A type derivation system for Erlang (extended version)

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

Dpto. de Sistemas Informáticos y Computación – Universidad Complutense de Madrid

fraguas@ucm.es montenegro@fdi.ucm.es gorka.suarez@ucm.es

Technical Report 01-17 / February 19, 2017

Abstract

Erlang is a dynamically typed concurrent functional language of increasing interest in industry and academy. 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 in progress is the starting point of a medium-term project aiming at improving both aspects, so that at its end we should have proposed a full type system able to infer polymorphic success types for Erlang programs, accompanied by solid theoretical foundations including adequateness results for the type system. In this first step we only provide a derivation system of monomorphic success types for Erlang, along with correctness results with respect to a suitable semantics for the language.

Contents

1 Introduction

Erlang is a concurrent functional language arousing increasing interest in industry and academy for its strength in producing robust, easy to build and mantain, 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 easier to manage, static compilation time errors obtained by type systems *à la* Hindley-Milner [2] 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 not practically successful attempts [8], the Typer [5] and Dialyzer tools [4, 6, 3] were proposed and are currently incorporated to the official distribution of Erlang. It can be used to extract the implicit type information from the programs, both for documentation purposes and for finding errors at compile time. In order to respect the actual flexibility of Erlang, a essential design principle of Dialyzer was that it should not produce *false positives*, that is, signalling a type error should only happen in situations where it is certain that a runtime error will occur[1]. As it said in [9], 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*[6], 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. On the practical side, its ability to deal with functions that are typically polymorphic is rather poor. Dialyzer is not designed to infer by itself polymorphic types. To overcome that, user-given polymorphic type specifications were considered in [3]. Dialyzer takes into account those specifications but in such a way that most of polymorphism is lost (see [7] for many examples of that).

In [7] we made a first contribution to address that problem, where we significantly improve Dialyzer's behavior with respect to polymorphism by using Dialyzer itself, following a transformational approach: given an Erlang program with user-given polymorphic type specifications, we synthesize a new Erlang program such that Dialyzer, when run over the transformed program, infers more precise types for expressions that use polymorphic functions.

However, making new progresses along that path is seriously limited by its tight dependence of Dialyzer. Any change made to the underlying type system of the tool, or to its implementation, could affect and even invalidate the transformations proposed in [7]. Moreover, proving any theoretical result about our technique rely on trusting on non rigorously proved properties of Dialyzer. This is a second relevant –to the purpose of this report– 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.

So we arrive to the main motivation of our work: in the medium term we want to develop a full type system, independent of Dialyzer, with associated type checking and type inference mechanisms that follows Dialyzer's philosophy of success types, coping appropriately with the issues of polymorphism and having at the same time rigorous theoretical foundations. This work in progress is a first step towards this aim. Concretely, we propose a

[1] To be more precise, this can only be ensured for terminating computations.
[2] Again, this strictly applies only to terminating computations.

set of typing rules for deriving monomorphic 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 non-determinism. This is admittedly a limited achievement with respect to our complete plan, but we consider it as a needed step that, in addition, is not technically trivial. We discussed in [7] that the mere concept of polymorphic success type is subtle and its 'definitive' definition is still to come. We prefer to consolidate first an appropriate type system for the monomorphic case, which is of course simpler but not 'that' simpler. For instance, one of the technicalities we need to face is that, within the success types view, having an ill-typed subexpression does not imply that the whole expression is also ill-typed, contrary to the case of Hindley-Milner systems, where well-typed expressions require all their pieces to be also well-typed. We will need to take this into account in our typing rules, in particular when considering typing of functional abstractions, since abstractions have always a non-empty type, even if the body is ill-typed. However we must take care because the application of the abstraction may result in ill-typedness.

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

2 Preliminaries

2.1 The syntax of the language

Our work will focus on Core Erlang, a simpler version of Erlang. The full syntax of Core Erlang can be found in [1], but we will use the subset shown in Figure 1.

