\, 3)\, e$ – is a very coarse approximation which can seem unsatisfactory. We could try to amend the rule to say that if $e_1$ has type $t' \to t$, then $e_1\, e_2$ has type $t$ instead of $t \vee \bot$. However, we prefer to keep the simpler rules since they achieve our goal of giving a sound type system that still enjoys most benefits of semantic subtyping.

An advantage of the simpler system is that it allows us to treat $\bot$ as an internal type that does not need to be written explicitly by programmers. Since the language is explicitly typed, if $\bot$ were to be treated more precisely, programmers would presumably need to include it or exclude it explicitly from function signatures. This would make the type system significantly different from conventional ones where divergence is not explicitly expressed in the types. In the present system, instead, we can assume that programmers annotate programs using standard set-theoretic types and $\bot$ is introduced only behind the scenes and, thus, is transparent to programmers.

We define this type system for a call-by-need variant of the language studied in [20], and we prove its soundness in terms of progress and subject reduction.

The choice of call-by-need rather than call-by-name stems from the behaviour of semantic subtyping on intersections of arrow types. Our type system would actually be unsound for call-by-name if the language were extended with constructs that can reduce non-deterministically to different answers. For example, the expression $\mathsf{rnd}(t)$ of [20] that returns a random value of type $t$ could not be added while keeping soundness. This is because in call-by-name, if such an expression is duplicated, each occurrence could reduce differently; in call-by-need, instead, its evaluation would be shared. Intersection and union types make the type system precise enough to expose this difference. In the absence of such non-deterministic

constructs, call-by-name and call-by-need can be shown to be observationally equivalent, so that soundness should hold for both; however, call-by-need also simplifies the technical work to prove soundness.

We show an example of this, though we will return on this point later. Consider the following derivation, where $\bar{e}$ is an expression of type $\mathsf{Int} \vee \mathsf{Bool}$.

$$\dfrac{\dfrac{\dfrac{x\colon \mathsf{Int} \vdash (x,x)\colon \mathsf{Int}\times\mathsf{Int} \qquad x\colon \mathsf{Bool} \vdash (x,x)\colon \mathsf{Bool}\times\mathsf{Bool}}{\vdash \lambda x.\,(x,x)\colon (\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})}}{\vdash \lambda x.\,(x,x)\colon \mathsf{Int}\vee\mathsf{Bool}\to(\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})}\,[\leq] \qquad \vdash \bar{e}\colon \mathsf{Int}\vee\mathsf{Bool}}{\vdash (\lambda x.\,(x,x))\,\bar{e}\colon (\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})}$$

In a system with intersection types, the function $\lambda x.\,(x,x)$ can be given the type $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$ because it has both arrow types (in practice, the function will have to be annotated with the intersection). Then, the step marked with $[\leq]$ is allowed because, in semantic subtyping, $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$ is a subtype of $(\mathsf{Int}\vee\mathsf{Bool})\to((\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool}))$ (in general, $(t_1\to t_1')\wedge(t_2\to t_2')\leq t_1\vee t_2\to t_1'\vee t_2'$). Therefore, the application $(\lambda x.\,(x,x))\,\bar{e}$ is well-typed with type $(\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})$. In call-by-name, it reduces to $(\bar{e},\bar{e})$: therefore, for the system to satisfy subject reduction, we must be able to type $(\bar{e},\bar{e})$ with the type $(\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})$ too. But this type is intuitively unsound for $(\bar{e},\bar{e})$ if each occurrence of $\bar{e}$ could reduce independently and non-deterministically either to an integer or to a boolean. Using a typecase we can actually exhibit a term that breaks subject reduction.

There are several ways to approach this problem. We could change the type system or the subtyping relation so that $\lambda x.\,(x,x)$ cannot be given the type $(\mathsf{Int}\vee\mathsf{Bool})\to((\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool}))$. However, this would curtail the expressive power of intersection types as used in the semantic subtyping approach. We could instead assume explicitly that the semantics is deterministic. In this case, the typing would not be unsound intuitively, but a proof of subject reduction would be difficult: we should give a complex union disjunction rule to type $(\bar{e},\bar{e})$. We choose instead to consider a call-by-need semantics because it solves both problems. With call-by-need, non-determinism poses no difficulty because of sharing. We still need a union disjunction rule, but it is simpler to state since we only need it to type the $\mathsf{let}$ bindings which represent shared computations.

## 1.4 Contributions

The main contribution of this work is the development of a type system for non-strict languages based on semantic subtyping; to our knowledge, this had not been studied before.

Although the idea of our solution is simple – to track divergence – its technical development is far from trivial. Our work highlights how a type system featuring union and intersection types is sensitive to the difference between strict and non-strict semantics and also, in the presence of non-determinism, to that between call-by-name and call-by-need. This shows once more how union and intersection types can express very fine properties of programs. Our main technical contribution is the description of sound typing for $\mathsf{let}$ bindings – a construct peculiar to most of the formalizations of call-by-need semantics – in the presence of union types. Finally, our work shows how to integrate the $\bot$ type, which is an explicit representation for divergence, in a semantic subtyping system. It can thus also be seen as a first step towards the definition of a type system based on semantic subtyping that performs a non-trivial form of termination analysis.

## 1.5 Related work

Previous work on semantic subtyping does not discuss non-strict semantics. Castagna and Frisch [8] describe how to add a type constructor $\mathsf{lazy}(t)$ to semantic subtyping systems, but this is meant just to have lazily constructed expressions within a call-by-value language.

Many type systems for functional languages – like the simply-typed $\lambda$-calculus or Hindley-Milner typing – are sound for both strict and non-strict semantics. However, difficulties similar to ours are found in work on refinement types. Vazou et al. [23] study how to adapt refinement types for Haskell. Their types contain logical predicates as refinements: e.g., the type of positive integers is $\{\, v\colon \mathsf{Int} \mid v > 0 \,\}$. They observe that the standard approach to typechecking in these systems – checking implication between predicates with an SMT solver – is unsound for non-strict semantics. In their system, a type like $\{\, v\colon \mathsf{Int} \mid \mathsf{false} \,\}$ is analogous to $\mathbb{0}$ in our system insofar as it is not inhabited by any value. These types can be given to diverging expressions, and their introduction into the environment causes unsoundness. To avoid this problem, they stratify types, with types divided in diverging and non-diverging ones. This corresponds in a way to our use of a type $\bot$ in types of possibly diverging expressions. As for ours, their type system can track termination to a certain extent. Partial correctness properties can be verified even without precise termination analysis. However, with their kind of analysis (which goes beyond what is expressible with set-theoretic types) there is a significant practical benefit to tracking termination more precisely. Hence, they also study how to check termination of recursive functions.

The notion of a stratification of types to keep track of divergence can also be found in work of a more theoretical strain. For instance, in [15] it is used to model partial functions in constructive type theory. This stratification can be understood as a monad for partiality, as it is treated in [7]. Our type system can also be seen, intuitively, as following this monadic structure. Notably, the rule for applications in a sense lifts the usual rule for application in this partiality monad. Injection in this monad is performed implicitly by subtyping via the judgment $t \leq t \vee \bot$. However, we have not developed this intuition formally.

The fact that a type system with union and intersection types can require changes to account for non-strict semantics is also remarked in work on refinement types. Dunfield and Pfenning [19, p. 8, footnote 3] notice how a union elimination rule cannot be used to eliminate unions in function arguments if arguments are passed by name: this is analogous to the aforementioned difficulties which led to our choice of call-by-need (their system uses a dedicated typing rule for what our system handles by subtyping). Dunfield [18, Section 8.1.5] proposes as future work to adapt a subset of the type system he considers (of refinement types for a call-by-value effectful language) to call-by-name. He notes some of the difficulties and advocates studying call-by-need as a possible way to face them. In our work we show, indeed, that a call-by-need semantics can be used to have the type system handle union and intersection types expressively without requiring complex rules.

Finally, Vouillon [24] – drawing on earlier work with Melliès [25] on interpreting types as sets of terms – studies the subtyping relation induced by such an interpretation for systems with union types. Many concerns raised in his work parallel ours. He remarks that some subtyping rules are only sound for specific calculi (e.g., only for call-by-value or only for deterministic semantics), while others are sound for large classes of calculi. He defines subtyping avoiding the rules of the first kind to have a relation which is more robust to language extensions or modifications than semantic subtyping as we use it (though, in doing so, he does not capture fully the set-theoretic intuition for strict languages). He also remarks how union elimination is problematic for non-deterministic call-by-name semantics. His interpretation of types as sets of terms is more adapted to describing non-strict semantics than

the semantic-subtyping approach of interpreting types as sets of values. However, his system does not account for negation types, that we include and interpret as set complementation: this would probably be challenging to integrate into his theory.

## 1.6 Outline

Our presentation proceeds as follows. In Section 2, we define the types and the subtyping relation which we use in our type system. In Section 3, we define the language we study, its syntax, and its operational semantics. In Section 4, we present the type system; we state the result of soundness for it and outline the main lemmas required to prove it; we also complete the discussion about why we chose a call-by-need semantics. In Section 5, we study the relation between the interpretation of types used to define subtyping and the expressions that are definable in the language; we show how we can look for a more precise interpretation. In Section 6 we conclude and point out more directions for future work.

For space reasons, some auxiliary definitions and results, as well as the proofs of the results we state, are omitted and can all be found in the extended version available online [22].

# 2 Types and subtyping

We begin by describing in more detail the types and the subtyping relation of our system.

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

▶ **Definition 2.1** (Types). *The set $\mathcal{T}$ of* types *is the set of terms $t$ coinductively produced by the following grammar*

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

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

We introduce the abbreviations $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2)$, and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$. We refer to $b$, $\times$, and $\to$ as *type constructors*, and to $\vee$, $\neg$, $\wedge$, and $\setminus$ as *type connectives*.

