EPiC Series in Computing

Volume 57, 2018, Pages 515–533

LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Polymorphic success types for Erlang*

Francisco J. López-Fraguas, Manuel Montenegro, and Gorka Suárez-García

[1] Universidad Complutense de Madrid, Madrid, Spain
fraguas@ucm.es
[2] Universidad Complutense de Madrid, Madrid, Spain
montenegro@fdi.ucm.es
[3] Universidad Complutense de Madrid, Madrid, Spain
gorka.suarez@ucm.es

Abstract

Erlang is a dynamically typed concurrent functional language of increasing interest in industry and academia. Official Erlang distributions come equipped with *Dialyzer*, a useful static analysis tool able to anticipate runtime errors by inferring so-called *success types*, which are overapproximations to the real semantics of expressions. However, Dialyzer exhibits two main weaknesses: on the practical side, its ability to deal with functions that are typically polymorphic is rather poor; and on the theoretical side, a fully developed theory for its underlying type system –comparable to, say, Hindley-Milner system– does not seem to exist, something that we consider a regrettable circumstance. This work presents a type derivation system to obtain polymorphic success types for Erlang programs, along with correctness results with respect to a suitable semantics for the language.

1 Introduction

Erlang is a concurrent functional language arousing increasing interest in industry and academia for its strength in producing robust, easy to build and maintain, scalable fault tolerant systems. As is typical with dynamically typed languages, it offers a remarkable flexibility to the task of programming. The price to pay for dynamic types is that many program errors will manifest only as runtime errors, in contrast to the static compilation time errors obtained by type systems *à la* Hindley-Milner [6] adopted by other functional languages like ML or Haskell.

This explains the interest of developing static analysis tools that anticipate to compilation time as many runtime errors as possible. In the case of Erlang, after some attempts [15], the Typer [11] and Dialyzer tools [10, 12, 9] were proposed and are currently into the official distribution of Erlang. They can be used to extract the implicit type information contained in the programs, both for documentation purposes and for finding errors at compile time. In order to respect the actual flexibility of Erlang, an essential design principle of Dialyzer was that it should neither require type annotations from the programmer nor produce *false*

*Work partially supported by the Spanish MINECO project CAVI-ART-2 (TIN2017-86217-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731).

positives. The latter means that signalling a type error should only happen in situations where it is certain that a runtime error will occur[1]. As said in [20], the lemma *'well-typed programs never go wrong'* of Hindley-Milner types is replaced in the Dialyzer approach by *'ill-typed programs always fail'*[2]. To achieve that, Dialyzer infers so-called *success types* [12], that are overapproximations to the real semantics of expressions, so that if a success type representing the empty set of possible values –the type `none()`, in Dialyzer– is inferred for a given expression, this implies that no possible computation for that expression can end successfully producing a value. Notice that we must speak of 'possible computation' because of the non-determinism introduced by concurrency. Expressions having a success type `none()` are the closest analog to ill-typed expressions in standard type systems.

Being a great tool, Dialyzer exhibits however some weaknesses. Its ability to deal with polymorphically typed functions is rather poor. It is not designed to infer by itself polymorphic types. To overcome that, user-given polymorphic type specifications were considered in [9]. But Dialyzer takes those specifications in such a way that most of polymorphism is lost.

A first contribution to address that problem was made in [14], where given an Erlang program with user-given polymorphic type specifications, a new one is synthesized such that Dialyzer, when run over the transformed program, infers more precise types for expressions that use polymorphic functions. However, this approach is limited by its tight dependence of Dialyzer. Any change made to the tool, could affect and even invalidate the transformations proposed. Moreover, proving any theoretical result rely on trusting on non rigorously proved properties of Dialyzer. This is a second relevant weakness of Dialyzer, the lack of a rigorous formalization and a well developed theoretical framework upon which one can justify the technical correctness of the proposals.

The main motivation of this work is to develop a full type system, independent of Dialyzer, with associated type checking and type inference mechanisms that follows the philosophy of success types, coping appropriately with the issues of polymorphism and having at the same time rigorous theoretical foundations. This work presents a polymorphic type system as first step towards this aim. Concretely, we propose a set of typing rules for deriving polymorphic success types for (Core) Erlang programs, and we prove correctness results with respect to a suitable semantics of programs.

The considered semantics is set-valued due to the nondeterministic behaviour of message passing-based concurrency. One problem to face –within the success types view– was that having an ill-typed subexpression does not imply that the whole expression is also ill-typed[3], e.g. functional abstractions have always a non-empty type, even if the body is ill-typed, but their application may have type `none()`. We had to take this into account in our typing rules.

The rest of the paper is organized as follows: Section 2 contains an informal introduction to success types. Section 3 contains some preliminaries about the language considered in this paper, its syntax and its semantics, covering the case of expressions with free variables. Sections 4 and 5 present the type system and its derivation rules, which are exemplified in Section 6. Correctness results are given in Section 7, and Section 8 concludes the paper.

2 An informal introduction to success types

In this section we give a general overview of success types, and highlight their particularities. Throughout this paper we use variables τ, τ_1, etc. to denote success types. Their syntax and

[1] To be more precise, this can only be ensured for terminating computations.
[2] Again, this strictly applies only to terminating computations.
[3] Well-typed expressions in Hindley-Milner systems, requires all their pieces to be also well-typed.

semantics will be formally introduced in Section 4, but for the purposes of this overview we shall say in advance that the types supported by Dialyzer include, among others, basic types (such as `integer()`, `number()`, `atom()`, etc.), singleton types (for instance, `0`, `foo` or `[]`), union types (written $\tau_1 \cup \ldots \cup \tau_n$), `any()` and `none()`. The type `any()` represents all the Erlang values, whereas `none()` represents the empty set, that is, the absence of values. Dialyzer also supports the type `maybe_improper_list(`τ_1`, `τ_2`)`, which contains those lists whose elements are of type τ_1 and their continuation (i.e. its "innermost tail") is of type τ_2. For example, the list `[1 | [2 | b]]` belongs to `maybe_improper_list(integer(), atom())`. If the continuation is an empty list, we shall write `[`τ`]` instead of `maybe_improper_list(`τ`, [])`.

As explained in Section 1, success types are overapproximations of the set of values that an expression may evaluate to. For instance, assume the following Erlang definition:

```
f(0) -> foo;
f(1) -> bar.
```

The function `f` accepts `(0 ∪ 1) → foo ∪ bar` as a success type, but `(integer()) → atom()` and `(any()) → any()` are valid success types as well. Although these three types overapproximate the semantics of `f`, the first one is more accurate than the others, and allows Dialyzer to report that an application such as `f(2)` fails to evaluate.

Sometimes Dialyzer infers success types that, although correct, are somewhat counterintuitive in comparison with standard Hindley-Milner types. For instance, assume the following function `nth` that, given a number `N` and a list `Xs`, returns the `N`-th element of `Xs`, or `false` if `N` is greater than the length of the list:

```
nth(_, [])       -> false;
nth(1, [X | Xs]) -> X;
nth(N, [_ | Xs]) -> nth(N - 1, Xs).
```

In a standard type setting one would expect a type such as `(number(), [any()]) → any()` for this function. However, this is not a success type, since it excludes the application `nth(foo, [])`, which is evaluated to `false`. Moreover, this type would demand `Xs` to be a proper list (i.e. with `[]` as a continuation), even when an application such as `nth(1, [a, [b | c]])` succeeds. A suitable success type for this function is `(any(), maybe_improper_list(any(), any())) → any()`. In fact, if we define a function `nth'` with the same clauses as `nth` but excluding the first one, this function will fail in case `N` is longer than the input list, as it is actually done in Erlang's standard library. As a consequence, we obtain the type `(number(), maybe_improper_list(any(), any())) → any()` for `nth'`, which rules out applications such as `nth'(foo, [])`.

In the current version of Dialyzer, a type specification may contain variables that relate the input and output of a function. For instance, given the `nth` function shown above, the specification `(any(), maybe_improper_list(`α`, any())) → `α` ∪ false` states that `nth` yields a value having the same type as the elements of the input list. From this specification we would expect Dialyzer to infer the type `b ∪ c ∪ false` for the expression `nth(2, [b | [c | []]])`, but unfortunately it infers `any()`. This is because Dialyzer does not directly use the polymorphic type given above, but a specific monomorphic instance: `(any(), maybe_improper_list(any(), any())) → any() ∪ false`, so the connection between input and output is lost. The aim of this paper is to devise a set of typing rules in order to obtain a more accurate type for an application such as `nth(2, [b | [c | []]])`. Moreover, with our typing rules we can also derive a polymorphic type for `nth`.

Success typings bring several particularities not occurring in standard Hindley-Milner systems. Firstly, every expression has at least one success type, which is `any()`. Secondly the notion of τ being a success type of an expression e, which will be formally defined in Section 4, is semantic rather than being directed by a set of rules. Therefore, no algorithm can, in general, compute the set of all success types of an expression. For example, assume the following expression:

$$e \equiv \texttt{case } b \texttt{ of true -> 1; false -> 2 end} \tag{1}$$

in which b is a complex expression that is always evaluated to `true`. In this case, `1` is a success type of e but, in order to infer this type, an algorithm needs to know whether b is always evaluated to `true`, which is an undecidable problem. Thirdly, there are some expressions lacking a minimal success type. For example, given the following function:

```
g() -> receive
          0 -> 0
          1 -> [g() | []]
       end
```

we can find an infinite strictly decreasing chain of success types:

$$\texttt{0} \cup \texttt{[any()]} \supseteq \texttt{0} \cup \texttt{[0} \cup \texttt{[any()]]} \supseteq \texttt{0} \cup \texttt{[0} \cup \texttt{[0} \cup \texttt{[any()]]]} \supseteq \ldots$$

The set of typing rules introduced in Section 5 allows one to obtain success types for a given expression. As we shall prove later, every type obtained by these rules is a success type, but not every success type of e can be derived by applying them, as the example in (1) shows. Moreover, every expression is well-typed according to our typing rules, since we can always derive the type `any()` for every expression. Therefore, in a strict sense, there are no ill-typed expressions in our system, although some of them can be inferred to have `none()` as a success type, which entails that their evaluation will always fail at runtime. Lastly, the notion of polymorphism is subtler in the context of success types than in Hindley-Milner type systems. In Hindley-Milner, any instance of a valid polymorphic type scheme for an expression is a valid type for that expression. For example, if $\forall.(\alpha) \to \alpha$ is a valid Hindley-Milner type for the identity function, then so are $(\texttt{bool()}) \to \texttt{bool()}$ and $(\texttt{int()}) \to \texttt{int()}$. This is not true when considering success types, since these two monomorphic success types are incompatible for the same expression (they correspond to disjoint function graphs). In fact, the first monomorphic type would forbid the application of the identity function to an integer, which is an expression that always succeeds.