$$
\begin{array}{rcl}
lit & ::= & \textsc{Atom} \mid \textsc{Integer} \mid \textsc{Float} \mid \textsc{Nil} \\
var & ::= & \textsc{VariableName} \\
fun & ::= & \textbf{fun}(var_1, \ldots, var_n) \rightarrow exp \\
exp & ::= & lit \mid var \mid [exp_1 \mid exp_2] \mid \{exp_1, \ldots, exp_n\} \mid fun \\
 & \mid & \textbf{let}\ var = exp_1\ \textbf{in}\ exp_2 \\
 & \mid & \textbf{letrec}\ var_1 = fun_1, \ldots, var_n = fun_n\ \textbf{in}\ exp \\
 & \mid & \textbf{case}\ var\ \textbf{of}\ cls_1 \ldots cls_n\ \textbf{end} \\
 & \mid & \textbf{receive}\ cls_1 \ldots cls_n\ \textbf{after}\ txp \rightarrow exp \\
 & \mid & var(var_1, \ldots, var_n) \\
cls & ::= & pat\ \textbf{when}\ exp_1 \rightarrow exp_2 \\
pat & ::= & var \mid lit \mid [pat_1 \mid pat_2] \mid \{pat_1, \ldots, pat_n\} \\
txp & ::= & \textsc{Integer} \mid \texttt{'infinity'} \mid var
\end{array}
$$

Figure 1: Subset of the Core Erlang syntax

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

There are some differences between our subset and Core Erlang. That help us to simplify the typing rules without losing generality. The first is that Core Erlang uses three ways to apply functions. The first one uses **apply** to invoke functions inside the current module or inside a variable. The second one uses **call** to invoke functions outside the current module. The last one uses **primop** to invoke some language primitives like `'raise'`.[3] In our subset we use only one way to apply functions, to remove some noise in our rules.

Core Erlang has tuples, defined with the notation $\{.^{n}.\}$, but to improve the efficiency it also uses sequences internally, defined with the notation $<.^{n}.>$, which are very similar to tuples. Sequences are used by Core Erlang in **let** or **case**, and some of the allowed forms of **let** are transformed into:

$$\textbf{let } x = e' \textbf{ in } e \quad \equiv \quad \textbf{let } <x> = <e'> \textbf{ in } e$$
$$\textbf{let } <x> = e' \textbf{ in } e \quad \equiv \quad \textbf{let } <x> = <e'> \textbf{ in } e$$

Unlike tuples, sequences can not be nested. In our subset of Core Erlang, since sequences are an optimization issue of the implementation of Erlang, we will use tuples instead of sequences because their semantics are pretty much similar from the point of view of typings.

Core Erlang also has some expressions which are syntactic sugar like **do** or **catch**. We assume that the program being type checked has already been desugared, so it does not contain this kind of expressions.

Exceptions are supported in Erlang, but the programming philosophy of the language discourages its use. Because of it, in the firsts steps of our types system we choose to put more effort on the basics of the language, and leave **try**/**catch** and exceptions as a future goal.

In our subset, the **case** discriminant and the application parameters uses only variables to simplify the typing rules.[4] This allow us to add type information. We also use variables to give name to the lambda abstractions in the syntax to store the types of these functions in our environments.

2.2 The semantics of the language expressions

In a previous work [7] the semantics of a closed expression was 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 tuples $((\overline{params}), value)$ to represent a graph where the input values in the parameters are connected to the result value. The motivation of these ideas was related to the nondeterminism of the Erlang language. To extend this to expressions with free variables we need to consider substitutions to give values to variables. A substitution θ is a total function $Var \to DVal$. We write **Subst** for 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

[3] The primitive `'raise'` is used to throw exceptions inside Erlang.

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

0 to the other variables. The semantics $\mathscr{E}[\![e]\!]^\star$ of an expression e is defined as a relation between substitutions and values, that is:

$$\mathscr{E}[\![e]\!]^\star \subseteq \mathbf{Subst} \times \mathit{DVal}$$

The idea is that if $(\theta, v) \in \mathscr{E}[\![e]\!]^\star$ then v is one of the possible values to which $e\theta$ can be reduced. The complete definition of $\mathscr{E}[\![e]\!]^\star$ is given in Figure 2. If e is a closed expression, its evaluation does not depend on the values mapped to by θ, so we can define a semantics $\mathscr{E}[\![\_]\!]$ for closed expressions that disregards the information of the substitutions, that is, $\mathscr{E}[\![e]\!] = \{v \mid (\theta, v) \in \mathscr{E}[\![e]\!]^\star \text{ for some } \theta\}$.