The regularity condition is necessary only to ensure the decidability of the subtyping relation. Contractivity, instead, is crucial because it excludes terms which do not have a meaningful interpretation as types or sets of values: for instance, the trees satisfying the equations $t = t \vee t$ (which gives no information on which values are in it) or $t = \neg t$ (which cannot represent any set of values). Contractivity also ensures that the binary relation $\triangleright \subseteq \mathcal{T}^2$ defined by $t_1 \vee t_2 \triangleright t_i$ and $\neg t \triangleright t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}$ that we will use without further reference to the relation (e.g., in Definition 2.3). This induction principle allows us to apply the induction hypothesis below type connectives (union and negation), but not below type constructors (product and arrow). As a consequence of contractivity, types cannot contain infinite unions or intersections.

In the semantic subtyping approach we give an interpretation of types as sets; this interpretation is used to define the subtyping relation in terms of set containment. We want to see a type as the set of the values of the language that have that type. However, this set of values cannot be used directly to define the interpretation, because of a problem of circularity. Indeed, in a higher-order language, values include well-typed $\lambda$-abstractions; hence to know which values inhabit a type we need to have already defined the type system (to type $\lambda$-abstractions), which depends on the subtyping relation, which in turn depends on the interpretation of types. To break this circularity, types are actually interpreted as subsets of a set $\mathcal{D}$, an *interpretation domain*, which is not the set of values, though it corresponds to it intuitively (in [20], a correspondence is also shown formally: we return to this in Section 5). We use the following domain which includes an explicit representation for divergence.

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

$$d ::= \bot \mid c \mid (d, d) \mid \{(d, d_\Omega), \dots, (d, d_\Omega)\} \qquad\qquad d_\Omega ::= d \mid \Omega$$

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

The elements of $\mathcal{D}$ correspond, intuitively, to the results of the evaluation of expressions. The element $\bot$ stands for divergence. Expressions can produce as results constants or pairs of results, so we include both in $\mathcal{D}$. For example, a result can be a pair of a terminating computation returning $\mathsf{true}$ and a diverging computation: we represent this by $(\mathsf{true}, \bot)$. Finally, in a higher-order language, the result of a computation can be a function. Functions are represented in this model by finite relations of the form $\{(d^1, d^1_\Omega), \dots, (d^n, d^n_\Omega)\}$, where $\Omega$ (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 constant $\Omega$ is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used $d$ instead of $d_\Omega$, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, indenpendently from the type of its argument (see Section 4.2 of [20] for details). The restriction to *finite* relations is standard in semantic subtyping [20]; we say more about it in Section 5.

We define the interpretation $[\![t]\!]$ of a type $t$ so that it satisfies the following equalities, where $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ and where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets:

$$[\![\bot]\!] = \{\bot\} \qquad [\![b]\!] = \mathbb{B}(b) \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \left\{ R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \;\middle|\; \forall (d, d') \in R.\, d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \right\}$$
$$[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![\mathbb{0}]\!] = \varnothing$$

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

▶ **Definition 2.3** (Set-theoretic interpretation of types). *We define a binary predicate* $(d_\Omega : t)$ *("the element* $d_\Omega$ *belongs to the type* $t$*"), where* $d_\Omega \in \mathcal{D} \cup \{\Omega\}$ *and* $t \in \mathcal{T}$*, by induction on the*

*pair* $(d_\Omega, t)$ *ordered lexicographically. The predicate is defined as follows:*

$$\begin{aligned}
(\bot : \bot) &= \mathsf{true} \\
(c : b) &= c \in \mathbb{B}(b) \\
((d_1, d_2) : t_1 \times t_2) &= (d_1 : t_1) \text{ and } (d_2 : t_2) \\
(\{(d^1, d^1_\Omega), \dots, (d^n, d^n_\Omega)\} : t_1 \to t_2) &= \forall i \in \{1, \dots, n\}.\ \mathsf{if}\ (d^i : t_1)\ \mathsf{then}\ (d^i_\Omega : t_2) \\
(d : t_1 \vee t_2) &= (d : t_1) \text{ or } (d : t_2) \\
(d : \neg t) &= \mathsf{not}\ (d : t) \\
(d_\Omega : t) &= \mathsf{false} \qquad \textit{otherwise}
\end{aligned}$$

*We define the* set-theoretic interpretation $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ *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)**.** *We define the* subtyping *relation* $\leq$ *and the* subtyping equivalence *relation* $\simeq$ *as* $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ *and* $t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leq t_2) \text{ and } (t_2 \leq t_1)$ .

## 3 Language syntax and semantics

We consider a language based on that studied in [20]: a $\lambda$-calculus with recursive explicitly annotated functions, pair constructors and destructors, and a typecase construct. This is the *source language* in which programs are written. We define the semantics on a slightly different *internal language* and show how to compile source programs to this internal language. The main reason for introducing the internal language is that, to describe call-by-need semantics in a small-step operational style, we need to add to the source language a let construct, a form of explicit substitution which models sharing of computations (following a standard approach [2,3,21]). The internal language is not an extension of the source language, however, because we also restrict the allowed syntax of typecases to simplify the semantics.

First, we give some auxiliary definitions on types. We introduce the abbreviations: $\langle t \rangle \stackrel{\text{def}}{=} t \vee \bot$; $t_1 \twoheadrightarrow t_2 \stackrel{\text{def}}{=} \langle t_1 \rangle \to \langle t_2 \rangle$; and $t_1 \otimes t_2 \stackrel{\text{def}}{=} \langle t_1 \rangle \times \langle t_2 \rangle$ . These are compact notations for types including $\bot$. The first, $\langle t \rangle$, is an abbreviated way to write the type of possibly diverging expressions whose result has type $t$. The latter two are used in type annotations. The intent is that programmers never write $\bot$ explicitly. Rather, they use the $\twoheadrightarrow$ and $\otimes$ constructors instead of $\to$ and $\times$ so that $\bot$ is introduced implicitly. The $\to$ and $\times$ constructors are never written directly in program. We define the following restricted grammars of types

$$T ::= b \mid T \otimes T \mid T \twoheadrightarrow T \mid T \vee T \mid \neg T \mid \mathbb{0} \qquad \tau ::= b \mid \tau \otimes \tau \mid \mathbb{0} \to \mathbb{1} \mid \tau \vee \tau \mid \neg \tau \mid \mathbb{0}$$

both of which are interpreted coinductively, with the same restrictions of regularity and contractivity as in the definition of types. The types defined by these grammars are the only ones which appear in programs: neither includes $\bot$ explicitly.

In particular, functions are annotated with $T$ types, where the $\otimes$ and $\twoheadrightarrow$ forms are used to ensure that every type below a constructor is of the form $t \vee \bot$.

Typecases, instead, check $\tau$ types. The only arrow type that can appear in them is $\mathbb{0} \to \mathbb{1}$, which is the top type of functions (every well-typed function has this type). This restriction means that typecases will not be able to test the types of functions, but only, at most, whether a value is a function or not. This restriction is not imposed in [20], and actually it could be lifted here without difficulty. We include it because the purpose of typecases in our language is, to some extent, the modelling of pattern matching, which cannot test the type of functions. Restricting typecases on arrow types also facilitates the extension of the system with polymorphism and type inference.

## 3.1 Source language

The *source language expressions* are the terms $\mathbf{e}$ produced inductively by the grammar

$$\begin{aligned}&\mathbf{e} ::= x \mid c \mid \mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid (\mathbf{e},\mathbf{e}) \mid \pi_i\,\mathbf{e} \mid (x = \mathbf{e}) \in \tau\ ?\ \mathbf{e} : \mathbf{e}\\ &\mathcal{I} ::= \textstyle\bigwedge_{i\in I} T'_i \to T_i \qquad\qquad |I| > 0\end{aligned}$$

where $f$ and $x$ range over a set $\mathcal{X}$ of *expression variables*, $c$ over the set $\mathcal{C}$ of constants, $i$ in $\pi_i\,\mathbf{e}$ over $\{1,2\}$, and where $\tau$ in $(x = \mathbf{e}) \in \tau\ ?\ \mathbf{e} : \mathbf{e}$ is such that $\tau \not\simeq \mathbb{0}$ and $\tau \not\simeq \mathbb{1}$.

Source language expressions include variables, constants, $\lambda$-abstractions, applications, pairs constructors $(\mathbf{e},\mathbf{e})$ and destructors $\pi_1\,\mathbf{e}$ and $\pi_2\,\mathbf{e}$, plus the typecase $(x = \mathbf{e}) \in \tau\ ?\ \mathbf{e} : \mathbf{e}$.

A $\lambda$-abstraction $\mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$ is a possibly recursive function, with recursion parameter $f$ and argument $x$, both of which are bound in the body; the function is explicitly annotated with its type $\mathcal{I}$, which is a finite intersection of types of the form $T' \to T$.

A typecase expression $(x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2$ has the following intended semantics: $\mathbf{e}_0$ is evaluated until it can be determined whether it has type $\tau$ or not, then the selected branch ($\mathbf{e}_1$ if the result of $\mathbf{e}_0$ has type $\tau$, $\mathbf{e}_2$ if it has type $\neg\tau$: one of the two cases always occurs) is evaluated in an environment where $x$ is bound to the result of $\mathbf{e}_0$. Actually, to simplify the presentation, we will give a non-deterministic semantics in which we allow to evaluate $\mathbf{e}_0$ more than what is needed to ascertain whether it has type $\tau$.

In the syntax definition above we have restricted the types $\tau$ in typecases asking both $\tau \not\simeq \mathbb{1}$ and $\tau \not\simeq \mathbb{0}$. A typecase checking the type $\mathbb{1}$ is useless: since all expressions have type $\mathbb{1}$, it immediately reduces to its first branch. Likewise, a typecase checking the type $\mathbb{0}$ reduces directly to the second branch. Therefore, the two cases are uninteresting to consider. We forbid them because this allows us to give a simpler typing rule for typecases. Allowing them is just a matter of adding two (trivial) typing rules specific to these cases, as we show later.