3 Preliminaries

3.1 Language syntax

This work is focused on a subset of Core Erlang [4] —a simpler version of Erlang—shown in Figure 1. There are some differences between the subset chosen in this paper and Core Erlang, which are meant to simplify the typing rules without losing generality.

This subset has literals, variables, lists, tuples, lambda abstractions, **let** expressions to introduce new variables, **letrec** expressions to introduce new recursive functions, **case** expressions to branch the execution, **receive** expressions to branch the execution when a message is received, and function calls. The variable in the **after** clause of a **receive** expression can be an integer or the atom `'infinity'`. In this last case the **after** clause will never be reached.

$$
\begin{array}{rcl}
\mathit{var} & ::= & \textsc{VariableName} \\
\mathit{lit} & ::= & \textsc{Atom} \mid \textsc{Integer} \mid \textsc{Float} \mid \texttt{[]} \\
\mathit{pat} & ::= & \mathit{var} \mid \mathit{lit} \mid \texttt{[}\mathit{pat}_1\texttt{|}\mathit{pat}_2\texttt{]} \mid \texttt{\{}\overline{\mathit{pat}_i}^{\,n}\texttt{\}} \\
\mathit{cls} & ::= & \mathit{pat}\ \textbf{when}\ \mathit{exp}_1 \to \mathit{exp}_2 \\
\mathit{fun} & ::= & \textbf{fun}(\overline{\mathit{var}_i}^{\,n}) \to \mathit{exp}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{exp} & ::= & \mathit{var} \mid \mathit{lit} \mid \mathit{fun} \mid \texttt{[}\mathit{exp}_1\texttt{|}\mathit{exp}_2\texttt{]} \mid \texttt{\{}\overline{\mathit{exp}_i}^{\,n}\texttt{\}} \\
 & & \mid \mathit{var}(\overline{\mathit{var}_i}^{\,n}) \mid \textbf{let}\ \mathit{var} = \mathit{exp}_1\ \textbf{in}\ \mathit{exp}_2 \\
 & & \mid \textbf{letrec}\ \overline{\mathit{var}_i = \mathit{fun}_i}^{\,n}\ \textbf{in}\ \mathit{exp} \\
 & & \mid \textbf{case}\ \mathit{var}\ \textbf{of}\ \overline{\mathit{cls}_i}^{\,n}\ \textbf{end} \\
 & & \mid \textbf{receive}\ \overline{\mathit{cls}_i}^{\,n}\ \textbf{after}\ \mathit{var} \to \mathit{exp}
\end{array}
$$

Figure 1: Subset of the Core Erlang syntax

Exceptions are supported in Erlang, but the programming philosophy of the language discourages its use. Thus we choose to leave **try/catch** and exceptions as a future goal. In the chosen subset, we only allow variables in **case** discriminants and application parameters in order to simplify the typing rules.[4] This allows us to attach type information to the discriminant when typing the branches. We also assume that, in the context of function applications, the function being applied and the arguments are variables, so that their types can be stored in a typing environment when analysing a function application.

3.2 Semantics of language expressions

In previous work [14] the semantics of a closed expression is defined as a subset of **DVal**, where **DVal** represents all the possible values that can be reached with the language expressions. To represent functions inside **DVal** we use graphs, which are sets of tuples $((\overline{\mathit{args}}), \mathit{value})$ where a sequence of values *args* (the arguments) is related to a result value. Due to the nondeterministic nature of concurrent Erlang, a tuple *args* may be related to more than one result inside a function graph. In this sense, the semantics of a function is a mathematical relation. To represent data structures inside **DVal** we also use tuples $(\mathit{ctor}, \overline{\mathit{args}})$, where *ctor* is the constructor of the structure and *args* is a sequence of values taken by the constructor. The constructors we have in our language are:

- $\texttt{\{}\cdot^{n}\cdot\texttt{\}}$ an Erlang tuple with n elements whose values are *args*.
- `[_|_]` an Erlang list constructor, where the first value of $\overline{\mathit{args}}$ is the head value of the list, and the second is the tail. We also use the notation $(\texttt{[\_|\_]}, v_1, \ldots, v_{n-1}, v_n)$ to denote $n-1$ nested list constructors.

To extend these concepts to expressions with free variables we need to consider substitutions that give values to variables. A substitution θ is a total function $\mathbf{Var} \to \mathbf{DVal}$, where **Var** is the set of all variables. **Subst** denotes the set of all substitutions. The notation $[\,]$ is used to assign the default value 0 to all variables (any default value other than 0 would serve). The notation $[x_1/v_1, \ldots, x_n/v_n]$ is used to represent the substitution that assigns the value v_i to the variable x_i and 0 to the other variables.

The semantics $\mathcal{E}[\![e]\!]$ of an expression e is defined as a relation $\mathcal{E}[\![e]\!] \subseteq \mathbf{Subst} \times \mathbf{DVal}$. The idea is that if $(\theta, v) \in \mathcal{E}[\![e]\!]$ then v is one of the possible values to which $e\theta$ can be reduced. The complete definition of $\mathcal{E}[\![e]\!]$ is given in Figure 2.

[4] Using only variables in arguments is no loss of generality, because we can use (possibly nested) let-bindings to introduce non-variable arguments.

$$\mathcal{E}\,[\![c]\!] = \{(\theta, c) \mid \theta \in \mathbf{Subst}\} \quad \mathcal{E}\,[\![x]\!] = \{(\theta, \theta(x)) \mid \theta \in \mathbf{Subst}\} \quad \mathcal{E}\,[\![f(\overline{x_i}^n)]\!] = \left\{(\theta, v) \mid ((\overline{\theta(x_i)}^n), v) \in \theta(f)\right\}$$

$$\mathcal{E}\,[\![\{\overline{e_i}^n\}]\!] = \left\{\left(\theta, \left(\{\cdot^n\cdot\}, \overline{v_i}^n\right)\right) \mid \forall i \in \{1..n\} : (\theta, v_i) \in \mathcal{E}\,[\![e_i]\!]\right\}$$

$$\mathcal{E}\,[\![[e_1 \mid e_2]]\!] = \{(\theta, ([\_|\_], v_1, v_2)) \mid (\theta, v_1) \in \mathcal{E}\,[\![e_1]\!], (\theta, v_2) \in \mathcal{E}\,[\![e_2]\!]\}$$

$$\mathcal{E}\,[\![\mathbf{fun}(\overline{x_i}^n) \to e]\!] = \left\{\left(\theta, \left\{((\overline{v_i}^n), v) \mid (\theta[\overline{x_i/v_i}], v) \in \mathcal{E}\,[\![e]\!]\right\}\right) \mid \theta \in \mathbf{Subst}\right\}$$

$$\mathcal{E}\,[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!] = \{(\theta, v) \mid (\theta, v_1) \in \mathcal{E}\,[\![e_1]\!], (\theta[x_1/v_1], v) \in \mathcal{E}\,[\![e_2]\!]\}$$

$$\mathcal{E}\left[\!\left[\mathbf{case}\ x\ \mathbf{of}\ \overline{cls_i}^n\right]\!\right] = \bigcup_{i=1}^{n} \Big\{(\theta, v) \mid \overline{v_j}^m \in \mathbf{DVal}, (\theta[\overline{x_{ij}/v_j}], v) \in \mathcal{C}\,[\![cls_i]\!]_{\{\theta(x)\}}, \Big(\forall k < i. \forall \overline{v_j}'. \forall v'. (\theta[\overline{x_{kj}/v_j'}], v') \notin \mathcal{C}\,[\![cls_k]\!]_{\{\theta(x)\}}\Big)\Big\}$$

$$\text{where } \forall i \in \{1..n\} : cls_i = (p_i\ \mathbf{when}\ e_i \to e_i')\ \text{ and } vars(p_i) = \{\overline{x_{ij}}\}$$

$$\mathcal{E}\left[\!\left[\mathbf{receive}\ \overline{cls_i}^n\ \mathbf{after}\ e_t \to e\right]\!\right] = \bigcup_{i=1}^{n} \Big\{(\theta, v) \mid \overline{v_j}^m \in \mathbf{DVal}, (\theta[\overline{x_{ij}/v_i}], v) \in \mathcal{C}\,[\![cls_i]\!]_{\mathbf{DVal}}, \Big(\forall k < i. \forall \overline{v_j}'. \forall v'. (\theta[\overline{x_{kj}/v_j'}], v') \notin \mathcal{C}\,[\![cls_k]\!]_{\mathbf{DVal}}\Big), (\theta, v_t) \in \mathcal{E}\,[\![e_t]\!], v_t \in \texttt{integer()} \cup \{\mathit{infinity}\}\} \cup \{(\theta, v) \mid (\theta, v) \in \mathcal{E}\,[\![e]\!], (\theta, v_t) \in \mathcal{E}\,[\![e_t]\!], v_t \in \texttt{integer()}\}$$

$$\text{where } \forall i \in \{1..n\} : cls_i = (p_i\ \mathbf{when}\ e_i \to e_i')\ \text{ and } vars(p_i) = \{\overline{x_{ij}}\}$$

$$\mathcal{E}\,[\![\mathbf{letrec}\ \overline{x_i = e_i}^n\ \mathbf{in}\ e]\!] = \left\{(\theta, v) \mid (\overline{v_i}^n) = \mathit{lfp}\ F_\theta, (\theta[\overline{x_i/v_i}], v) \in \mathcal{E}\,[\![e]\!]\right\}$$

$$\text{where } F_\theta(\overline{v_i}^n) = (\overline{v_i'}^n) \text{ and } \forall k \in \{1..n\}.\, \{v_k'\} = \left\{v \mid (\theta[\overline{x_i/v_i}], v) \in \mathcal{E}\,[\![e_k]\!]\right\}$$

$$\mathcal{C}\,[\![p\ \mathbf{when}\ e_g \to e]\!]_V = \{(\theta, v) \mid (\forall v' \in V. (\theta, v') \in \mathcal{E}\,[\![p]\!]), (\theta, \mathit{true}) \in \mathcal{E}\,[\![e_g]\!], (\theta, v) \in \mathcal{E}\,[\![e]\!]\}$$

Figure 2: Denotational semantics of expressions

4 Type system

In this section we describe the syntax and semantics of the types that can be derived from the typing rules introduced in the next section. We assume the existence of a set $\mathbb{B}$ of *basic types* such as `integer()`, `atom()`, `number()`, etc. each one denoting a set of Erlang values. For each basic type $B \in \mathbb{B}$ the notation $\mathcal{B}\,[\![B]\!]$ represents the set of values denoted by this type. For instance, $\mathcal{B}\,[\![\texttt{integer()}]\!]$ includes the set of integer numbers, whereas $\mathcal{B}\,[\![\texttt{atom()}]\!]$ denotes the set of Erlang atoms (i.e. symbolic constants).