$$\mathscr{E}[\![c]\!]^\star = \{(\theta, c) \mid \theta \in \mathbf{Subst}\}$$
$$\mathscr{E}[\![x]\!]^\star = \{(\theta, \theta(x)) \mid \theta \in \mathbf{Subst}\}$$
$$\mathscr{E}[\![\{\overline{e_i}^n\}]\!]^\star = \left\{\left(\theta, \left(\{\cdot^n\cdot\}, v_1, \ldots, v_n\right)\right) \mid (\theta, v_1) \in \mathscr{E}[\![e_1]\!]^\star, \ldots, (\theta, v_n) \in \mathscr{E}[\![e_n]\!]^\star\right\}$$
$$\mathscr{E}[\![[e_1 \mid e_2]]\!]^\star = \left\{(\theta, ([\_|\_], v_1, v_2)) \mid (\theta, v_1) \in \mathscr{E}[\![e_1]\!]^\star, (\theta, v_2) \in \mathscr{E}[\![e_2]\!]^\star\right\}$$
$$\mathscr{E}[\![\mathbf{fun}(\overline{x_i}^n) \to e]\!]^\star = \left\{\left(\theta, \left\{((v_1, \ldots, v_n), v) \mid (\theta[\overline{x_i / v_i}], v) \in \mathscr{E}[\![e]\!]^\star\right\}\right) \mid \theta \in \mathbf{Subst}\right\}$$
$$\mathscr{E}[\![f(\overline{x_i}^n)]\!]^\star = \left\{(\theta, v) \mid ((\theta(x_1), \ldots, \theta(x_n)), v) \in \theta(f)\right\}$$
$$\mathscr{E}[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2]\!]^\star = \left\{(\theta, v) \mid (\theta, v_1) \in \mathscr{E}[\![e_1]\!]^\star, (\theta[x_1 / v_1], v) \in \mathscr{E}[\![e_2]\!]^\star\right\}$$
$$\mathscr{E}\left[\!\left[\mathbf{case}\ x\ \mathbf{of}\ \overline{cls_i}^n\right]\!\right]^\star = \bigcup_{i=1}^{n} \Big\{(\theta, v) \mid v_1, \ldots, v_m \in \mathit{DVal}, (\theta[\overline{x_{ij} / v_j}], v) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}}$$
$$(\forall k < i. \forall \overline{v_j}'. \forall v'. (\theta[\overline{x_{kj} / v_j'}], v') \notin \mathscr{C}[\![cls_k]\!]_{\{\theta(x)\}})\Big\}$$
$$\text{where } cls_i = \left(p_i\ \mathbf{when}\ e_i \to e_i'\right) \text{ for every } i \in \{1..n\} \text{ and } \mathit{vars}(p_i) = \{\overline{x_{ij}}\}$$
$$\mathscr{E}\left[\!\left[\mathbf{receive}\ \overline{cls_i}^n\ \mathbf{after}\ e_t \to e\right]\!\right]^\star = \bigcup_{i=1}^{n} \{(\theta, v) \mid v_1, \ldots, v_n \in \mathit{DVal},$$
$$(\theta[\overline{x_{ij} / v_i}], v) \in \mathscr{C}[\![cls_i]\!]_{\mathit{DVal}}, (\forall k < i. \forall \overline{v_j}'. \forall v'. (\theta[\overline{x_{kj} / v_j'}], v') \notin \mathscr{C}[\![cls_k]\!]_{\mathit{DVal}}),$$
$$(\theta, v_t) \in \mathscr{E}[\![e_t]\!]^\star, v_t \in \mathit{Integer} \cup \{\mathit{infinity}\}\}$$
$$\cup \{(\theta, v) \mid (\theta, v) \in \mathscr{E}[\![e]\!]^\star, (\theta, v_t) \in \mathscr{E}[\![e_t]\!]^\star, v_t \in \mathit{Integer}\}$$
$$\text{where } cls_i = \left(p_i\ \mathbf{when}\ e_i \to e_i'\right) \text{ for every } i \in \{1..n\} \text{ and } \mathit{vars}(p_i) = \{\overline{x_{ij}}\}$$
$$\mathscr{E}[\![\mathbf{letrec}\ \overline{x_i = e_i}^n\ \mathbf{in}\ e]\!]^\star = \left\{(\theta, v) \mid (v_1, \ldots, v_n) = \mathit{lfp}\, F_\theta, (\theta[\overline{x_i / v_i}], v) \in \mathscr{E}[\![e]\!]^\star\right\}$$
$$\text{where } F_\theta(v_1, \ldots, v_n) = (v_1', \ldots, v_n')$$
$$\text{where } \forall k \in \{1..n\}. \forall k \in \{1..n\}. \left\{v_k'\right\} = \left\{v \mid (\theta[\overline{x_i / v_i}], v) \in \mathscr{E}[\![e_k]\!]^\star\right\}$$
$$\mathscr{C}[\![p\ \mathbf{when}\ e_g \to e]\!]_V = \left\{(\theta, v) \mid (\forall v' \in V. (\theta, v') \in \mathscr{E}[\![p]\!]^\star), (\theta, \mathit{true}) \in \mathscr{E}[\![e_g]\!]^\star, (\theta, v) \in \mathscr{E}[\![e]\!]^\star\right\}$$