As customary, we consider expressions up to renaming of bound variables. In $\mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$, $f$ and $x$ are bound in $\mathbf{e}$. In $(x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2$, $x$ is bound in $\mathbf{e}_1$ and $\mathbf{e}_2$.

We do not provide mechanisms to define cyclic data structures. For example, we do not have a direct syntactic construct to define the infinitely nested pair $(1,(1,\dots))$. We can define it by writing a fixpoint operator (which can be typed in our system since types can be recursive) or by defining and applying a recursive function which constructs the pair. A general letrec construct as in [2] might be useful in practice (for efficiency or to provide greater sharing) but we omit it here since we are only concerned with typing.

## 3.2 Internal language

The *internal language expressions* are the terms $e$ produced inductively by the grammar

$$\begin{aligned}&e ::= x \mid c \mid \mu f\colon \mathcal{I}.\, \lambda x.\, e \mid e\,e \mid (e,e) \mid \pi_i\,e \mid (x = \varepsilon) \in \tau\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\\ &\varepsilon ::= x \mid c \mid \mu f\colon \mathcal{I}.\, \lambda x.\, e \mid (\varepsilon,\varepsilon)\end{aligned}$$

where metavariables and conventions are as in the source language. There are two differences with respect to the source language. One is the introduction of the construct $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, which is a binder used to model sharing of computations in call-by-need semantics (in $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, $x$ is bound in $e_2$). The other difference is that typecases cannot check arbitrary expressions, but only expressions of the restricted form given by $\varepsilon$. This restriction simplifies the semantics of typecases.

A source language expression $\mathbf{e}$ can be compiled to an internal language expression $\lceil \mathbf{e} \rceil$ as follows. Compilation is straightforward for all expressions apart from typecases:

$$\lceil x \rceil = x \qquad \lceil c \rceil = c \qquad \lceil \mu f : \mathcal{I}.\, \lambda x.\, \mathbf{e} \rceil = \mu f : \mathcal{I}.\, \lambda x.\, \lceil \mathbf{e} \rceil$$
$$\lceil \mathbf{e}_1\, \mathbf{e}_2 \rceil = \lceil \mathbf{e}_1 \rceil\, \lceil \mathbf{e}_2 \rceil \qquad \lceil (\mathbf{e}_1, \mathbf{e}_2) \rceil = (\lceil \mathbf{e}_1 \rceil, \lceil \mathbf{e}_2 \rceil) \qquad \lceil \pi_i\, \mathbf{e} \rceil = \pi_i\, \lceil \mathbf{e} \rceil$$

and for typecases it introduces a $\mathsf{let}$ binder to ensure that the checked expression is a variable:

$$\lceil (x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2 \rceil = \mathsf{let}\ y = \lceil \mathbf{e}_0 \rceil\ \mathsf{in}\ (x = y) \in \tau\ ?\ \lceil \mathbf{e}_1 \rceil : \lceil \mathbf{e}_2 \rceil$$

where $y$ is chosen not free in $\mathbf{e}_1$ and $\mathbf{e}_2$. (The other forms for $\varepsilon$ appear during reduction.)

## 3.3 Semantics

We define the operational semantics of the internal language as a small-step reduction relation using call-by-need. The semantics of the source language is then given indirectly through the translation. The choice of call-by-need rather than call-by-name was briefly motivated in the Introduction and will be discussed more extensively in Section 4.

We first define the sets of *answers* (ranged over by $a$) and of *values* (ranged over by $v$) as the subsets of expressions produced by the following grammars:

$$a ::= c \mid \mu f : \mathcal{I}.\, \lambda x.\, e \mid (e, e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ a \qquad\qquad v ::= c \mid \mu f : \mathcal{I}.\, \lambda x.\, e$$

Answers are the results of evaluation. They correspond to expressions which are fully evaluated up to their top-level constructor (constant, function, or pair) but which may include arbitrary expressions below that constructor (so we have $(e, e)$ rather than $(a, a)$). Since they also include $\mathsf{let}$ bindings, they represent closures in which variables can be bound to arbitrary expressions. Values are a subset of answers treated specially in a reduction rule.

The semantics uses evaluation contexts to direct the order of evaluation. A *context* $C$ is an expression with a hole (written $[\,]$) in it. We write $C[e]$ for the expression obtained by replacing the hole in $C$ with $e$. We write $C\lceil e \rceil$ for $C[e]$ when the free variables of $e$ are not bound by $C$: for example, $\mathsf{let}\ x = e_1\ \mathsf{in}\ x$ is of the form $C[x]$ – with $C \equiv (\mathsf{let}\ x = e_1\ \mathsf{in}\ [\,])$ – but not of the form $C\lceil x \rceil$; conversely, $\mathsf{let}\ x = e_1\ \mathsf{in}\ y$ is both of the form $C[y]$ and $C\lceil y \rceil$.

*Evaluation contexts* $E$ are the subset of contexts generated by the following grammar:

$$E ::= [\,] \mid E\, e \mid \pi_i\, E \mid (x = F) \in \tau\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E\lceil x \rceil$$
$$F ::= [\,] \mid (F, \varepsilon) \mid (\varepsilon, F)$$

Evaluation contexts allow reduction to occur on the left of applications and below projections, but not on the right of applications and below pairs. For typecases alone, the contexts allow reduction also below pairs, since this reduction might be necessary to be able to determine whether the expression has type $\tau$ or not. This is analogous to the behaviour of pattern matching in lazy languages, which can force evaluation below constructors. The contexts for $\mathsf{let}$ are from standard presentations of call-by-need [2, 21]. They allow reduction of the body of the $\mathsf{let}$, while they only allow reductions of the bound expression when it is required to continue evaluating the body: this is enforced by requiring the body to have the form $E\lceil x \rceil$.

Figure 1 presents the reduction rules. They rely on the $\mathsf{typeof}$ function, which assigns types to expressions of the form $\varepsilon$. It is defined as follows:

$$\mathsf{typeof}(x) = \mathbb{1} \qquad \mathsf{typeof}(\mu f : \mathcal{I}.\, \lambda x.\, e) = \mathbb{0} \to \mathbb{1}$$
$$\mathsf{typeof}(c) = b_c \qquad \mathsf{typeof}((\varepsilon_1, \varepsilon_2)) = \mathsf{typeof}(\varepsilon_1) \times \mathsf{typeof}(\varepsilon_2)$$

$$
\begin{array}{rrcll}
[\textsc{Appl}] & (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\; e' & \rightsquigarrow & \mathsf{let}\ f = (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\ \mathsf{in}\ \mathsf{let}\ x = e'\ \mathsf{in}\ e & \\
[\textsc{ApplL}] & (\mathsf{let}\ x = e\ \mathsf{in}\ a)\; e' & \rightsquigarrow & \mathsf{let}\ x = e\ \mathsf{in}\ a\; e' & \\
[\textsc{Proj}] & \pi_i\ (e_1, e_2) & \rightsquigarrow & e_i & \\
[\textsc{ProjL}] & \pi_i\ (\mathsf{let}\ x = e\ \mathsf{in}\ a) & \rightsquigarrow & \mathsf{let}\ x = e\ \mathsf{in}\ \pi_i\ a & \\
[\textsc{LetV}] & \mathsf{let}\ x = v\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & (E[\![x]\!])[v/x] & \\
[\textsc{LetP}] & \mathsf{let}\ x = (e_1, e_2)\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ (E[\![x]\!])[(x_1,x_2)/x] & \\
[\textsc{LetL}] & \mathsf{let}\ x = (\mathsf{let}\ y = e\ \mathsf{in}\ a)\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & \mathsf{let}\ y = e\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!] & \\
[\textsc{Case1}] & (x = \varepsilon) \in \tau\ ?\ e_1 : e_2 & \rightsquigarrow & \mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_1 & \text{if } \mathsf{typeof}(\varepsilon) \leq \tau \\
[\textsc{Case2}] & (x = \varepsilon) \in \tau\ ?\ e_1 : e_2 & \rightsquigarrow & \mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_2 & \text{if } \mathsf{typeof}(\varepsilon) \leq \neg\tau \\
[\textsc{Ctx}] & E[e] & \rightsquigarrow & E[e'] & \text{if } e \rightsquigarrow e'
\end{array}
$$

**Figure 1** Operational semantics.

[Appl] is the standard application rule for call-by-need: the application $(\mu f\colon \mathcal{I}.\, \lambda x.\, e)\; e'$ reduces to $e$ prefixed by two let bindings that bind the recursion variable $f$ to the function itself and the parameter $x$ to the argument $e'$. [ApplL] instead deals with applications with a let expression in function position: it moves the application below the let. The rule is necessary to prevent loss of sharing: substituting the binding of $x$ to $e$ in $a$ would duplicate $e$. Symmetrically, there are two rules for pair projections, [Proj] and [ProjL].

There are three rules for let expressions. They rewrite expressions of the form $\mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]$: that is, let bindings where the bound expression is an answer and the body is an expression whose evaluation requires the evaluation of $x$. If $a$ is a value $v$, [LetV] applies and the expression is reduced by just replacing $v$ for $x$ in the body. If $a$ is a pair, [LetP] applies: the occurrences of $x$ in the body are replaced with a pair of variables $(x_1, x_2)$ and each $x_i$ is bound to $e_i$ by new let bindings (replacing $x$ directly by $(e_1, e_2)$ would duplicate expressions). Finally, the [LetL] rule just moves a let binding out of another.

There are two rules for typecases, by which a typecase construct $(x = \varepsilon) \in \tau\ ?\ e_1 : e_2$ can be reduced to either branch, introducing a new binding of $x$ to $\varepsilon$. The rules apply only if either of $\mathsf{typeof}(\varepsilon) \leq \tau$ or $\mathsf{typeof}(\varepsilon) \leq \neg\tau$ holds. If neither holds, then the two rules do not apply, but the [Ctx] rule can be used to continue the evaluation of $\varepsilon$.