We also assume the existence of a set **TypeVar** of *type variables*, each of which are represented by α, β, etc. Type variables are used to obtain polymorphic types in our system and, depending on the context in which they appear, they can denote a single value or a set of values.

4.1 Syntax of polymorphic types

We denote by **Type** the set of types respectively generated by the following syntax rules:

$$\begin{aligned} \tau &::= \texttt{none()} \mid \texttt{any()} \mid B \mid v \mid \texttt{\{}\overline{\tau_i}^n\texttt{\}} \mid \tau_1 \cup \tau_2 \mid \texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)} \mid (\overline{\tau_i}^n) \xrightarrow{C} \tau \mid \alpha \mid \sigma \\ C &::= \{\tau_1 \subseteq \tau_1', \ldots, \tau_n \subseteq \tau_n'\} \\ \sigma &::= \forall \alpha_1 \subseteq \tau_1, \ldots, \alpha_n \subseteq \tau_n.\tau \end{aligned}$$

where $B \in \mathbb{B}$, $v \in \mathbf{DVal}$, and $\alpha \in \mathbf{TypeVar}$.

The type `none()` denotes the absence of values. If an expression has `none()` as a success type, then it does not evaluate to any value, that is, its evaluation will either fail or diverge. On the contrary, the type `any()` denotes the set **DVal** containing all values, so this type always overapproximates the set of values to which an expression is evaluated. In other words, `any()` is a success type of every expression. The type system also features singleton types, in the sense that for every value $v \in \mathbf{DVal}$ there is a type v denoting the set $\{v\}$.

The tuple type $\{\tau_1, \ldots, \tau_n\}$ denotes those tuples whose i-th component is contained within the type τ_i for each $i \in \{1..n\}$. The type `nelist(`τ_1`,`τ_2`)` represents those nonempty lists whose elements are within the type τ_1 and their continuations (i.e. their innermost tails) belong to the set denoted by the type τ_2. In Erlang we can make distinction between *proper* and *improper* lists. A list is proper if its innermost tail is the empty list, and it is improper otherwise. For instance, the expression `[1|[2|[3|[]]]]` evaluates to a proper list, whereas `[a|[b|[c|d]]]` does not, since its innermost tail (`d`) is not the empty list. Nonempty proper lists whose elements have type τ can be represented by the type `nelist(`τ`,[])`, where `[]` is the singleton type denoting the empty list. The proper list shown before is usually shortened to `[1,2,3]`.

The union type $\tau_1 \cup \tau_2$ denotes the set of values contained in τ_1, τ_2, or both. For instance, the type `nelist(`τ`,[])` $\cup$ `[]` represents all the (possibly empty) proper lists whose elements have type τ. In the following we use `[`τ`]` as a shorthand for this type. Note that the type `maybe_improper_list(`τ_1`, `τ_2`)` used in the examples of Section 2 is actually a shorthand for `nelist(`τ_1`,`τ_2`)` $\cup$ `[]`.

The type $(\tau_1, \ldots, \tau_n) \xrightarrow{C} \tau$ denotes the set of n-ary functions that accept values in $(\tau_1, \ldots, \tau_n)$ and yield a result in τ. The C lying above the arrow is a sequence of constraints on the type variables occurring in the functional type and/or the typing context. These constraints pose necessary conditions for the evaluation of the function described by this type. Their role will be detailed in Section 4.3.

A type variable α is a placeholder which may denote a single value or a set of values. If a type variable appears more than once in a type or type environment, all its occurrences will be "related" in some way that will be more accurately described later in this section. For example, according to the semantic definitions given later, it turns out that the type $\{\alpha, \alpha\}$ denotes the set of pairs whose components are the same, and the type $\{$`nelist(`α`,[])`$, \alpha\}$ represents those pairs made up of a nonempty list and a value such that the latter is contained within the list. In the following we use $ftv(\tau)$ to denote the set of free type variables occurring in τ.

A type scheme $\forall \alpha_1 \subseteq \tau_1, \ldots, \alpha_n \subseteq \tau_n . \tau$ denotes a polymorphic type τ where the $\alpha_1, \ldots, \alpha_n$ are bound type variables, each of which must satisfy the restriction $\alpha_i \subseteq \tau_i$, denoting that α_i is a subtype of τ_i. If τ_i is omitted, it is assumed to be `any()`.

4.2 Type instantiations

In order to figure out the set of values denoted by a given type τ (i.e. the semantics of τ) we have to address the case in which τ contains free type variables. The first step is to determine what these variables stand for. In a standard Hindley-Milner setting, type variables can be instantiated by types, so an instance of a polymorphic type is a substitution that maps type variables to types. Since a type denotes a set of values, in this paper we take a slightly more generic approach: a type variable is directly replaced by a set of values, rather than by a type. That is why define a *type instantiation* as a mapping from type variables to sets of values. We use the variables π, π_1, etc. to denote type instantiations, so we get $\pi : \mathbf{TypeVar} \to \mathcal{P}(\mathbf{DVal})$. We say that a type variable α is *instantiated* by π iff $\pi(\alpha) \neq \emptyset$. We denote by $dom\ \pi$ the set of

type variables instantiated by π.

We denote by $[\,]$ the instantiation π such that $dom\ \pi = \emptyset$ and by $[\overline{\alpha_i \mapsto V_i}^n]$ the instantiation in which α_i is instantiated to V_i for every $i \in \{1..n\}$ and the other variables are left uninstantiated. Moreover, the notation $\pi\backslash\{\alpha_1, \ldots, \alpha_n\}$ denotes the same instantiation as π but with the variables $\alpha_1, \ldots, \alpha_n$ uninstantiated.

An order relation $\subseteq$ is defined between type instantiations in a pointwise basis, i.e. $\pi_1 \subseteq \pi_2$ iff $\pi_1(\alpha) \subseteq \pi_2(\alpha)$ for every $\alpha \in \mathbf{TypeVar}$. We similarly define the union and intersection of type instantiations, respectively denoted by $\pi_1 \cup \pi_2$ and $\pi_1 \cap \pi_2$. When defining the semantics of a type, we shall need to check whether a type variable is instantiated to different non-disjoint sets in its several occurrences. Hence we say that two instantiations π_1 and π_2 are *compatible* iff for every variable α instantiated by both π_1 and π_2 it holds that $\pi_1(\alpha) \cap \pi_2(\alpha) \neq \emptyset$. Given this, the notation $\pi_1 \oplus \pi_2$ denotes the intersection of π_1 and π_2 whenever these type instantiations are compatible; otherwise the result of $\pi_1 \oplus \pi_2$ is undefined. More generally, the intersection $\pi_1 \oplus \ldots \oplus \pi_n$ is defined provided $\bigcap_{i \in \{1..n\}, \alpha \in dom\ \pi_i} \pi_i(\alpha) \neq \emptyset$ for every α.

4.3 Type environments

In a Hindley-Milner type system, a type environment contains the type of every variable in scope. The type environments introduced in this section take on the same role but, in addition, they contain information constraining the type variables. Therefore, we define a *type environment* Γ as a pair $\langle \gamma, C \rangle$, in which γ is a total function from variables to types and C is a set of constraints. Although γ is total, we assume that γ assigns a type different from `any()` to a finite subset of variables. Throughout the paper we use the following notation to represent type environments $[x_1 : \tau_1, \ldots, x_n : \tau_n \mid \tau'_1 \subseteq \tau''_1, \ldots, \tau'_m \subseteq \tau''_m]$, which denotes the environment $\Gamma = \langle \gamma, C \rangle$ such that $\gamma(x_i) = \tau_i$ for every $i \in \{1..n\}$, $\gamma(z) =$ `any()` for every other variable, and $C = \{\tau'_1 \subseteq \tau''_1, \ldots, \tau'_m \subseteq \tau''_m\}$. As a particular case, $[\,]$ denotes the environment mapping every variable to `any()` with an empty set of constraints, whereas $\bot$ will be used to describe the environment that maps one of its variables to `none()`[5]. If we want to refer to a specific component of an environment Γ, we use $\Gamma|_\gamma$ and $\Gamma|_C$ to denote the first and second components of Γ, respectively. For the sake of clarity, we use $\Gamma(x)$ to denote $\gamma(x)$. Given a type Γ and a set X of variables, the notation $\Gamma\backslash X$ stands for the environment that results from replacing the types of the variables of X by `any()`. For the sake of conciseness, in the case in which X is the singleton set $\{x\}$, we leave out the curly braces so as to get $\Gamma\backslash x$.

The role of type variables in environments is the same as in types: they denote relations between the different variables in scope. For example, according to the definition that will be given in Section 4.4, the environment $[X : \alpha, Y : \alpha]$ specifies that the values of X and Y must be equal, whereas the environment $[Xs :$ `nelist(`α`,[])`$, Z : \{\alpha,$ `int()`$\} \mid \alpha \subseteq$ `atom()`$]$ specifies that Xs must contain a nonempty list of atoms, and that the first component of the tuple contained in Z must be one of the elements of that list.