Figure 2: Denotational semantics of expressions

3 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. Each type is meant to overapproximate a set of values. 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 the notation $\mathscr{B}[\![B]\!]$ represents the set of values denoted by this type. For instance, $\mathscr{B}[\![\texttt{integer()}]\!]$ includes the set of integer numbers, whereas $\mathscr{B}[\![\texttt{atom()}]\!]$ denotes the set of Erlang atoms (i.e. symbolic constants).

3.1 Syntax of types and environments

We denote by **Type** the set of types generated by the following grammar:

$$\tau ::= \texttt{none()} \mid \texttt{any()} \mid B \mid v \mid \{\tau_1,\ldots,\tau_n\} \mid \texttt{nelist(}\tau_1\texttt{,}\tau_2\texttt{)} \mid \tau_1 \cup \tau_2 \mid (\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau$$

where $B \in \mathbb{B}$ and $v \in \mathit{DVal}$.

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, the evaluation will fail or diverge. On the contrary, the type `any()` denotes the set *DVal* containing all values. Therefore, `any()` always overapproximates the set of values to which an expression is evaluated. In other words, `any()` is a success type for every expression. Besides this, the type system features singleton types, in the sense that for every value $v \in \mathit{DVal}$ there is a type v denoting the set $\{v\}$.

The type $\{\tau_1,\ldots,\tau_n\}$ denotes those tuples whose i-th component is contained within the values represented by τ_i, for each $i \in \{1..n\}$. The type `nelist(`τ_1`,`τ_2`)` represents those 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 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 $[\tau]$ as a shorthand for this type.

The type $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau$ denotes the set of n-ary functions that accept values in $(\tau_1,\ldots,\tau_n)$ and yield a result in τ. The Γ lying above the arrow is a type environment which may pose constraints on the free variables occurring in the function's body. These constraints represent necessary conditions for the evaluation of the function. The role of this environment Γ will be explained later in this section.

A type environment is a total function from variables to types. Notice that, unlike the standard Hindley-Milner type systems, the definitions of type environments and types are mutually recursive, as environments contain types, and functional types contain environments. We use `[]` to denote an environment which maps every variable to `any()` whereas $\bot$ denotes the bottom environment mapping every variable to `none()`. Given an environment Γ (resp. a type τ) and a set X of variables, the notation $\Gamma\backslash X$ (resp. $\tau\backslash X$) stands for the environment that results from replacing the types of the variables of X by `any()`. This does not only include the bindings in the environment Γ, but also applies recursively to the environments attached to the functional types in Γ. More specifically, $\Gamma\backslash X$

denotes the environment Γ' such that

$$\Gamma'(y) = \begin{cases} \texttt{any()} & \text{if } y \in X \\ \Gamma(y)\backslash X & \text{otherwise} \end{cases}$$

where $\Gamma(y)\backslash X$ is the type that results from replacing each functional type $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau$ by the type $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma\backslash X} \tau$. 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 $\Gamma\backslash x$ or $\tau\backslash x$.

As an example, let us consider the expression $\textbf{fun}(X) \to X+1$. We shall see later that $\texttt{number()} \xrightarrow{[]} \texttt{number()}$ is a success type for this expression. This means that if this function is given an argument which is not a number, its execution will fail. Otherwise, the execution of the function may succeed or not, but in the case it does, it will return a numeric value. The empty environment [] above the arrow specifies that the execution of this function does not demand further conditions on any variable other than X. Now, let us consider the expression $\textbf{fun}(X) \to X+Y$, in which Y is a free variable. The execution of the function, when it is applied to a value v, does not only depend on the fact of v being a number, but also on the value of Y. In fact, Y must contain a numeric value, or the evaluation of the function's body would fail otherwise. However, the latter fact cannot be reflected neither in the type of the input parameters nor in the type of the result. In fact, $\textbf{fun}(X) \to X+1$ accepts the following success type,

$$\texttt{number()} \xrightarrow{[Y:\texttt{number()}]} \texttt{number()} \tag{1}$$

which means that the fact of Y having type `number()` is a necessary condition for the evaluation of $X+Y$. In the following, we shall leave out the environment from the arrow of functional types when it is empty.