**Comparison to other presentations of call-by-need.** These reduction rules mirror those from standard presentations of call-by-need [2, 3, 21]. A difference is that, in [LetV] or [LetP], we replace *all* occurrences of $x$ in $E[\![x]\!]$ at once, whereas in the cited presentations only the occurrence in the hole is replaced: for example, in [LetV] they reduce to $E[\![v]\!]$ instead of $(E[\![x]\!])[v/x]$. Our [LetV] rule is mentioned as a variant in [21, p. 38]. We use it because it simplifies the proof of subject reduction while maintaining an equivalent semantics.

**Non-determinism in the rules.** The semantics is not deterministic. There are two sources of non-determinism, both related to typecases. One is that the contexts $F$ include both $(F, \varepsilon)$ and $(\varepsilon, F)$ and thereby impose no constraint on the order with which pairs are examined.

The second source of non-determinism is that the contexts for typecases allow us to reduce the bindings of variables in the checked expression even when we can already apply [Case1] or [Case2]. For example, take $\mathsf{let}\ x = e\ \mathsf{in}\ (y = (3, x)) \in (\mathsf{Int} \otimes \mathbb{1})\ ?\ e_1 : e_2$.

$$[\text{S-Subsum}]\ \frac{\Gamma \vdash \mathbf{e} \colon t' \qquad t' \leq t}{\Gamma \vdash \mathbf{e} \colon t} \qquad [\text{S-Var}]\ \frac{}{\Gamma \vdash x \colon t}\ \Gamma(x) = t \qquad [\text{S-Const}]\ \frac{}{\Gamma \vdash c \colon b_c}$$

$$[\text{S-Abstr}]\ \frac{\forall i \in I. \quad \Gamma, f \colon \mathcal{I}, x \colon \langle T_i' \rangle \vdash \mathbf{e} \colon \langle T_i \rangle}{\Gamma \vdash (\mu f \colon \mathcal{I}.\, \lambda x.\, \mathbf{e}) \colon \mathcal{I}}\ \mathcal{I} = \textstyle\bigwedge_{i \in I} T_i' \to T_i \qquad [\text{S-Appl}]\ \frac{\Gamma \vdash \mathbf{e}_1 \colon \langle t' \to t \rangle \qquad \Gamma \vdash \mathbf{e}_2 \colon t'}{\Gamma \vdash \mathbf{e}_1\, \mathbf{e}_2 \colon \langle t \rangle}$$

$$[\text{S-Pair}]\ \frac{\Gamma \vdash \mathbf{e}_1 \colon t_1 \qquad \Gamma \vdash \mathbf{e}_2 \colon t_2}{\Gamma \vdash (\mathbf{e}_1, \mathbf{e}_2) \colon t_1 \times t_2} \qquad [\text{S-Proj}]\ \frac{\Gamma \vdash \mathbf{e} \colon \langle t_1 \times t_2 \rangle}{\Gamma \vdash \pi_i\, \mathbf{e} \colon \langle t_i \rangle}$$

$$[\text{S-Case}]\ \frac{\Gamma \vdash \mathbf{e}_0 \colon \langle t' \rangle \quad \big(\text{either } t' \leq \neg\tau \text{ or } \Gamma, x \colon (t' \wedge \tau) \vdash \mathbf{e}_1 \colon t\big) \quad \big(\text{either } t' \leq \tau \text{ or } \Gamma, x \colon (t' \setminus \tau) \vdash \mathbf{e}_2 \colon t\big)}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2\big) \colon \langle t \rangle}$$

**Figure 2** Typing rules for the source language.

It can be immediately reduced to $\mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ y = (3, x)\ \mathsf{in}\ e_1$ by applying [Ctx] and [Case1], because $\mathsf{typeof}((3, x)) = b_3 \times \mathbb{1} \leq \mathsf{Int} \otimes \mathbb{1}$. However, we can also use [Ctx] to reduce $e$, if it is reducible: we do so by writing the expression as $\mathsf{let}\ x = e\ \mathsf{in}\ E[x]$, where $E$ is $(y = (3, [\,]))) \in (\mathsf{Int} \otimes \mathbb{1})\ ?\ e_1 : e_2$. To model a lazy implementation more faithfully, we should forbid this reduction and state that $(x = F) \in \tau\ ?\ e : e$ is a context only if it cannot be reduced by [Case1] or [Case2].

In both cases, we have chosen a non-deterministic semantics because it is less restrictive: as a consequence, the soundness result will also hold for semantics which fix an order.

## 4 Type system

We define two typing relations for the source language and the internal language.

A *type environment* $\Gamma$ is a finite mapping of type variables to types. We write $\varnothing$ for the empty environment. We say that a type environment $\Gamma$ is *well-formed* if, for all $(x \colon t) \in \Gamma$, we have $t \not\simeq \mathbb{0}$. Since we want to ensure that the empty type is never derivable, we will only consider well-formed type environments in the soundness proof.

### 4.1 Type system for the source language

Figure 2 presents the typing rules for the source language. The subsumption rule [S-Subsum] is used to apply subtyping. Notably, it allows expressions with surely converging types (like a pair with type $\mathsf{Int} \times \mathsf{Bool}$) to be used where diverging types are expected: $t \leq \langle t \rangle$ holds for every $t$ (since $[\![t]\!] \subseteq [\![t]\!] \cup \{\bot\} = [\![t \vee \bot]\!] = [\![\langle t \rangle]\!]$). The rules [S-Var] and [S-Const] for variables and constants are standard. The [S-Abstr] rule for functions is also straightforward. Function interfaces have the form $\bigwedge_{i \in I} T_i' \to T_i$, that is, $\bigwedge_{i \in I} \langle T_i' \rangle \to \langle T_i \rangle$ (expanding the definition of $\to$). To type a function $\mu f \colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$, we check that it has all the arrow types in $\mathcal{I}$. Namely, for every arrow $T_i' \to T_i$ (i.e., $\langle T_i' \rangle \to \langle T_i \rangle$), we assume that $x$ has type $\langle T_i' \rangle$ and that the recursion variable $f$ has type $\mathcal{I}$, and we check that the body has type $\langle T_i \rangle$.

The [S-Appl] rule is the first one that deals with $\bot$ in a non-trivial way. In call-by-value semantic subtyping systems, to type an application $\mathbf{e}_1\, \mathbf{e}_2$ with a type $t$, the standard *modus ponens* rule (e.g., the one from the simply-typed $\lambda$-calculus) is used: $\mathbf{e}_1$ must have type $t' \to t$

and $\mathbf{e}_2$ must have type $t'$. Here, instead, we allow the function to have the type $\langle t' \to t \rangle$ (i.e., $(t' \to t) \vee \bot$) to make application possible also when $\mathbf{e}_1$ might diverge. We use $\langle t \rangle$ as the type of the whole application, signifying that it might diverge. As anticipated, we do not try to predict whether applications will converge. The rule [S-Pair] for pairs is standard; [S-Proj] handles $\bot$ as in applications.

[S-Case] is the most complex rule, but it corresponds closely to that of [20]. Strictly speaking it is not a single inference rule, but a shorthand way of writing four distinct rules with partially different premises and side conditions, here abbreviated in the form "either ... or ...". To type $(x = \mathbf{e}_0) \in \tau \ ? \ \mathbf{e}_1 : \mathbf{e}_2$ we first type $\mathbf{e}_0$ with some type $\langle t' \rangle$. Then, we type the two branches $\mathbf{e}_1$ and $\mathbf{e}_2$. We do not always have to type both (because of the "either ... or ..." conditions) but for now assume that we do. While typing either branch, we extend the environment with a binding for $x$. For the first branch, the type for $x$ is $t' \wedge \tau$, a subtype of $\langle t' \rangle$: this type is sound because the first branch is only evaluated if $\mathbf{e}_0$ evaluates to an answer (meaning we can remove the union with $\bot$ in $\langle t' \rangle$) and if this answer has type $\tau$. Conversely, for the second branch, $x$ is given type $t' \setminus \tau$, that is, $t' \wedge \neg\tau$. Finally, if the branches have type $t$, the whole typecase is given type $\langle t \rangle$ since its evaluation may diverge in case $\mathbf{e}_0$ diverges.

Now let us consider the conditions "either ... or ...". We need to type the first branch only when $t' \not\leq \neg\tau$; if, conversely, $t' \leq \neg\tau$, then we know that the first branch can never be selected (an expression of type $\neg\tau$ cannot reduce to a result of type $\tau$) and thus we do not need to type it. The reasoning for the second branch is analogous. The two conditions are pivotal to type overloaded functions defined by typecases. For example, a negation function implemented as $\mu f\colon \mathcal{I}.\, \lambda x.\, (y = x) \in b_{\mathsf{true}} \ ? \ \mathsf{false} : \mathsf{true}$, with $\mathcal{I} = (b_{\mathsf{true}} \to b_{\mathsf{false}}) \wedge (b_{\mathsf{false}} \to b_{\mathsf{true}})$, could not be typed without these conditions.