4.4 Semantics of polymorphic types

Having introduced all the concepts in the previous sections, our next step is to put all the pieces together and provide a semantics for types and environments. In principle, a type denotes a set of values, so we need a function $\mathcal{T}[\![\_]\!]$ that, given a type τ, it returns a set $\mathcal{T}[\![\tau]\!] \subseteq \mathbf{DVal}$ containing the values of the language abstracted by τ. However, types might contain type variables, so we need a type instantiation π that tells us the values denoted by each of them.

[5] We assume that this particular variable has been fixed in advance.

$$\mathcal{T}_\pi[\![\texttt{none()}]\!] = \emptyset \qquad \mathcal{T}_\pi[\![\tau_1 \cup \tau_2]\!] = \mathcal{T}_\pi[\![\tau_1]\!] \cup \mathcal{T}_\pi[\![\tau_2]\!]$$

$$\mathcal{T}_\pi[\![\texttt{any()}]\!] = \begin{cases} \mathbf{DVal} & \text{if } \pi = [\,] \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathcal{T}_\pi[\![B]\!] = \begin{cases} \mathcal{B}[\![B]\!] & \text{if } \pi = [\,] \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{T}_\pi[\![v]\!] = \begin{cases} \{v\} & \text{if } \pi = [\,] \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathcal{T}_\pi[\![\alpha]\!] = \begin{cases} \{v\} & \text{if } \pi = [\alpha \mapsto \{v\}] \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{T}_\pi[\![\{\overline{\tau_i}^n\}]\!] = \left\{ \left(\{\overset{n}{\cdots}\}, \overline{v_i}^n\right) \middle| \pi = \bigoplus_{i=1}^{n} \pi_i, \forall i \in \{1..n\}\,.v_i \in \mathcal{T}_{\pi_i}[\![\tau_i]\!] \right\}$$

$$\mathcal{T}_\pi[\![\texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)}]\!] = \left\{ ([\_|\_], \overline{v_i}^n, v') \middle| n \geq 1, \pi = \bigcup_{i=1}^{n} \pi_i \oplus \pi', \forall i \in \{1..n\}\,.v_i \in \mathcal{T}_{\pi_i}[\![\tau_1]\!], v' \in \mathcal{T}_{\pi'}[\![\tau_2]\!] \right\}$$

$$\mathcal{T}_\pi\left[\!\!\left[(\overline{\tau_i}^n) \xrightarrow{C} \tau\right]\!\!\right] = \left\{ f \middle| \pi = \bigcup_{w \in f} \pi_w, f \subseteq \left\{ ((\overline{v_i}^n), v) \middle| \pi_w \vDash C, \pi_w \restriction_{ftv(\overline{\tau_i}^n)} = \bigoplus_{i=1}^{n} \pi_i, \right.\right.$$
$$\left.\left. \forall i \in \{1..n\}\,.v_i \in \mathcal{T}_{\pi_i}[\![\tau_i]\!], v \in \mathcal{T}_{\pi'_w}[\![\tau]\!], \pi'_w \subseteq \pi_w \right\} \right\}$$

$$\mathcal{T}^{\pi}_{Env}[\![\Gamma]\!] = \left\{ \theta \in \mathbf{Subst} \middle| \pi \restriction_{ftv(\Gamma|_\gamma)} = \bigoplus_{x \in \mathbf{Var}} \pi_x, \forall x \in \mathbf{Var}.\theta(x) \in \mathcal{T}_{\pi_x}[\![\Gamma(x)]\!], \pi \vDash \Gamma|_C \right\}$$

$$\mathcal{S}_\pi\left[\!\!\left[\overline{\forall \alpha_i \subseteq \tau_i}^n.\tau\right]\!\!\right] = \left\{ v \middle| \forall i \in \{1..n\}\,.V_i \subseteq \mathbf{DVal}, \pi' = \pi\left[\overline{\alpha_i \mapsto V_i}^n\right], \pi' \vDash \left\{\overline{\alpha_i \subseteq \tau_i}^n\right\}, \right.$$
$$\left. \pi' \restriction_{ftv(\tau)} = \pi'', v \in \mathcal{T}_{\pi''}[\![\tau]\!] \right\}$$

Figure 3: Semantics of polymorphic types, type environments and type schemes

Therefore, instead of $\mathcal{T}[\![\tau]\!]$ we write $\mathcal{T}_\pi[\![\tau]\!]$ to denote the semantics of the type τ under a type instantiation π.

The notation $\pi \restriction_{\{\alpha_1,\ldots,\alpha_n\}}$ denotes that the type instantiation π is restricted to a set of type variables $\{\alpha_1, \ldots, \alpha_n\}$, this means any other type variables instantiated will be removed from π.

The definition of $\mathcal{T}_\pi[\![\tau]\!]$ is shown in Figure 3. Notice that in the cases `any()`, v, and B, we demand the type instantiation π to be empty in order to obtain a non-empty semantics, whereas in the case of a type variable we demand π to instantiate only this variable with a singleton set. In general, the definition of $\mathcal{T}_\pi[\![\tau]\!]$, will demand π to be a *minimal* instantiation that makes v belong to the semantics of τ. If τ does not contain type variables, such an instantiation will be empty. If we want a value v to belong to the semantics of a type variable α, then $[\alpha \mapsto \{v\}]$ is the minimal instantiation that will make this possible. The choice of having minimal instantiations throughout our semantics is justified by the case of tuple types, in which the type instantiation π is decomposed into several instantiations (one for each component), all of which must be compatible. Without the minimal instantiation requirement we would have, for instance, that $\{4, a\}$ belongs to $\{\alpha, \alpha\}$ under the instantiation $\pi = [\alpha \mapsto \{4, a\}]$, which would render type variables useless, since saying that $v \in \mathcal{T}_\pi[\![\{\alpha, \alpha\}]\!]$ for some π would be the same as saying that $v \in \mathcal{T}_\pi[\![\{\texttt{any()}, \texttt{any()}\}]\!]$. On the contrary, with the minimal instantiation requirement we are able to express the intended meaning of $\{\alpha, \alpha\}$, that is, whenever $\{v_1, v_2\} \in \mathcal{T}_\pi[\![\{\alpha, \alpha\}]\!]$, it must hold that $v_1 = v_2$ and $\pi = [\alpha \mapsto \{v_1\}]$.

The semantics of `nelist(`τ_1`,`τ_2`)` is the set of nonempty lists containing elements from τ_1 and a continuation in τ_2. In this case π is decomposed into a union $\bigcup_{i=1}^{n} \pi_i$ for the list elements

and π' for the continuation. Unlike the case of tuples, the different π_i need not be compatible. If they were forced to be compatible, the type `nelist(`α`,[])` would only contain those lists containing the same element repeated (e.g. $[v, v, \ldots, v]$), which is not the intended meaning. On the contrary, with our actual definition we get that $[v_1, \ldots, v_n] \in \mathcal{T}_\pi [\![\texttt{nelist(}\alpha\texttt{,[])}]\!]$ whenever $\pi = [\alpha \mapsto \{v_1, \ldots, v_n\}]$. In fact, one can prove that $\{[v_1, \ldots, v_n], v\} \in \mathcal{T}_\pi [\![\{\texttt{nelist(}\alpha\texttt{,[])}, \alpha\}]\!]$ implies $v = v_i$ for some $i \in \{1..n\}$.

In the case of functional types $(\overline{\tau_i}^n) \xrightarrow{C} \tau$, we obtain the set of functions f that map arguments from τ_i to values from τ. For each function f we must be able to decompose π into several π_w, one for each tuple of the graph of f. Each π_w is decomposed into the instantiations $\pi_1, \ldots, \pi_n$ for the arguments (in a similar way to tuple types), and the instantiation of the result must be a subset of this π_w. The latter restriction tries to capture the notion of parametricity as in Reynold's abstraction theorem [19] and Wadler's free theorems [24]. For instance, we can prove that if f belongs to the semantics of $\alpha \to \alpha$ then f must be a subset of the identity function, and that if g belongs to $[\alpha] \to \alpha \cup \mathit{false}$, then the result of $g(Xs)$ must be either one of the elements of the list Xs or the atom *false*. If a variable does not get instantiated in the left-hand side of a functional type, it must not get instantiated in the right-hand side. Hence $g([\,])$ must be evaluated to *false* in the previous example.

In the definition of the semantics of functional types we demand that the instantiation π_w corresponding to each tuple w satisfies the set C of constraints specified in the arrow of the type. This is denoted by $\pi \vDash C$. In order to give a proper notion of satisfiability we have to take the semantics of the types in both sides of each equation into account, but dropping the minimality requirement on type instantiations. Given a type τ and an instantiation we define $\mathcal{T}[\![\tau\pi]\!]$ as follows: $\mathcal{T}[\![\tau\pi]\!] = \bigcup_{\pi' \subseteq \pi} \mathcal{T}_{\pi'}[\![\tau]\!]$. Therefore, we say that $\pi \vDash \{\tau_1 \subseteq \tau'_1, \ldots, \tau_n \subseteq \tau'_n\}$ iff $\mathcal{T}[\![\{\tau_1, \ldots, \tau_n\}\,\pi]\!] \subseteq \mathcal{T}[\![\{\tau'_1, \ldots, \tau'_n\}\,\pi]\!]$. As an example, we get that $[\alpha \mapsto \{1, 2\}]$ satisfies $\{\alpha \subseteq \texttt{int()}\}$, but $[\alpha \mapsto \{1, 2, a\}]$ does not.