3.2 Semantics of types

The next step is to provide a semantics for each of these types. 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 $\mathcal{T}[\![\tau]\!] \subseteq \mathit{DVal}$ containing the values of the language abstracted by τ. However, the example above shows that there are types whose semantics does not solely depend on the type itself, but also on the types assigned to some free variables. For instance, if τ is the type shown in (1), and Y contains a numeric value, this type would denote the set of functions from numbers to numbers. Since we represent function values by their graphs, we would have the following semantics for τ:

$$\{F \mid F \subseteq \{((v), v') \mid v, v' \in \mathit{Number}\}\}$$

However, if Y were bound to a nonnumeric value, then the intended semantics of (1) is the function that always fails. This function is represented by the empty graph, so the semantics of τ would be a single function which is the empty graph, that is, $\{\emptyset\}$. This example proves the need for a semantics that is also parametric on the *context* in which a type occurs. This context is given by a substitution θ that contains the value assigned to every variable. Therefore, instead of $\mathcal{T}[\![\tau]\!]$ we write $\mathcal{T}_\theta[\![\tau]\!]$ to denote the semantics of the type τ under a substitution θ. The definition of $\mathcal{T}_\theta[\![\tau]\!]$ is shown in Figure 3. This definition is mutually recursive with a function $\mathcal{T}_{Env}[\![\_]\!]$ that yields the semantics of a type environment. Since a type environment Γ is a mapping from variables to types, it is natural to define its semantics as the set of those substitutions θ such that $\theta(x)$ is contained within the semantics of $\Gamma(x)$ for each variable x.

Similarly to expressions, whose semantics are defined as a relation between substitutions and values, we adapt

$$
\begin{aligned}
\mathscr{T}_\theta[\![\texttt{none()}]\!] &= \emptyset \\
\mathscr{T}_\theta[\![\texttt{any()}]\!] &= \mathit{DVal} \\
\mathscr{T}_\theta[\![B]\!] &= \mathscr{B}[\![B]\!] \\
\mathscr{T}_\theta[\![v]\!] &= \{v\} \\
\mathscr{T}_\theta[\![\{\tau_1,\ldots,\tau_n\}]\!] &= \left\{\left(\{\cdot^n_\cdots\}, v_1,\ldots,v_n\right) \mid \forall i \in \{1..n\}.v_i \in \mathscr{T}_\theta[\![\tau_i]\!]\right\} \\
\mathscr{T}_\theta[\![\texttt{nelist}(\tau_1,\tau_2)]\!] &= \mathit{lfp}\,\lambda Z.\left\{([\_|\_], v_1, v_2) \mid v_1 \in \mathscr{T}_\theta[\![\tau_1]\!], v_2 \in \mathscr{T}_\theta[\![\tau_2]\!]\right\} \\
&\qquad \cup \left\{([\_|\_], v_1, z) \mid v_1 \in \mathscr{T}_\theta[\![\tau_1]\!], z \in Z\right\} \\
\mathscr{T}_\theta[\![\tau_1 \cup \tau_2]\!] &= \mathscr{T}_\theta[\![\tau_1]\!] \cup \mathscr{T}_\theta[\![\tau_2]\!] \\
\mathscr{T}_\theta\left[\!\left[(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau\right]\!\right] &= \begin{cases} \{\emptyset\} \text{ if } \theta \notin \mathscr{T}_{Env}[\![\Gamma]\!] \\ \left\{F \mid F \subseteq \left\{((v_1,\ldots,v_n), v) \mid \forall i.v_i \in \mathscr{T}_\theta[\![\tau_i]\!], v \in \mathscr{T}_\theta[\![\tau]\!]\right\}\right\} \text{ otherwise} \end{cases}
\end{aligned}
$$

$$
\mathscr{T}_{Env}[\![\Gamma]\!] = \left\{\theta \in \mathbf{Subst} \mid \forall x \in \mathit{Var}.\theta(x) \in \mathscr{T}_\theta[\![\Gamma(x)]\!]\right\}
$$