In the syntax we have restricted the type $\tau$ in typecases requiring $\tau \not\simeq \mathbb{1}$ and $\tau \not\simeq \mathbb{0}$. Typecases where these conditions do not hold are uninteresting, since they do not actually check anything. The rule [S-Case] would be unsound for them because these typecases can reduce to one branch even if $\mathbf{e}_0$ is a diverging expression that does not evaluate to an answer. For instance, if $\bar{\mathbf{e}}$ has type $\bot$ (that is, $\langle \mathbb{0} \rangle$), then $(x = \bar{\mathbf{e}}) \in \mathsf{Int} \ ? \ 1 : 2$ could be given any type, including unsound ones like $\langle \mathsf{Bool} \rangle$. To allow these typecases, we could add the side condition "$\tau \not\simeq \mathbb{1}$ and $\tau \not\simeq \mathbb{0}$" to [S-Case] and give two specialized rules as follows:

$$\frac{\Gamma \vdash \mathbf{e}_0 \colon t' \qquad \Gamma, x \colon t' \vdash \mathbf{e}_1 \colon t}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \tau \ ? \ \mathbf{e}_1 : \mathbf{e}_2\big) \colon \langle t \rangle} \ \tau \simeq \mathbb{1} \qquad \frac{\Gamma \vdash \mathbf{e}_0 \colon t' \qquad \Gamma, x \colon t' \vdash \mathbf{e}_2 \colon t}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \tau \ ? \ \mathbf{e}_1 : \mathbf{e}_2\big) \colon \langle t \rangle} \ \tau \simeq \mathbb{0}$$

## 4.2 Type system for the internal language

Figure 3 presents the typing rules for the internal language. These include a new rule for let expressions and a modified rule for $\lambda$-abstractions; the other rules are the same as those for the source language (except for the different syntax of typecases).

The [S-Abstr] rule for the source language derived the type $\mathcal{I}$ for $\mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$. The rule for the internal language, instead, allows us to derive a subtype of $\mathcal{I}$ of the form $\mathcal{I} \wedge t$, where $t$ is an intersection of negations of arrow types. The arrows in $t$ can be chosen freely providing that the intersection $\mathcal{I} \wedge t$ remains non-empty. This rule (directly taken from [20]) can look surprising. For example, it allows us to type $\mu f\colon (\mathsf{Int} \twoheadrightarrow \mathsf{Int}).\, \lambda x.\, x$ as $(\mathsf{Int} \twoheadrightarrow \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ even though, disregarding the interface, the function does map booleans to booleans. But the language is explicitly typed, and thus we can't ignore interfaces (indeed, the function does not have type $\mathsf{Bool} \to \mathsf{Bool}$). The purpose of the rule is to ensure that, given any function and any type $t$, either the function has type $t$ or it has type $\neg t$.

$$[\textsc{Subsum}]\ \frac{\Gamma \vdash e\colon t' \qquad t' \leq t}{\Gamma \vdash e\colon t} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x\colon t}\ \Gamma(x) = t \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c\colon b_c}$$

$$[\textsc{Abstr}]\ \frac{\forall i \in I.\quad \Gamma, f\colon \mathcal{I}, x\colon \langle T'_i \rangle \vdash e\colon \langle T_i \rangle}{\Gamma \vdash (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\colon \mathcal{I} \wedge t} \quad \begin{array}{l} \mathcal{I} = \bigwedge_{i\in I} T'_i \to T_i \\ t = \bigwedge_{j\in J} \neg(t'_j \to t_j) \\ \mathcal{I} \wedge t \not\simeq \mathbb{0} \end{array} \qquad [\textsc{Appl}]\ \frac{\Gamma \vdash e_1\colon \langle t' \to t \rangle \qquad \Gamma \vdash e_2\colon t'}{\Gamma \vdash e_1\, e_2\colon \langle t \rangle}$$

$$[\textsc{Pair}]\ \frac{\Gamma \vdash e_1\colon t_1 \qquad \Gamma \vdash e_2\colon t_2}{\Gamma \vdash (e_1, e_2)\colon t_1 \times t_2} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e\colon \langle t_1 \times t_2 \rangle}{\Gamma \vdash \pi_i\, e\colon \langle t_i \rangle}$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash \varepsilon\colon \langle t' \rangle \quad \big(\text{either } t' \leq \neg\tau \text{ or } \Gamma, x\colon (t' \wedge \tau) \vdash e_1\colon t\big) \quad \big(\text{either } t' \leq \tau \text{ or } \Gamma, x\colon (t' \setminus \tau) \vdash e_2\colon t\big)}{\Gamma \vdash \big((x = \varepsilon) \in \tau\ ?\ e_1 : e_2\big)\colon \langle t \rangle}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1\colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \Gamma, x\colon t_i \vdash e_2\colon t}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\colon t}$$

**Figure 3** Typing rules for the internal language.

This property matches the intuitive view of types as sets of values that underpins semantic subtyping. While in our system we do not really interpret types as sets of values (since $\bot$ is non-empty and yet uninhabited by values), the property is still needed to prove subject reduction. A consequence of this property is that a value (i.e., a constant or a $\lambda$-abstraction) of type $t_1 \vee t_2$ has always either type $t_1$ or type $t_2$. (In the case of constants, this is obtained directly by reasoning on subtyping, so we don't need a rule to assign negation types to them.)

The [Let] rule combines a standard rule for (monomorphic) binders with a union disjunction rule: it lets us decompose the type of $e_1$ as a union and type the body of the let once for each summand in the union. The purpose of this rule was hinted at in the Introduction and will be discussed again in Section 4.3, where we show that this rule – combined with the property on union types above – is central to this work: it is the key technical feature that ensures the soundness of the system (see in particular Lemma 4.9 later on). For the time being, just note that the type of $e_1$ can be decomposed in arbitrarily complex ways by applying subsumption. For example, if $e_1$ is a pair of type $(\mathsf{Int} \vee \mathsf{Bool}) \times (\mathsf{Int} \vee \mathsf{Bool})$, by applying [Subsum] we can type it as $(\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Int} \times \mathsf{Bool}) \vee (\mathsf{Bool} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ and then type $e_2$ once for each of the four summands.

The [Abstr] and [Let] rules introduce non-determinism in the choice of the negations to introduce and of how to decompose types as unions. This would not complicate a practical implementation, since a typechecker would only need to check the source language.

## 4.3 Properties of the type system

Full results about the type system, including proofs, are available in the extended version [22]. Here we report the main results and describe the technical difficulties we met to obtain them.

First, we can easily show by induction that compilation from the source language to the internal language preserves typing.

▶ **Proposition 4.1.** *If* $\Gamma \vdash \mathbf{e}\colon t$, *then* $\Gamma \vdash \lceil \mathbf{e} \rceil\colon t$.

We show the soundness property for our type system ("well-typed programs do not go wrong"), following the well-known syntactic approach of Wright and Felleisen [26], by proving the two properties of *progress* and *subject reduction* for the internal language.

▶ **Theorem 4.2** (Progress). *Let $\Gamma$ be a well-formed type environment. Let $e$ be an expression that is well-typed in $\Gamma$ (that is, $\Gamma \vdash e : t$ holds for some $t$). Then $e$ is an answer, or $e$ is of the form $E[\![x]\!]$, or $\exists e'.\, e \rightsquigarrow e'$.*

▶ **Theorem 4.3** (Subject reduction). *Let $\Gamma$ be a well-formed type environment. If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.*

The statement of progress is adapted to call-by-need: it applies also to expressions that are typed in a non-empty $\Gamma$ and it allows a well-typed expression to have the form $E[\![x]\!]$.

As a corollary of these results, we obtain the following statement for soundness.

▶ **Corollary 4.4** (Type soundness). *Let $e$ be a well-typed, closed expression (that is, $\varnothing \vdash e : t$ holds for some $t$). If $e \rightsquigarrow^* e'$ and $e'$ cannot reduce, then $e'$ is an answer and $\varnothing \vdash e' : t$.*

The soundness result for the internal language implies soundness for the source language.

▶ **Corollary 4.5** (Type soundness for the source language). *Let $\mathbf{e}$ be a well-typed, closed source language expression (that is, $\varnothing \vdash \mathbf{e} : t$ holds for some $t$). If $\lceil \mathbf{e} \rceil \rightsquigarrow^* e'$ and $e'$ cannot reduce, then $e'$ is an answer and $\varnothing \vdash e' : t$.*

We summarize here some of the crucial properties required to derive the results above. We also resume the discussion of the motivations behind our choice of call-by-need.

We introduced the $\bot$ type for diverging expressions because assigning the type $\mathbb{0}$ to any expression causes unsoundness. We must hence ensure that no expression can be assigned the type $\mathbb{0}$. In well-formed type environments, we can prove this easily by induction.

▶ **Lemma 4.6.** *Let $\Gamma$ be a well-formed type environment. If $\Gamma \vdash e : t$, then $t \not\simeq \mathbb{0}$.*

**Call-by-name and call-by-need.** In the Introduction, we have given two reasons for our choice of call-by-need rather than call-by-name. One is that the system is only sound for call-by-name if we make assumptions on the semantics that might not hold in an extended language: for example, introducing an expression that can reduce non-deterministically either to an integer or to a boolean would break soundness. The other reason is that, even when these assumptions hold (and when presumably call-by-name and call-by-need are observationally equivalent), call-by-need is better suited to the soundness proof.

Let us review the example from the Introduction. Consider the function $\mu f : \mathcal{I}.\, \lambda x.\, (x, x)$ in the source language, where $\mathcal{I} = (\mathsf{Int} \twoheadrightarrow \mathsf{Int} \otimes \mathsf{Int}) \wedge (\mathsf{Bool} \twoheadrightarrow \mathsf{Bool} \otimes \mathsf{Bool})$. It is well-typed with type $\mathcal{I}$. By subsumption, it also has the type $(\mathsf{Int} \vee \mathsf{Bool}) \twoheadrightarrow (\mathsf{Int} \otimes \mathsf{Int}) \vee (\mathsf{Bool} \otimes \mathsf{Bool})$, which is a supertype of $\mathcal{I}$: in general we have $(t_1' \to t_1) \wedge (t_2' \to t_2) \leq (t_1' \vee t_2') \to (t_1 \vee t_2)$ and therefore $(t_1' \twoheadrightarrow t_1) \wedge (t_2' \twoheadrightarrow t_2) \leq (t_1' \vee t_2') \twoheadrightarrow (t_1 \vee t_2)$.