In order to motivate the existence of sets of constraints above the arrows of functional types, let us consider the following expression $e = \mathbf{fun}(X) \to \mathbf{fun}(Y) \to \{X, X + Y\}$. A success type of e would be `number()` $\to$ `number()` $\to \{$`number(),number()`$\}$. However, this type does not capture the fact that the first component of the result is the parameter X bound in the outer λ-abstraction. A more accurate success type of e would be $\forall\alpha.\alpha \to$ `number()` $\to \{\alpha,$ `number()`$\}$. Moreover, since the addition operator only succeeds when applied to numeric arguments, we would be tempted to give the following type for e: $\forall\alpha \subseteq$ `number()`$.\alpha \to$ `number()` $\to \{\alpha,$ `number()`$\}$. However, this is *not* a success type for e, as this type forbids any instantiation of α with non-numeric arguments, while we could, for example, apply e to a list and successfully obtain a closure. The execution of this closure would fail when applied to any value, but the application of the outer λ-abstraction has been evaluated successfully, even if it yields a function that always fails when applied. If we want to convey that α must be instantiated by a numeric type only when applying the inner abstraction, we could use $\forall\alpha.\alpha \to$ (`number()`) $\xrightarrow{\alpha \subseteq \texttt{number()}}$ `number()`, which is a success type of e.

The semantics of an environment Γ (written $\mathcal{T}^\pi_{Env}[\![\Gamma]\!]$) is the set of substitutions θ such that one can decompose the π into several instantiations π_x (one for each variable x), so that $\theta(x) \in \mathcal{T}_{\pi_x}[\![\Gamma(x)]\!]$ for every x and π satisfies the constraints in Γ. Otherwise $\mathcal{T}^\pi_{Env}[\![\Gamma]\!]$ is empty. This definition is formalized in Figure 3. If we are only concerned about the existence of a π rather than the π itself, we can leave out the instantiation from $\mathcal{T}^\pi_{Env}[\![\_]\!]$:

$$\mathcal{T}_{Env}[\![\Gamma]\!] \quad = \quad \{\theta \in \mathbf{Subst} \mid \theta \in \mathcal{T}^\pi_{Env}[\![\Gamma]\!] \text{ for some } \pi\}$$

The semantic definition of type environments induces a pre-order between them: we say that

$\Gamma_1 \subseteq \Gamma_2$ iff $\mathcal{T}_{Env}\,[\![\Gamma_1]\!] \subseteq \mathcal{T}_{Env}\,[\![\Gamma_2]\!]$, and we use $\Gamma_1 \equiv \Gamma_2$ to denote the conjunction of $\Gamma_1 \subseteq \Gamma_2$ and $\Gamma_2 \subseteq \Gamma_1$. Although the pair $(\Gamma, \subseteq)$ is not a complete lattice, we can prove the existence of an operator $\sqcap$ such that $\mathcal{T}_{Env}\,[\![\Gamma_1 \sqcap \Gamma_2]\!] \supseteq \mathcal{T}_{Env}\,[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}\,[\![\Gamma_2]\!]$ for every $\Gamma_1, \Gamma_2 \in \mathbf{Env}$. More details on this operator will be given in Section 7.1.

4.5 A generalization of success types

According to previous work [14] a type τ is said to be a success type of e if the set of values obtained from $\mathcal{E}\,[\![e]\!]$ (disregarding substitutions) is a subset of $\mathcal{T}\,[\![\tau]\!]$. However, this definition only applies to closed expressions (i.e. without free variables) and monomorphic types. It does not pose any constraints on the free variables of e. For instance, we can say that `number()` is a success type for $X + 1$, since the values of $\mathcal{E}\,[\![X + 1]\!]$ are a subset of $\mathcal{T}\,[\![\texttt{number()}]\!]$, but we also want to convey that $\theta(X)$ must be a number in order to evaluate this expression successfully. The latter restriction can be expressed as $\theta \in \mathcal{T}_{Env}\,[\![[X : \texttt{number()}]]\!]$, so in this case there is a pair $\langle \texttt{number()}; [X : \texttt{number()}] \rangle$ which overapproximates the semantics of $X + 1$. In general, we can approximate the semantics of expressions by pairs $\langle \tau; \Gamma \rangle$, where τ is a type and Γ is a type environment. Given that τ and Γ might have type variables in common, we can relate the type variables in τ to those in Γ just in the same way as one relates the type variables of the right-hand side of a functional type to those in the left-hand side: by demanding that the type instantiation used in τ is a subset of the type instantiation used in Γ. Therefore, the semantics of a pair $\langle \tau; \Gamma \rangle$ is defined as follows:

$$\begin{array}{rcl} \mathcal{T}_\pi\,[\![\tau; \Gamma]\!] & = & \{(\theta, v) \mid \theta \in \mathcal{T}^{\pi}_{Env}\,[\![\Gamma]\!]\,, v \in \mathcal{T}_{\pi'}\,[\![\tau]\!]\,, \pi' \subseteq \pi\} \\ \mathcal{T}\,[\![\tau; \Gamma]\!] & = & \{(\theta, v) \mid (\theta, v) \in \mathcal{T}_\pi\,[\![\tau; \Gamma]\!] \text{ for some } \pi\} \end{array}$$

so we say that the pair $\langle \tau; \Gamma \rangle$ overapproximates the expression e whenever $\mathcal{E}\,[\![e]\!] \subseteq \mathcal{T}\,[\![\tau; \Gamma]\!]$.

The semantic definition of pairs $\langle \tau; \Gamma \rangle$ also induces a pre-order between them: $\langle \tau_1; \Gamma_1 \rangle \subseteq \langle \tau_2; \Gamma_2 \rangle$ iff $\mathcal{T}\,[\![\tau_1; \Gamma_1]\!] \subseteq \mathcal{T}\,[\![\tau_2; \Gamma_2]\!]$. Similarly to type environments, we can also prove the existence of an operator $\sqcap$ on pairs such that $\mathcal{T}\,[\![\langle \tau_1; \Gamma_1 \rangle \sqcap \langle \tau_2; \Gamma_2 \rangle]\!] \supseteq \mathcal{T}\,[\![\tau_1; \Gamma_1]\!] \cap \mathcal{T}\,[\![\tau_2; \Gamma_2]\!]$ for any τ_1, Γ_1, τ_2, and Γ_2.

5 Typing judgements

The definition of success types given in [9] states that $\tau_1 \to \tau_2$ is a success type of the function f if and only if, for all $v, v' \in \mathbf{DVal}$, such that $f(v)$ evaluates to v', then v is contained in τ_1 and v' is contained in τ_2. In other words, if the graph of the function denoted by f is contained within the semantics of $\tau_1 \to \tau_2$.

With the type rules shown in this section we shall obtain, for each expression e, a type and an environment, the latter expressing necessary conditions for the evaluation of e. However, it will be convenient to add to our judgements another environment which will reflect some (already known) assumptions on the free variables of the expression e. Therefore, our judgements will be of the form $\Gamma \vdash e : \tau, \Gamma'$, with the following meaning: assuming that the values of the free variables in e are contained within their corresponding types in Γ, if e is evaluated to a value v, then the values of the free variables in e are contained within the types in Γ' for some π and v is of type τ for $\pi' \subseteq \pi$. More precisely, if $\theta \in \mathcal{T}_{Env}\,[\![\Gamma]\!]$ and $(\theta, v) \in \mathcal{E}\,[\![e]\!]$ then $\theta \in \mathcal{T}^{\pi}_{Env}\,[\![\Gamma']\!]$ for some π and $v \in \mathcal{T}_{\pi'}\,[\![\tau]\!]$ for some $\pi' \subseteq \pi$. This can be expressed more succinctly as $\mathcal{E}\,[\![e]\!] \upharpoonright_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}\,[\![\tau; \Gamma']\!]$. In the following we use the terms *assumption environment* and *final environment* to refer to Γ and Γ' respectively.

$$\frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \Gamma_1' \subseteq \Gamma_1}{\Gamma_1' \vdash e : \tau, \Gamma_2} \text{ [SUB-1]} \qquad \frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \langle \tau; \Gamma_2 \rangle \subseteq \langle \tau'; \Gamma_2' \rangle}{\Gamma_1 \vdash e : \tau', \Gamma_2'} \text{ [SUB-2]}$$

$$\frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \Gamma_2 \vdash e : \tau', \Gamma_3}{\Gamma_1 \vdash e : \tau', \Gamma_3} \text{ [TRANS]} \qquad \frac{}{\Gamma \vdash c : c, \Gamma} \text{ [CONS]} \qquad \frac{}{\Gamma \vdash x : \Gamma(x), \Gamma} \text{ [VAR]}$$

$$\frac{\Gamma \vdash e_i : \tau_i, \Gamma_i' \qquad \langle \tau_1; \Gamma_1' \rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n' \rangle = \langle \tau; \Gamma' \rangle}{\Gamma \vdash \{\overline{e_i}^n\} : \tau, \Gamma'} \text{ [TUPLE]} \qquad \frac{\Gamma \vdash e_1 : \tau_1, \Gamma_1 \qquad \Gamma \vdash e_2 : \tau_2, \Gamma_2 \qquad \langle \tau_1; \Gamma_1 \rangle \otimes \langle \tau_2; \Gamma_2 \rangle = \langle \{\tau_1, \tau_2\}; \Gamma' \rangle}{\Gamma \vdash [e_1 \mid e_2] : \texttt{nelist}(\tau_1, \tau_2), \Gamma'} \text{ [LIST]}$$

$$\frac{\Gamma \sqcap [f : \left(\overline{\texttt{any()}}^n\right) \to \texttt{any()}] \subseteq \Gamma_0 \qquad \Gamma_0(f) = \forall \overline{\alpha_j \subseteq \texttt{any()}}^m . (\overline{\tau_i}^n) \xrightarrow{C} \tau' \qquad \langle \tau'; [\overline{x_i : \tau_i}^n \mid C] \rangle \sqcap \langle \texttt{any()}; \Gamma_0 \rangle \subseteq \langle \tau; \Gamma' \rangle}{\Gamma \vdash f(\overline{x_i}^n) : \tau, \Gamma'} \text{ [APP-1]}$$