Therefore, if $\bar{\mathbf{e}}$ has type $\mathsf{Int} \vee \mathsf{Bool} \vee \bot$, the application $(\mu f : \mathcal{I}.\, \lambda x.\, (x, x))\, \bar{\mathbf{e}}$ is well-typed with type $(\mathsf{Int} \otimes \mathsf{Int}) \vee (\mathsf{Bool} \otimes \mathsf{Bool}) \vee \bot$. Assume that $\bar{\mathbf{e}}$ can reduce either to an integer or to a boolean: for instance, assume that both $\bar{\mathbf{e}} \rightsquigarrow 3$ and $\bar{\mathbf{e}} \rightsquigarrow \mathsf{true}$ can occur.

With call-by-name, $(\mu f : \mathcal{I}.\, \lambda x.\, (x, x))\, \bar{\mathbf{e}}$ reduces to $(\bar{\mathbf{e}}, \bar{\mathbf{e}})$; then, the two occurrences of $\bar{\mathbf{e}}$ reduce independently. It is intuitively unsound to type it as $(\mathsf{Int} \otimes \mathsf{Int}) \vee (\mathsf{Bool} \otimes \mathsf{Bool}) \vee \bot$: there is no guarantee that the two components of the pair will be of the same type once they are reduced. We can find terms that break subject reduction. Assume for example that there exists a boolean "$\mathsf{and}$" operation; then this typecase is well-typed (as $\langle \mathsf{Bool} \rangle$) but unsafe:

$$(y = (\mu f : \mathcal{I}.\, \lambda x.\, (x, x))\, \bar{\mathbf{e}}) \in (\mathsf{Int} \otimes \mathsf{Int})\ ?\ \mathsf{true} : (\pi_1\, y\ \mathsf{and}\ \pi_2\, y)\ .$$

Since the application has type $\langle(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})\rangle$, to type the second branch of the typecase we can assume that $y$ has the type $((\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool}))\setminus(\mathsf{Int}\otimes\mathsf{Int})$, which is a subtype of $\mathsf{Bool}\otimes\mathsf{Bool}$ (it is actually equivalent to $(\mathsf{Bool}\otimes\mathsf{Bool})\setminus(\bot\times\bot)$). Therefore, both $\pi_1\,y$ and $\pi_2\,y$ have type $\langle\mathsf{Bool}\rangle$. We deduce then that $(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)$ has type $\langle\mathsf{Bool}\rangle$ as well (we assume that "and" is defined so as to handle arguments of type $\bot$ correctly).

A possible reduction in a call-by-name semantics would be the following:

$$\begin{aligned}&(y=(\mu f\colon\mathcal{I}.\,\lambda x.\,(x,x))\,\bar{\mathbf{e}})\in(\mathsf{Int}\otimes\mathsf{Int})\ ?\ \mathsf{true}:(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)\\ \rightsquigarrow\ &(y=(\bar{\mathbf{e}},\bar{\mathbf{e}}))\in(\mathsf{Int}\otimes\mathsf{Int})\ ?\ \mathsf{true}:(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)\end{aligned}$$

(the typecase must force the evaluation of $(\bar{\mathbf{e}},\bar{\mathbf{e}})$ to know which branch should be selected)

$$\rightsquigarrow^*\ (y=(\mathsf{true},\bar{\mathbf{e}}))\in(\mathsf{Int}\otimes\mathsf{Int})\ ?\ \mathsf{true}:(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)$$

(now we know that the first branch is impossible, so the second is chosen)

$$\rightsquigarrow\ \pi_1\,(\mathsf{true},\bar{\mathbf{e}})\ \mathsf{and}\ \pi_2\,(\mathsf{true},\bar{\mathbf{e}})\ \rightsquigarrow\ \mathsf{true}\ \mathsf{and}\ \bar{\mathbf{e}}\ \rightsquigarrow\ \bar{\mathbf{e}}\ \rightsquigarrow\ 3$$

The integer 3 is not a Bool: this disproves subject reduction for call-by-name if the language contains expressions like $\bar{\mathbf{e}}$. No such expressions exist in our current language, but they could be introduced if we extended it with non-deterministic constructs like $\mathsf{rnd}(t)$ from [20].

Since we use a call-by-need semantics, instead, expressions such as $\bar{\mathbf{e}}$ do not pose problems for soundness. With call-by-need, $(\mu f\colon\mathcal{I}.\,\lambda x.\,(x,x))\,\bar{\mathbf{e}}$ reduces to $\mathsf{let}\ f=\mu f\colon\mathcal{I}.\,\lambda x.\,(x,x)\ \mathsf{in}\ \mathsf{let}\ x=\bar{\mathbf{e}}\ \mathsf{in}\ (x,x)$. The occurrences of $x$ in the pair are only substituted when $\bar{\mathbf{e}}$ has been reduced to an answer, so they cannot reduce independently.

To ensure subject reduction, we allow the rule for let bindings to split unions in the type of the bound term. This means that the following derivation is allowed.

$$\frac{\Gamma\vdash\bar{\mathbf{e}}\colon\mathsf{Int}\vee\mathsf{Bool}\qquad\Gamma,x\colon\mathsf{Int}\vdash(x,x)\colon\mathsf{Int}\otimes\mathsf{Int}\qquad\Gamma,x\colon\mathsf{Bool}\vdash(x,x)\colon\mathsf{Bool}\otimes\mathsf{Bool}}{\Gamma\vdash\mathsf{let}\ x=\bar{\mathbf{e}}\ \mathsf{in}\ (x,x)\colon(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})}$$

**Proving subject reduction: main lemmas.** While the typing rule for let bindings is simple to describe, proving subject reduction for the reduction rules [LetV] and [LetP] (those that actually perform substitutions) is challenging. For the reduction $\mathsf{let}\ x=v\ \mathsf{in}\ E[\![x]\!]\rightsquigarrow(E[\![x]\!])[v/x]$, we show the following results.

▶ **Lemma 4.7.** *Let $v$ be a value that is well-typed in $\Gamma$ (i.e., $\Gamma\vdash v\colon t'$ holds for some $t'$). Then, for every type $t$, we have either $\Gamma\vdash v\colon t$ or $\Gamma\vdash v\colon\neg t$.*

▶ **Corollary 4.8.** *If $\Gamma\vdash v\colon\bigvee_{i\in I}t_i$, then there exists an $i_0\in I$ such that $\Gamma\vdash v\colon t_{i_0}$.*

Consider for example the reduction $\mathsf{let}\ x=v\ \mathsf{in}\ (x,x)\rightsquigarrow(v,v)$. If $v$ has type $\mathsf{Int}\vee\mathsf{Bool}$, then $\mathsf{let}\,x=v\ \mathsf{in}\ (x,x)$ has type $(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})$ as in the derivation above. Without this corollary, for $(v,v)$ we could only derive the type $(\mathsf{Int}\vee\mathsf{Bool})\times(\mathsf{Int}\vee\mathsf{Bool})$, which is not a subtype of the type deduced for the redex. Applying the corollary, we deduce that $v$ has either type Int or Bool; in both cases $(v,v)$ can be given the type $(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})$.

These results are also needed in semantic subtyping for strict languages to prove subject reduction for applications. To ensure them, following [20], we have added in the type system for the internal language the possibility of typing functions with negations of arrow types.

The reduction $\mathsf{let}\ x=(e_1,e_2)\ \mathsf{in}\ E[\![x]\!]\rightsquigarrow\ \mathsf{let}\ x_1=e_1\ \mathsf{in}\ \mathsf{let}\ x_2=e_2\ \mathsf{in}\ (E[\![x]\!])[(x_1,x_2)/x]$, instead, is dealt with by the following lemma.

▶ **Lemma 4.9.** *If* $\Gamma \vdash (e_1, e_2) \colon \bigvee_{i\in I} t_i$*, then there exist two types* $\bigvee_{j\in J} t_j$ *and* $\bigvee_{k\in K} t_k$ *such that* $\Gamma \vdash e_1 \colon \bigvee_{j\in J} t_j$*,* $\Gamma \vdash e_2 \colon \bigvee_{k\in K} t_k$*, and* $\forall j \in J.\, \forall k \in K.\, \exists i \in I.\, t_j \times t_k \leq t_i$*.*

This is the result we need for the proof: $\mathsf{let}\, x = (e_1, e_2)\, \mathsf{in}\, E[\![x]\!]$ is typed by assigning a union type to $(e_1, e_2)$ and then typing $E[\![x]\!]$ once for every $t_i$ in the union, while the reduct $\mathsf{let}\, x_1 = e_1\, \mathsf{in}\, \mathsf{let}\, x_2 = e_2\, \mathsf{in}\, (E[\![x]\!])[^{(x_1,x_2)}/_x]$ must be typed by typing $e_1$ and $e_2$ with two union types and then typing the substituted expression with every product $t_j \times t_k$. Showing that each $t_j \times t_k$ is a subtype of a $t_i$ ensures that the substituted expression is well-typed. The proof consists in recognizing that the union $\bigvee_{i\in I} t_i$ must be a decomposition into a union of some type $t_1 \times t_2$ and that therefore $t_1$ and $t_2$ can be decomposed separately into two unions.

These results rely on the distinction between types that contain $\bot$ and those that do not: they would not hold if we assumed that every type implicitly contained $\bot$. For instance, adding $\bot$ implicitly to any type would essentially mean interpreting products as $[\![t_1 \times t_2]\!] = ([\![t_1]\!] \cup \{\bot\}) \times ([\![t_2]\!] \cup \{\bot\})$ instead of $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$. This would make Lemma 4.9 fail. Its proof relies on being able to find, given any type $t$ such that $t \leq \mathbb{1} \times \mathbb{1}$ (that is, a type whose set-theoretic interpretation consists entirely of pairs), a union type $\bigvee_{i\in I} t_i^1 \times t_i^2$ such that $t \simeq \bigvee_{i\in I} t_i^1 \times t_i^2$ (Lemma A.10 in the extended version [22]). This would not hold with the modified interpretation: for example, the type $(\mathsf{Int} \times \mathsf{Bool}) \setminus (\mathbb{0} \times \mathbb{0})$ is a subtype of $\mathbb{1} \times \mathbb{1}$ but cannot be expressed as a union of product types.

Despite some technical difficulties, call-by-need seems quite suited to the soundness proof. Hence, it would probably be best to use it for the proof even if we assumed explicitly that the language does not include problematic expressions like $\mathsf{rnd}(t)$. Soundness would then also hold for a call-by-name semantics that it is observationally equivalent to call-by-need.

## 5 A discussion on the interpretation of types

We have shown in the previous sections that a set-theoretic interpretation of types, adapted to take into account divergence (Definition 2.3), can be the basis for designing a sound type system for languages with lazy evaluation. In this section, we analyze the relation between such an interpretation and the expressions that are actually definable in the language.

Let us first recap some notions from [20]. The initial intuition which guides semantic subtyping is to see a type as the set of values of that type in the language we consider: for example, to see $\mathsf{Int} \to \mathsf{Bool}$ as the set of $\lambda$-abstractions of type $\mathsf{Int} \to \mathsf{Bool}$. However, we cannot directly define the interpretation of a type $t$ as the set $\{\, v \mid \varnothing \vdash v \colon t \,\}$, because the typing relation $\varnothing \vdash v \colon t$ depends on the definition of subtyping, which depends in turn on the interpretation of types. Frisch, Castagna and Benzaken [20] avoid this circularity by giving an interpretation $[\![\cdot]\!]$ of types as subsets of an interpretation domain where finite relations replace $\lambda$-abstractions.

This interpretation (like ours except that there is no $\bot$) is used to define subtyping and the typing relation. Then, the following result is shown:

$$\forall t_1, t_2.\, [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!]_{\mathcal{V}} \subseteq [\![t_2]\!]_{\mathcal{V}} \qquad \text{where } [\![t]\!]_{\mathcal{V}} \stackrel{\text{def}}{=} \{\, v \mid \varnothing \vdash v \colon t \,\}$$

This result states that a type $t_1$ is a subtype of a type $t_2$ ($t_1 \leq t_2$, which is defined as $[\![t_1]\!] \subseteq [\![t_2]\!]$) if and only if every value $v$ that can be assigned the type $t_1$ can also be assigned the type $t_2$. Showing the result above implies that, once the type system is defined, we can indeed reason on subtyping by reasoning on inclusion between sets of values.[1]

[1] The circularity is avoided since the typing relation in $\{\, v \mid \varnothing \vdash v \colon t \,\}$ is defined using $[\![\cdot]\!]$ and not $[\![\cdot]\!]_{\mathcal{V}}$.

This result is useful in practice, since, when typechecking fails because a subtyping judgment $t_1 \leq t_2$ does not hold, we know that there exists a value $v$ such that $\varnothing \vdash v\colon t_1$ holds while $\varnothing \vdash v\colon t_2$ does not. This value $v$ can be shown as a witness to the unsoundness of the program while reporting the error.[2] Moreover, at a more foundational level, the result nicely formalizes the intuition that types statically approximate computations, in the sense that a type $t$ corresponds to the set of all possible values of expressions of type $t$.

In the following we discuss how an analogous result could hold with a non-strict semantics. First of all, clearly the correspondence cannot be between interpretations of types and sets of values as in [20], since then we would identify $\bot$ with $\mathbb{0}$. Hence we should consider, rather than values, sets of "results" of some kind, including (a representation of) divergence.

However, whichever notion of result we consider, it is hard to define an interpretation domain of types such that the desired correspondence holds, that is, such that a type $t$ corresponds to the set of all possible results of expressions of type $t$. As the reader can expect, the key challenge is to provide an interpretation where an arrow type $t_1 \to t_2$ corresponds, as it seems sensible, to the set of $\lambda$-abstractions $\{\, (\mu f\colon \mathcal{I}.\, \lambda x.\, e) \mid \varnothing \vdash (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\colon t_1 \to t_2 \,\}$. For instance, our proposed definition of $[\![\cdot]\!]$ is sound with respect to this correspondence, but not complete, that is, not precise enough. We devote the rest of this section to explain why and to discuss the possibility of obtaining a complete definition. Consider the type $\mathsf{Int} \to \mathbb{0}$. By Definition 2.3, we have

$$\begin{aligned}[\![\mathsf{Int} \to \mathbb{0}]\!] &= \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \in [\![\mathsf{Int}]\!] \implies d' \in [\![\mathbb{0}]\!] \,\} \\ &= \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \notin [\![\mathsf{Int}]\!] \,\}\end{aligned}$$

(since $[\![\mathbb{0}]\!] = \varnothing$, the implication can only be satisfied if $d \notin [\![\mathsf{Int}]\!]$). This type is not empty, therefore, if a result similar to that of [20] held, we would expect to be able to find a function $\mu f\colon \mathcal{I}.\, \lambda x.\, e$ such that $\varnothing \vdash (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\colon \mathsf{Int} \to \mathbb{0}$. Alas, no such function can be defined in our language. This is easy to check: interfaces must include $\bot$ in the codomain of every arrow (since they use the $\to$ form), so no interface can be a subtype of $\mathsf{Int} \to \mathbb{0}$. Lifting this syntactic restriction to allow any arrow type in interfaces would not solve the problem: for a function to have type $\mathsf{Int} \to \mathbb{0}$, its body must have type $\mathbb{0}$, which is impossible, and indeed *must* be impossible for the system to be sound. It is therefore to be expected that $\mathsf{Int} \to \mathbb{0}$ is uninhabited in the language. This means that our current definition of $[\![\mathsf{Int} \to \mathbb{0}]\!]$ as a non-empty type is imprecise.

Changing $[\![\cdot]\!]$ to make the types of the form $t \to \mathbb{0}$ empty is easy, but it does not solve the problem in general. Using intersection types we can build more challenging examples: for instance, consider the type $(\mathsf{Int} \vee \mathsf{Bool} \to \mathsf{Int}) \wedge (\mathsf{Int} \vee \mathsf{String} \to \mathsf{Bool})$. While neither codomain is empty, and neither arrow should be empty, the whole intersection should: no function, when given an $\mathsf{Int}$ as argument, can return a result which is both an $\mathsf{Int}$ and a $\mathsf{Bool}$.

In the call-by-value case, it makes sense to have $\mathsf{Int} \to \mathbb{0}$ and the intersection type above be non-empty, because they are inhabited by functions that diverge on integers. This is because divergence is not represented in the types (or, to put it differently, because it is represented by the type $\mathbb{0}$). A type like $t_1 \to t_2$ is interpreted as a specification of *partial correctness*: a function of this type, when given an argument in $t_1$, either diverges or returns a result in $t_2$. In our system, we have introduced a separate non-empty type for divergence. Hence, we should see a type as specifying *total* correctness, where divergence is allowed only for functions whose codomain includes $\bot$.

[2] In case of a type error, the CDuce compiler shows to the programmer a default value for the type $t_1 \setminus t_2$. Some heuristics are used to build a value in which only the part relevant to the error is detailed.

Let us look again at the current interpretation of arrow types.

$$[\![t_1 \to t_2]\!] = \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \,\}$$

An arrow type is seen as a set of finite relations: we represent functions extensionally and approximate them with all their finite subsets. We use relations instead of functions to account for non-determinism. Within a relation, a pair $(d, d')$ means that the function returns the output $d'$ on the input $d$; a pair $(d, \Omega)$ that the function crashes on $d$; divergence is represented simply by the absence of a pair. In this way, as said above, a function diverging on some element of $[\![t_1]\!]$ could erroneously belong to the set even if $[\![t_2]\!]$ does not contain $\bot$.

To formalize the requirement of totality on the domain, we could modify the definition in this way:

$$[\![t_1 \to t_2]\!] = \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \mathsf{dom}(R) \supseteq [\![t_1]\!] \text{ and } \forall (d, d') \in R.\, d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \,\}$$

(where $\mathsf{dom}(R) = \{\, d \mid \exists d' \in \mathcal{D}.\, (d, d') \in R \,\}$).

However, if we consider only finite relations as above, the definition makes no sense, since $[\![t_1]\!] \subseteq \mathsf{dom}(R)$ can hold only when $[\![t_1]\!]$ is finite, whereas types can have infinite interpretations. On the contrary, if we allowed relations to be infinite, then the set $\mathcal{D}$ would have to satisfy the equality $\mathcal{D} = \mathcal{C} \uplus (\mathcal{D} \times \mathcal{D}) \uplus \mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega)$ (where $\uplus$ denotes disjoint union), but no such set exists: the cardinality of $\mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega)$ is always strictly greater than that of $\mathcal{D}$.

Frisch, Castagna and Benzaken [20] point out this problem and use finite relations in the domain to avoid it. They motivate this choice with the observation that, while finite relations are not really appropriate to describe functions in a language (since these might have an infinite domain), they are suitable to describe types as far as subtyping is concerned. Indeed, we do not really care what the elements in the interpretation of a type are, but only how they are related to those in the interpretations of other types. It can be shown that

$$\forall t_1, t_1', t_2, t_2'.\ [\![t_1' \to t_1]\!] \subseteq [\![t_2' \to t_2]\!] \iff ([\![t_1']\!] \rightharpoonup [\![t_1]\!]) \subseteq ([\![t_2']\!] \rightharpoonup [\![t_2]\!])$$

where $X \rightharpoonup Y \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \in X \implies d' \in Y \,\}$ builds the set of possibly infinite relations. This can be generalized to more complex types:

$$\left[\!\!\left[\bigwedge_{i \in P} t_i' \to t_i\right]\!\!\right] \subseteq \left[\!\!\left[\bigvee_{i \in N} t_i' \to t_i\right]\!\!\right] \iff \bigcap_{i \in P} ([\![t_i']\!] \rightharpoonup [\![t_i]\!]) \subseteq \bigcup_{i \in N} ([\![t_i']\!] \rightharpoonup [\![t_i]\!])\,.$$

In [20], the authors argue that the restriction to finite relations does not compromise the precision of subtyping. For reasons of space we do not elaborate further on this, and we direct the interested reader to their work and the notions of *extensional interpretation* and of *model* therein.