$$\frac{\Gamma \vdash e : \tau, \Gamma' \qquad \Gamma' = [\overline{x_i : \tau_i}^n, \overline{y_i : \beta_i}^m \mid C] \qquad ftv(\overline{\tau_i}) \cap \{\overline{\beta_i}\} = \emptyset \qquad \{\overline{\alpha_i}\} = ftv(\Gamma') \setminus \{\overline{\beta_i}\} \qquad \left\langle \left(\forall \overline{\alpha_i}. (\overline{\tau_i}^n) \xrightarrow{C} \tau\right), [\overline{y_i : \beta_i}^m] \right\rangle \sqcap \langle \texttt{any()}; \Gamma \rangle \subseteq \langle \tau', \Gamma'' \rangle}{\Gamma \vdash \textbf{fun}(\overline{x_i}^n) \to e : \tau', \Gamma''} \text{ [ABS]} \qquad \frac{\Gamma \sqcap [f : \left(\overline{\texttt{any()}}^n\right) \to \texttt{any()}] \equiv \bot}{\Gamma \vdash f(\overline{x_i}^n) : \texttt{none()}, \bot} \text{ [APP-2]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma_1 \qquad \Gamma_1[x : \tau_1[\overline{\alpha_i / \alpha_i'}] \mid \overline{\alpha_i' \subseteq \alpha_i}] \vdash e_2 : \tau_2, \Gamma_2 \qquad \{\overline{\alpha_i}\} = ftv(\tau_1) \qquad \{\overline{\alpha_i}'\} \cap (ftv(\tau_1) \cup ftv(\Gamma_1)) = \emptyset}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2, \Gamma_2 \backslash x} \text{ [LET]} \qquad \frac{cls_i = (p_i \textbf{ when } e_i' \to e_i'') \qquad \Gamma \Vdash_{\{x\}} cls_i : \tau_i, \Gamma_i \qquad \langle \tau_i; \Gamma_i \backslash vars(p_i) \rangle \subseteq \langle \tau; \Gamma' \rangle}{\Gamma \vdash \textbf{case } x \textbf{ of } \overline{cls_i}^n : \tau, \Gamma'} \text{ [CASE]}$$

$$\frac{cls_i = (p_i \textbf{ when } e_i' \to e_i'') \qquad \Gamma \sqcap [x_t : \texttt{integer()} \cup \texttt{'infinity'}] \Vdash_\emptyset cls_i : \tau_i, \Gamma_i \qquad \Gamma \sqcap [x_t : \texttt{integer()}] \vdash e : \tau, \Gamma' \qquad \langle \tau_i; \Gamma_i \backslash vars(p_i) \rangle \subseteq \langle \tau; \Gamma' \rangle}{\Gamma \vdash \textbf{receive } \overline{cls_i}^n \textbf{ after } x_t \to e : \tau, \Gamma'} \text{ [RECEIVE]}$$

$$\frac{\Gamma_0 \equiv \Gamma \sqcap [\overline{x_j : \tau_j}^n] \qquad \Gamma_0 \vdash f_i : \tau_i, \Gamma_0 \qquad \Gamma_0 \vdash e : \tau, \Gamma'}{\Gamma \vdash \textbf{letrec } \overline{x_i = f_i}^n \textbf{ in } e : \tau, \Gamma' \setminus \{\overline{x_j}^n\}} \text{ [LREC]}$$

$$\frac{\Gamma \vdash p : \tau_p, \Gamma_p \qquad \Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g \qquad \Gamma_g \sqcap [\texttt{'true'} \subseteq \tau_g] \vdash e : \tau, \Gamma'}{\Gamma \Vdash_X p \textbf{ when } e_g \to e : \tau, \Gamma'} \text{ [CLS]} \qquad \frac{\Gamma_1 \Vdash_X cls : \tau, \Gamma_2 \qquad \Gamma_2 \Vdash_X cls : \tau', \Gamma_3}{\Gamma_1 \Vdash_X cls : \tau', \Gamma_3} \text{ [CLS-TRANS]}$$

Figure 4: Typing rules for expressions and clauses.

The typing rules are shown in Figure 4. The rule [SUB-1] specifies that we can replace the assumption environment Γ_1 by a stronger (i.e. more restrictive) one, whereas rule [SUB-2] allow us to weaken the type τ and the final environment Γ_2 accordingly.

The [TRANS] rule specifies that, whenever we have a judgement $\Gamma_1 \vdash e : \tau, \Gamma_2$ we can re-evaluate the type of e, this time under the assumptions given in Γ_2. This re-evaluation might allow us to further refine the type of e.

The [CONS] and [VAR] rules specify that the final environment poses no further constraints besides those in the assumption environment, whereas [TUPLE] and [LIST] merge the types and final environments of each subexpression with the operator $\otimes$. This operator, when applied to a sequence of pairs $\langle \tau_1; \Gamma_1 \rangle, \ldots, \langle \tau_n; \Gamma_n \rangle$ joins all the τ_i into a tuple type while applying the $\sqcap$ operator to the Γ_i, since a substitution θ must belong to the semantics of every Γ_i in order to evaluate the whole expression under θ. Although a concrete definition of $\otimes$ will be given later, for now it is enough to assume that this operator satisfies the following condition for every

substitution θ and values $v_1, \ldots, v_n$:

$$\forall i \in \{1..n\}.(\theta, v_i) \in \mathcal{T}[\![\tau_i; \Gamma_i]\!] \Longrightarrow (\theta, (\{\overset{n}{\cdot\cdot\cdot}\}, \overline{v_i}^n)) \in \mathcal{T}[\![\langle\tau_1; \Gamma_1\rangle \otimes \ldots \otimes \langle\tau_n; \Gamma_n\rangle]\!]$$

With respect to the [ABS] rule, the final environment is the same as the assumption environment, since the evaluation of a λ-abstraction always succeeds. We use the type variables $\overline{\beta_i}$ to denote the types of the free variables in the λ-abstraction. The functional type has to be generalized with the type variables $\overline{\alpha_i}$ appearing in Γ' except those of $\overline{\beta_i}$, since the latter relate the functional type to the context in which the λ-abstraction appears. The constraints C of final environment Γ' are only relevant when executing the function, so these are placed above the arrow in the functional type.

We have two rules for function applications: [APP-1] only makes sense when the type assumed for f is compatible with a functional type, whereas [APP-2] specifies that the evaluation of the expression will fail otherwise. In the first case, the final environment demands the values passed as arguments to be of the corresponding types $\tau_1, \ldots, \tau_n$, taking the constraints C above the arrow into account.

In the [LET] rule we obtain a pair $\langle\tau_1; \Gamma_1\rangle$ for the bound expression. According to the semantics of this pair (see Section 4.5) the instantiation of the free variables in τ_1 must be a subset of the instantiation of the same variables in Γ_1. We reflect this condition by introducing a fresh variable α'_i for each type variable in α_i in τ_1 and specifying the constraint $\alpha'_i \subseteq \alpha_i$ in the initial environment that will be used to analyse the main expression e_2.

In order to derive a type for a **case** or a **receive** expression, we have to derive one for each of its clauses. With the rules [CLS] and [CLS-TRANS] rules we obtain judgements of the form $\Gamma \Vdash_X cls : \tau, \Gamma'$, where X may be a singleton set (containing the discriminant variable of a **case** expression) or an empty set (in the case of **receive** expressions). If X is a singleton set $\{x\}$ the notation $[X : \tau]$ abbreviates $[x : \tau]$, whereas if X is empty, this notation abbreviates the empty environment. The rule [CLS] handles those cases in which the type of the discriminant is compatible with the type of the pattern and the type of the guard contains the atom *true* to evaluate the body expression. Having derived the typing judgements relative to every clause in a **case** or in a **receive**, the rule [CASE] takes the type and final environment of each clause and removes the pattern variables, since these are no longer free. The rule [RECEIVE], in order to evaluate the clauses, demands the variable x_t to have a type inside `integer()` $\cup$ `'infinity'`, but, in order to evaluate the body of the **after** expression, the variable x_t must be a subtype of `integer()`.

Our first result states that we can always find a type derivation for a given expression:

Proposition 1. *Given any expression e and initial environment Γ, there exist τ and Γ' such that $\Gamma \vdash e : \tau, \Gamma'$. In particular, $\Gamma \vdash e :$* `any()`*, [ ]*

Proof. Straightforward, by inspection of the typing rules. Side-conditions involving the inclusion relation $\subseteq$ between environments or pairs $\langle\tau; \Gamma\rangle$ can always be satisfied by choosing [ ] and $\langle$`any()`$;$[ ]$\rangle$ respectively on the right-hand side of these conditions. If the remaining side condition of [APP-1] does not hold, then the rule [APP-2] can be applied. With respect to [LET] rule, the side condition $\{\overline{\alpha_i}'\} \cap (ftv(\tau_1) \cup ftv(\Gamma_1)) = \emptyset$ can always be satisfied by forcing $\overline{\alpha_i}'$ to be a vector of zero length.

Once we prove have derived the judgement $\Gamma \vdash e : \tau, \Gamma'$ for some τ and Γ', and given that $\langle\tau; \Gamma'\rangle \subseteq \langle$`any()`$;$[ ]$\rangle$, we can use rule [SUB-2] in order to obtain $\Gamma \vdash e :$ `any()`, [ ]. □