Let us try to proceed analogously in our case: that is, find a new interpretation of types that matches the behaviour of possibly infinite relations that are total on their domain, while introducing an approximation to ensure that the domain is definable. The latter point means, notably, that functions must be represented as finite objects. The following definition of a *model* specifies the properties that such an interpretation should satisfy.

▶ **Definition 5.1** (Model). *A function $(\!|\cdot|\!) : \mathcal{T} \to \mathcal{P}(\overline{\mathcal{D}})$ is a* model *if the following hold:*

- *the set $\overline{\mathcal{D}}$ satisfies $\overline{\mathcal{D}} = \{\bot\} \uplus \mathcal{C} \uplus (\overline{\mathcal{D}} \times \overline{\mathcal{D}}) \uplus \overline{\mathcal{D}}^{\mathsf{fun}}$ for some set $\overline{\mathcal{D}}^{\mathsf{fun}}$;*
- *for all $b$, $t$, $t_1$, and $t_2$,*

$$(\!|\bot|\!) = \{\bot\} \qquad (\!|b|\!) = \mathbb{B}(b) \qquad (\!|t_1 \times t_2|\!) = (\!|t_1|\!) \times (\!|t_2|\!) \qquad (\!|t_1 \to t_2|\!) \subseteq (\!|\mathbb{0} \to \mathbb{1}|\!) = \overline{\mathcal{D}}^{\mathsf{fun}}$$
$$(\!|t_1 \vee t_2|\!) = (\!|t_1|\!) \cup (\!|t_2|\!) \qquad (\!|\neg t|\!) = \overline{\mathcal{D}} \setminus (\!|t|\!) \qquad (\!|\mathbb{0}|\!) = \varnothing$$

- *for every finite, non-empty intersection* $\bigwedge_{i\in P} t'_i \to t_i$ *and every finite union* $\bigvee_{i\in N} t'_i \to t_i$,

$$(\!|\textstyle\bigwedge_{i\in P} t'_i \to t_i|\!) \subseteq (\!|\bigvee_{i\in N} t'_i \to t_i|\!) \iff \bigcap_{i\in P} \left((\!|t'_i|\!) \to (\!|t_i|\!)\right) \subseteq \bigcup_{i\in N} \left((\!|t'_i|\!) \to (\!|t_i|\!)\right)$$

*where* $X \to Y \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\overline{\mathcal{D}} \times \overline{\mathcal{D}}) \mid \mathsf{dom}(R) \supseteq X \text{ and } \forall (d, d') \in R.\, d \in X \implies d' \in Y \,\}$.

We set three conditions for an interpretation of types $(\!|\cdot|\!) : \mathcal{T} \to \mathcal{P}(\overline{\mathcal{D}})$ to be a model. The first constrains $\overline{\mathcal{D}}$ to have the same structure as $\mathcal{D}$, except that we do not fix the subset $\overline{\mathcal{D}}^{\mathsf{fun}}$ in which arrow types are interpreted. The second condition fixes the definition of $(\!|\cdot|\!)$ completely except for arrow types. The third condition ensures that subtyping on arrow types behaves as set containment between the sets of relations that are total on the domains of the arrow types.[3]

An interesting result is that, even though we do not know whether an interpretation of types which is a model can actually be found, we can compare a hypothetical model with the interpretation $[\![\cdot]\!]$ defined in Section 2. Indeed $[\![\cdot]\!]$ turns out to be a sound approximation of every model; that is, the subtyping relation $\leq$ defined in Definition 2.4 from $[\![\cdot]\!]$ is contained in every subtyping relation $\leq_{(\!|\cdot|\!)}$ defined from some model $(\!|\cdot|\!)$. We have proven that this holds for non-recursive types:

▶ **Proposition 5.2.** *Let* $(\!|\cdot|\!) : \mathcal{T} \to \mathcal{P}(\overline{\mathcal{D}})$ *be a model. Let* $t_1$ *and* $t_2$ *be two finite (i.e., non-recursive) types. If* $[\![t_1]\!] \subseteq [\![t_2]\!]$*, then* $(\!|t_1|\!) \subseteq (\!|t_2|\!)$.

We conjecture that the result holds for recursive types too, but this proof is left for future work.

Showing that $(\!|\cdot|\!)$ exists would be important to understand the connection between our types and the semantics. To use $(\!|\cdot|\!)$ to define subtyping for the use of a typechecker, though, we would also need to show that the resulting definition is decidable. Otherwise, $[\![\cdot]\!]$ would remain the definition used in a practical implementation since it is sound and decidable, though less precise.

## 6 Conclusion

We have shown how to adapt the framework of semantic subtyping [20] to languages with non-strict semantics. Our type system uses the subtyping relation from [20] unchanged (except for the addition of $\bot$), while the typing rules are reworked to avoid the pathological behaviour of semantic subtyping on empty types. Notably, typing rules for constructs like application and projection must handle $\bot$ explicitly. This ensures soundness for call-by-need.

This approach ensures that the subtyping relation still behaves set-theoretically: we can still see union, intersection, and negation in types as the corresponding operations on sets. We can still use intersection types to express overloading.

The type $\bot$ we introduce has no analogue in well-known type systems like the simply typed $\lambda$-calculus or Hindley-Milner typing. However, $\bot$ never appears explicitly in programs (it does not appear in types of the forms $T$ and $\tau$ given at the beginning of Section 3). Hence, programmers do not need to use it and to consider the difference between terminating and non-terminating types while writing function interfaces or typecases. Still, sub-expressions of a program can have types with explicit $\bot$ (e.g., the type $\mathsf{Int} \vee \bot$). Such types are not expressible in the grammar of types visible to the programmer. Accordingly, error reporting

[3] We do not use the error element $\Omega$ in the definition of $X \to Y$, because the totality requirement makes it unnecessary: errors on a given input can be represented in a relation by the absence of a pair.

is required to be more elaborated, to avoid mentioning internal types that are unknown to the programmer.

A different approach to use semantic subtyping with non-strict languages would be to change the interpretation of types (and, as a result, the definition of subtyping) to avoid the pathological behaviour on $\mathbb{0}$, and then to use standard typing rules.

We have explored this alternative approach, but we have not found it promising. A modified subtyping relation loses important properties – especially results on the decomposition of product types – that we need to prove soundness via subject reduction. The approach we have adopted here is more suited to this technical work. However, a modified subtyping relation could yield an alternative type system for the source language, provided that we can relate it to the current system for the internal language.

We also plan to study more expressive typing rules that can track termination with some precision. For example, we could change the application rule so that it does not always introduce $\bot$. In function interfaces, some arrows could include $\bot$ and some could not: then, overloaded function types would express that a function behaves differently on terminating or diverging arguments. For example, the function $\lambda x.\, x + 1$ could have type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\bot \to \bot)$, while $\lambda x.\, 3$ could have type $\mathbb{1} \to \mathsf{Int}$: the first diverges on diverging arguments, the other always terminates. It would be interesting for future work to explore forms of termination analysis to obtain greater precision. The difficulty is to ensure that the type $\mathbb{0}$ remains uninhabited and that all diverging expressions still have types that include $\bot$. This is trivial in the current system, but it is no longer straightforward with more refined typing rules.

A further direction for future work is to extend the language and the type system we have considered with more features. Notably, polymorphism, gradual typing, and record types are needed to be able to type effectively the Nix Expression Language, which was the starting inspiration for our work.

## References


1 Davide Ancona and Andrea Corradi. Semantic subtyping for imperative object-oriented languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 568–587. ACM, 2016. doi:10.1145/2983990.2983992.

2 Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

3 Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246. ACM, 1995. doi:10.1145/199448.199507.

4 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 51–63. ACM, 2003. doi:10.1145/944705.944711.

5 Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of NoSQL languages. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 101–114. ACM, 2013. doi:10.1145/2429069.2429083.

6 Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 105–116. ACM, 2010.

7 Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, Volume 1, Issue 2, 2005. doi:10.2168/LMCS-1(2:1)2005.

8 Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 198–199. ACM, 2005. doi:10.1145/1069774.1069793.

9 Giuseppe Castagna, Hyeonseung Im, Kim Nguyen, and Véronique Benzaken. A core calculus for XQuery 3.0. In *Programming Languages and Systems*, pages 232–256. Springer, 2015.

10 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):41:1–41:28, 2017. doi:10.1145/3110285.

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

12 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. Part 1: syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 5–17. ACM, 2014. doi:10.1145/2535838.2535840.

13 Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391. ACM, 2016. doi:10.1145/2951913.2951928.

14 Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 94–106. ACM, 2011. doi:10.1145/2034773.2034788.

15 Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 183–193, 1987.

16 Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. In *Formal Techniques for Distributed Systems*, pages 66–82. Springer, 2013.

17 Eelco Dolstra and Andres Löh. NixOS: a purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378. ACM, 2008. doi:10.1145/1411204.1411255.

18 Joshua Dunfield. *A unified system of type refinements.* PhD thesis, Carnegie Mellon University, 2007.

19 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structures*, pages 250–266. Springer, 2003.

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

21 John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

22 T. Petrucciani, G. Castagna, D. Ancona, and E. Zucca. Semantic subtyping for non-strict languages. Extended version. https://arxiv.org/abs/1810.05555, 2018.

23 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282. ACM, 2014.

24 Jérôme Vouillon. Subtyping Union Types. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, pages 415–429. Springer, 2004.

25 Jérôme Vouillon and Paul-André Melliès. Semantic Types: A Fresh Look at the Ideal Model for Types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 52–63. ACM, 2004. doi:10.1145/964001.964006.

**26** Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

**27** Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: A rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):113:1–113:27, 2018. doi:10.1145/3276483.