6 Examples

6.1 A simple function

In this example[6] we use the symbol + as the variable that represents the function `'erlang':'+'`, whose type is $\tau_+ = (\texttt{number()}, \texttt{number()}) \to \texttt{number()}$ and it will be given in the initial environment $\Gamma_0 = [+ : \tau_+]$. The code is the following:

$$\textbf{fun}(A) \to \textbf{let } B = 1 \textbf{ in } \{A, A + B\}$$

The type $\overline{\forall}.\,(\alpha) \xrightarrow{\alpha \subseteq \texttt{number()}} \{\alpha, \texttt{number()}\}$ is obtained with the following derivation:

$$\frac{\begin{array}{c}\Gamma_1 \sqcap [+ : (\texttt{any()}, \texttt{any()}) \to \texttt{any()}] \subseteq \Gamma_1 \\ \langle \texttt{number()}, [A : \texttt{number()}, B : \texttt{number()}] \rangle \sqcap \langle \texttt{any()}, \Gamma_1 \rangle \subseteq \langle \texttt{number()}, \Gamma_3 \rangle \\ \Gamma_3 = [+ : \alpha_+, A : \alpha_A, B : \alpha_B \mid \alpha_A \subseteq \texttt{number()}, \alpha_B \subseteq 1, \alpha_+ \subseteq \tau_+]\end{array}}{\Gamma_1 \vdash A + B : \texttt{number()}, \Gamma_3} \text{ [APP-1]}$$

$$\frac{\Gamma_2 \vdash A : \alpha_A, \Gamma_2 \qquad \Gamma_1 \subseteq \Gamma_2 \qquad \Gamma_2 = [+ : \tau_+, A : \alpha_A, B : 1]}{\Gamma_1 \vdash A : \alpha_A, \Gamma_2} \text{ [SUB-1]} \qquad \frac{\Gamma_1 \vdash A : \alpha_A, \Gamma_2 \qquad \Gamma_1 \vdash A + B : \texttt{number()}, \Gamma_3}{\Gamma_1 \vdash \{A, A + B\} : \{\alpha_A, \texttt{number()}\}, \Gamma_3} \text{ [TUPLE]}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash 1 : 1, \Gamma_0 \qquad \Gamma_1 \vdash \{A, A + B\} : \{\alpha_A, \texttt{number()}\}, \Gamma_3 \qquad \Gamma_1 = [+ : \tau_+, B : 1] \\ \Gamma_4 = \Gamma_3 \backslash B = [+ : \alpha_+, A : \alpha_A \mid \alpha_A \subseteq \texttt{number()}, \alpha_B \subseteq 1, \alpha_+ \subseteq \tau_+]\end{array}}{\Gamma_0 \vdash \textbf{let } B = 1 \textbf{ in } \{A, A + B\} : \{\alpha_A, \texttt{number()}\}, \Gamma_4} \text{ [LET]}$$

$$\frac{\Gamma_0 \vdash \textbf{let } B = 1 \textbf{ in } \cdots : \{\alpha_A, \texttt{number()}\}, \Gamma_4}{\Gamma_0 \vdash \textbf{fun}(A) \to \cdots : \overline{\forall}.\,(\alpha) \xrightarrow{\alpha \subseteq \texttt{number()}} \{\alpha, \texttt{number()}\}, \Gamma_0} \text{ [ABS]}$$

This type is semantically different from, for instance, the type $\overline{\forall}.\,(\alpha) \xrightarrow{\alpha \subseteq \texttt{number()}} \{\alpha, \alpha\}$ which is not a success type for this example, since this type only accepts to return tuples whose both components are equal to the input value. On the other hand, the type $(\texttt{number()}) \to \texttt{\{number(),number()\}}$ is a success type for the expression, since its semantics contains the semantics of the type obtained in the derivation. But this larger type is less refined and it has no polymorphism, due to the lack of type variables to connect the parameters of the function with the type of the result.

6.2 Functions on lists

In this section we will show the types obtained for some functions involving lists, such as *Foldl*, *Reverse*, *Filter*, and *Nth*; some of them are higher-order functions. The code of these functions and a full derivation of an additional example (*Map*) can be found at [13].

The type $\forall\alpha.\forall\beta.\forall\gamma.\,((\alpha, \beta) \to \gamma, \beta', [\alpha']) \xrightarrow{\alpha' \subseteq \alpha} \gamma \cup \beta'$ is obtained for *Foldl* from a derivation with our type system, where the first parameter F is the function that mixes the received accumulated value in the second parameter A with the head of the third parameter L. The type $\forall\alpha.\forall\beta.\forall\gamma.\,((\alpha, \gamma) \to \beta, \gamma', [\alpha']) \xrightarrow{\alpha' \subseteq \alpha, \gamma' \subseteq \gamma} \beta$ is not a success type of this function, because when L is an empty list the parameter A need not be related to the type of F's result, since the mixer function is not going to be called. For this reason, when L is an empty list, the result obtained from the [CLS] rule is the pair $\langle \beta_1'; [F : (\alpha_1, \beta_1) \to \gamma_1, A : \beta_1', L : []] \rangle$ while in the clause handling nonempty lists, we obtain $\langle \gamma_2; [F : (\alpha_2, \beta_2) \to \gamma_2, A : \beta_2', L : \texttt{nelist(}\alpha_2'\texttt{,[])} \mid \alpha_2' \subseteq \alpha_2, \beta_2' \subseteq \beta_2] \rangle$. When these pairs are merged in the final result of the [CASE] rule, the connection between the variable A and the function F is lost since the union of β_1' and β_2' is `any()`.

[6] To keep the examples shorter, the use of [CONS] and [VAR] is not shown because their use is trivial.

On the other hand, L does not collapse to `any()` since the union of `[]` and `nelist(`α'_2`,[])` is `[`α'_2`]`, and the constraint $\alpha'_2 \subseteq \alpha_2$ is not lost.

The type $\forall\alpha.\,(\texttt{[}\alpha\texttt{]}) \to \texttt{[}\alpha\texttt{]}$ is obtained for *Reverse* from a derivation with our type system, where the only parameter L is a list. The list reversal is done through an auxiliary function $R2$ with two parameters, where the first parameter LS is a list and the second parameter A is an accumulator. Since the type we obtain for $R2$ is $\forall\alpha.\beta.\,(\texttt{[}\alpha\texttt{]},\beta) \to \beta \cup \texttt{nelist(}\alpha\texttt{,}\beta\texttt{)}$, a derivation for $R2(L, \texttt{[]})$ yields the following pair as result: $\langle \beta \cup \texttt{nelist(}\alpha\texttt{,}\beta\texttt{)}; [L : \alpha_L, K : \alpha_K \mid \alpha_L \subseteq \texttt{[}\alpha\texttt{]}, \alpha_K \subseteq \texttt{[]}, \alpha_K \subseteq \beta]\rangle$. For any element belonging to this pair, the corresponding instantiation will map both β and α_k to singleton sets, so β must be mapped to the empty list according to the constraints. Therefore, we can simplify the result of *Reverse* to `[`α`]`.

The type $\forall\alpha.\forall\beta.\,((\alpha) \to \texttt{any()}, \texttt{[}\beta\texttt{]}) \xrightarrow{\beta\subseteq\alpha} \texttt{[}\beta\texttt{]}$ is obtained for *Filter* from a derivation with our type system, where the first parameter P is a predicate function and the second parameter a list L. The type $\forall\alpha.\forall\beta.\,((\alpha) \to \texttt{bool()}, \texttt{[}\beta\texttt{]}) \xrightarrow{\beta\subseteq\alpha} \texttt{[}\beta\texttt{]}$ cannot be a success type for this function because P is not called when L is an empty list. When L is not an empty list, we know that the type of P is $(\alpha) \to \alpha'$ and variable B ends up with the following restrictions: $\alpha_B \subseteq \alpha'$ and $\alpha_B \subseteq$ `'true'` $\cup$ `'false'`. Since α' can be instantiated to a single value, the output of the predicate function must contain the `bool()` type to succeed in this path of execution. If L is empty, there is no restriction for α' and for that reason this information is discarded when both clauses are joined in the [CASE] rule.

The type $\forall\alpha.\,(\texttt{number()}, \texttt{nelist(}\alpha\texttt{,any())}) \to \alpha$ is obtained for *Nth*. Since the function might return even when the input list is not traversed completely, we cannot ensure that the continuation of this list is [ ]. This is why we obtain `nelist(`α`,any())`.

7 Correctness

In this section we introduce the theorem that states that the types derived by the set of rules are success types. Detailed proofs can be found at [13].

7.1 Greatest lower bounds on type environments

In Section 4.3 we assumed the existence of an operator $\sqcap$ on typing environments such that, for every Γ_1 and Γ_2, $\mathcal{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!] \supseteq \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$. In this subsection we shall prove this assumption by defining an operator that satisfies this property.

Firstly we are going to define a restricted notion of greatest lower bound which is only applicable to a pair of compatible environments. We say that Γ_1 and Γ_2 are *compatible* if they have the following form

$$\Gamma_1 = [x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid C_1] \qquad \Gamma_2 = [x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid C_2]$$

and the only free variables common to Γ_1 and Γ_2 are $\{\alpha_1, \ldots, \alpha_n\}$. Given a pair of compatible environments, we define $GLB(\Gamma_1, \Gamma_2)$ as the environment $[x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid C_1 \cup C_2]$, and we prove that GLB satisfies our desired property:

Lemma 1. *Let Γ_1 and Γ_2 be two compatible environments. It holds that $\mathcal{T}_{Env}[\![GLB(\Gamma_1, \Gamma_2)]\!] = \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$.*

In order to extend this notion to non-compatible environments, we need to transform the latter into compatible environments. This process is called *normalization*. Given an environment Γ and a finite set $X = \{x_1, \ldots, x_n\}$ containing all the variables x such that $\Gamma(x) \neq \texttt{any()}$, we define the *normalization* of Γ with respect to X (denoted by $norm(X, \Gamma)$) as follows:

$$norm(X,\Gamma) = [\overline{x_i : \alpha_i}^n \mid \{\alpha_1 \subseteq \Gamma(x_1), \ldots, \alpha_n \subseteq \Gamma(x_n)\} \cup \Gamma|_C]$$

where $\alpha_1, \ldots, \alpha_n$ are fresh type variables not occurring in Γ. It is easy to show that the normalization of an environment gives an equivalent one:

Lemma 2. *Assume an environment Γ and a finite set X of variables such that $\{x \in \mathbf{Var} \mid \Gamma(x) \neq \texttt{any()}\} \subseteq X$. Then $\mathcal{T}_{Env}[\![\Gamma]\!] \subseteq \mathcal{T}_{Env}[\![norm(X,\Gamma)]\!]$.*

As a consequence of this, we can define the greatest lower bound operator $\sqcap$ on environments in terms of the normalization and *GLB* operators:

$$\Gamma_1 \sqcap \Gamma_2 \stackrel{\text{def}}{=} GLB(norm(X,\Gamma_1), norm(X,\Gamma_2)) \quad \text{where } X = \{x \in \mathbf{Var} \mid \Gamma_1(x) \neq \texttt{any()} \vee \Gamma_2(x) \neq \texttt{any()}\}$$

Since we are using the same X for normalizing both Γ_1 and Γ_2, the resulting environments are compatible and hence we can apply the *GLB* operator on them. Moreover, the resulting $\sqcap$ operator satisfies the following: $\mathcal{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!] = \mathcal{T}_{Env}[\![GLB(norm(X,\Gamma_1), norm(X,\Gamma_2))]\!] = \mathcal{T}_{Env}[\![norm(X,\Gamma_1)]\!] \cap \mathcal{T}_{Env}[\![norm(X,\Gamma_2)]\!] \supseteq \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$.

So far we have applied the notion of normalization to type environments. Now we shall extend this notion to pairs $\langle \tau; \Gamma \rangle$. Assume that $ftv(\tau) = \{\beta_1, \ldots, \beta_m\}$. Given a finite set X of variables we define $norm(X, \langle \tau; \Gamma \rangle)$ as the pair $\langle \alpha; \Gamma' \rangle$, being Γ' the following environment:

$$[x_1 : \alpha_1, \ldots, x_n : \alpha_n \mid \{\overline{\alpha_i \subseteq \Gamma(x_i)}^n, \alpha \subseteq \tau[\overline{\beta_i/\beta'_i}^m], \overline{\beta'_i \subseteq \beta_i}^m\} \cup \Gamma|_C]$$

where $X = \{x_1, ..., x_n\}$, and $\{\alpha_1, ..., \alpha_n, \beta'_1, \ldots, \beta'_m, \alpha\}$ do not appear in Γ.

Lemma 3. *For any set of variables X and any pair $\langle \tau; \Gamma \rangle$ such that $\{x \in \mathbf{Var} \mid \Gamma(x) \neq \texttt{any()}\} \subseteq X$, it holds that $\mathcal{T}[\![\tau; \Gamma]\!] \subseteq \mathcal{T}[\![norm(X, \langle \tau; \Gamma \rangle)]\!]$.*

We can take advantage of the normalization on pairs $\langle \tau; \Gamma \rangle$ to give a proper definition for the $\otimes$ operator. Given n pairs, each one of the form $\langle \tau_i; \Gamma_i \rangle$, let us denote by X the set of variables x such that $\Gamma_i(x) \neq \texttt{any()}$ for some $i \in \{1..n\}$. Let us denote the result of $norm(X, \langle \tau_i; \Gamma_i \rangle)$ by $\langle \alpha_i; \Gamma'_i \rangle$ for each $i \in \{1..n\}$. Then the environments $\Gamma'_1, \ldots, \Gamma'_n$ are pairwise compatible, so we define the product $\langle \tau_1; \Gamma_1 \rangle \otimes \ldots \otimes \langle \tau_n; \Gamma_n \rangle$ as follows:

$$\langle \tau_1; \Gamma_1 \rangle \otimes \ldots \otimes \langle \tau_n; \Gamma_n \rangle \stackrel{\text{def}}{=} \langle \{\alpha_1, \ldots, \alpha_n\}; \Gamma'_1 \sqcap \ldots \sqcap \Gamma'_n \rangle$$

This allows us to show the property we had assumed in Section 5, which states that a collection of pairs $\langle \tau_1; \Gamma_1 \rangle, \ldots, \langle \tau_n; \Gamma_n \rangle$ can be used to build a tuple type:

Lemma 4. *Assume a substitution θ, a finite set of values $v_1, \ldots, v_n$, types $\tau_1, \ldots, \tau_n$ and environments $\Gamma_1, \ldots, \Gamma_n$. Assume that $(\theta, v_i) \in \mathcal{T}[\![\tau_i; \Gamma_i]\!]$ for each $i \in \{1..n\}$. Then it holds that $(\theta, (\{\overset{n}{\cdot\cdot\cdot}\}, v_1, \ldots, v_n)) \in \mathcal{T}[\![\langle \tau_1; \Gamma_1 \rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n \rangle]\!]$.*

7.2 Soundness results

Given a judgement $\Gamma \vdash e : \tau, \Gamma'$, the main correctness result states that, given a substitution $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ such that $e\theta$ evaluates to a value v, the latter is contained within the semantics of τ under a substitution π', and θ is contained within the final environment Γ' under a substitution π, where $\pi' \subseteq \pi$.

Theorem 1. *Let us assume some typing environments Γ and Γ', a type τ, an expression e, a clause cls and a set X of variables. If $\Gamma \vdash e : \tau, \Gamma'$, then $\mathcal{E}[\![e]\!] \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau; \Gamma']\!]$.*

The following corollary shows that, in the particular case of closed expressions, our rules derive indeed success types in the sense of [14], which was in turn a generalization of [12]:

Corollary 1. *If e is a closed expression and $[\,] \vdash e : \tau, [\,]$, then $\mathcal{E}[\![e]\!] \subseteq \mathcal{T}[\![\tau]\!]$.*

8 Conclusions and Related Work

We have presented a set of typing rules for a significant subset of Core Erlang. Formally, the type judgements derived by our rules obtain, under a given type environment, a type for an expression e together with a new type environment expressing conditions for the free variables in e that are necessary for the successful evaluation of e, and a set of type constraints for the type variables of the given type. When the rules are applied to closed expressions, they derive success types, i.e., overapproximations of the semantics.

The syntax of types presented in this paper involves the existence of universally quantified types nested inside other types, as in System F [8]. Although type inference in System F is undecidable, in our context this problem becomes trivial, as we can always derive a type for an expression. In fact, we can always derive the `any()` type, as stated in Proposition 1. The problem of finding an *accurate* type for a given expression is more involved, and hence left as future work.

A significant amount of research has been carried out in order to apply type-based static analysis to dynamically typed languages. A well-known example is Typescript [3], which is a superset of Javascript that provides static typing and class-based objects. These techniques have also been applied to other languages, such as Ruby [7, 1] and Python [2]. Although the latter is oriented towards the translation of Python into JVM and CLI primitive instructions (instead of emulating the Python model on top of the corresponding virtual machine), these systems allow the programmer to catch type errors at compile time. However, they follow the traditional approach of ensuring the absence of type errors at runtime, even if some false positives are reported. The type system introduced in this paper follows the opposite goal introduced by success types [12], that is, to avoiding false positives. Our goal is to assist the programmer in detecting as many definite errors as possible. Although some other subtler type errors may be left unreported, this approach can be combined with the variety of mechanisms that Erlang provides (such as supervision trees) for reporting and restarting the program state in the event of crashes.

Another approach to apply type-based static analysis is *soft typing* [5], which is a technique to find those places in a program where type consistency is not guaranteed, in order to insert run-time type checks. This approach shares Dialyzer's philosophy insofar it does not require type annotations from the programmer. A soft type checker does not reject programs with potential type errors, but unlike success typing, it is conservative in the sense that it inserts type checks whenever in doubt. Some implementations of soft type systems have been developed for Scheme [25] and Erlang [16], the latter introducing a specification language for specifying the interface of Erlang modules. As acknowledged by its author, the latter system might produce false positives such as in function `lists:nth/2` when there is no guarantee that the list is accessed within its bounds. Another difference of our system with respect to soft typing is the addition of type environments in functional types, which capture the necessary conditions for the evaluation of the function. This information is taken into account in order to precisely catch type errors when analysing function applications, while maintaining modularity.

Another area of research related to the integration of static typing into dynamically typed languages is that of *gradual typing* [21, 22]. Unlike the all-or-nothing approach provided by

traditional languages, a gradual type system allows programmers to partially annotate their programs with types, while the unannotated parts of the program have implicitly a dynamic type, which roughly corresponds to the `any()` type in this work. Gradual type systems have also been studied in the context of imperative languages [18, 23]. There exists an inference algorithm for gradual types [17] which consists in constraining the types of variables from their definitions and assignments (*inflows*) and from the context in which they appear (*outflows*). The latter bound the set of values which a variable may contain at runtime, in a similar way as our type environments represent an upper bound of the values of all the variables in scope. However, the goal of the gradual typing inference is not to detect type discrepancies, but to carry out performance optimizations, in the same way as soft type systems. In this sense, we can say that the type system presented here is closer to the notion of success types which we extend in order to obtain polymorphic types.

The set of rules presented in this paper provides a solid foundation that will allow us to design and implement an algorithm for inferring polymorphic success types, which is left as future work. Other additional line of future work would be to support overloaded function specifications in the sense of [9] which can capture the semantics of a function in more accurate way. Another future direction of this research is to adapt these ideas to other dynamically-typed mainstream languages, such as Javascript or Python.

References

[1] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 459–472, New York, NY, USA, 2011. ACM.

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM.

[3] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 2014.

[4] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0.3 language specification, november 2004.

[5] Robert Cartwright and Mike Fagan. Soft typing. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 278–292. ACM, 1991.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[7] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[8] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159 – 192, 1986.

[9] Miguel Jimenez, Tobias Lindahl, and Konstantinos F. Sagonas. A language for specifying type contracts in erlang and its interaction with success typings. In Simon J. Thompson and Lars-Åke Fredlund, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007*, pages 11–17. ACM, 2007.

[10] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Programming Languages and Systems*, pages 91–106. Springer, 2004.

[11] Tobias Lindahl and Konstantinos Sagonas. Typer: a type annotator of erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25. ACM, 2005.

[12] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM.

[13] Francisco J. López-Fraguas, Manuel Montenegro, and Gorka Suárez-García. Polymorphic success types for erlang (extended version). Technical report, Number 03/18. Dpto. de Sistemas Informáticos y Computación, 2018. Available at https://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos.

[14] Francisco Javier López-Fraguas, Manuel Montenegro, and Juan Rodríguez-Hortalá. Polymorphic types in erlang function specifications. In *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 181–197, 2016.

[15] Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 136–149, New York, NY, USA, 1997. ACM.

[16] Sven-Olof Nyström. A soft-typing system for erlang. In Bjarne Däcker and Thomas Arts, editors, *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, pages 56–71. ACM, 2003.

[17] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '12*, page 481, 2012.

[18] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM.

[19] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[20] Konstantinos F. Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 13–18. Springer, 2010.

[21] J Siek and W Taha. Gradual typing for functional languages. In *Scheme and Functional Programming*, pages 81–92, 2006.

[22] Jeremy Siek and Walid Taha. Gradual Typing for Objects. *ECOOP 2007 - Object-Oriented Programming*, pages 2–27, 2007.

[23] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 45–56, New York, NY, USA, 2014. ACM.

[24] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.

[25] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *LISP and Functional Programming*, pages 250–262, 1994.