Figure 3: Sematics of types and type environments.

our type semantics in the same way. We use the notation $\mathscr{T}[\![\tau]\!]^\star$ to denote the semantics of τ as a relation between substitutions and values, where each substitution θ is related to each value in $\mathscr{T}_\theta[\![\tau]\!]$. That is,

$$
\mathscr{T}[\![\tau]\!]^\star \stackrel{def}{=} \left\{(\theta, v) \mid \theta \in \mathbf{Subst}, v \in \mathscr{T}_\theta[\![\tau]\!]\right\}
$$

In the context of closed expressions, the substitution θ is not relevant to the semantics of a type, so we can define $\mathscr{T}[\![\tau]\!] = \{v \mid (\theta, v) \in \mathscr{T}[\![\tau]\!]^\star \text{ for some } \theta\}$.

With these definitions we can say that τ is a success type for the expression e if, for every substitution θ, the set of values to which $e\theta$ can be evaluated is contained within the semantics of τ under the same substitution. More precisely, τ is a success type for e if and only if $\mathscr{E}[\![e]\!]^\star \subseteq \mathscr{T}[\![\tau]\!]^\star$. This definition generalizes the one given previously in [7], because when e is closed and all the environments in τ are empty, it is equivalent to $\mathscr{E}[\![e]\!] \subseteq \mathscr{T}[\![\tau]\!]$.

4 Typing judgements

The definition of success types given in [3] states that $\tau_1 \to \tau_2$ is success type for a function f if and only if, for all $v, v' \in \mathit{DVal}$, 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 $\tau_1 \to \tau_2$. In [7] we generalize this notion to arbitrary closed expressions. However, if we want to derive the success type of an expression with free variables, the type system has to derive, in addition to the type of the expression itself, some necessary conditions on these free variables. For instance, a necessary (but not sufficient) condition for the evaluation of X/Y is that X and Y are numbers. This condition is expressed by the type environment $[X : \texttt{number()}, Y : \texttt{number()}]$.

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 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 free variables in e have values contained within the types in Γ, if e is evaluated to a value v, then v is of type τ and the free variables in e have values contained within the types in Γ'. This intuition can be expressed as follows:

$$\forall \theta \in \textbf{Subst}, v \in DVal : \theta \in \mathcal{T}_{Env}[\![\Gamma]\!] \wedge (\theta, v) \in \mathcal{E}[\![e]\!]^\star \Rightarrow (\theta, v) \in \mathcal{T}[\![\tau]\!]^\star \wedge \theta \in \mathcal{T}_{Env}[\![\Gamma']\!]$$

which can be rewritten more succintly as $\mathcal{E}[\![e]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$. In the following we use the terms *assumption environment* and *final enviroment* to refer to Γ and Γ' respectively.

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 rules [SUB-2] and [SUB-T] allow us to weaken the type τ or the final environment Γ_2. The subtyping relation between environments $\Gamma \subseteq \Gamma'$ is defined as $\mathcal{T}_{Env}[\![\Gamma]\!] \subseteq \mathcal{T}_{Env}[\![\Gamma']\!]$. Similarly, $\tau \subseteq \tau'$ if an only if $\mathcal{T}[\![\tau]\!]^\star \subseteq \mathcal{T}[\![\tau']\!]^\star$. The set of types with the $\subseteq$ order form a lattice, and so does the set of environments. We shall use the notation $\tau_1 \sqcup \tau_2$ and $\tau_1 \sqcap \tau_2$ to denote, respectively, the least upper bound and the greatest lower bound of τ_1 and τ_2. This notation is extended to environments in a similar way.

The rule [NONE] allows us to infer failure in case in which one of the variables in the assumption environment has type `none()`. The [TRANS] rule specifies that, whenever we have a judgement $\Gamma_1 \vdash e : \tau, \Gamma_2$ we can reevaluate the type of e under the assumptions given in Γ_2. This reevaluation allows us to infer possibly more precise types for some expressions, such as the one explained in Section 5.1.

The [CONST] and [VAR] rules specify that the final environment poses no further constraints besides those in the assumption environment, whereas [TUPLE] and [LIST] take the greatest lower bound of all the final environments in every subexpression, since all of them must be satisfied in order to evaluate the whole expression. 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. The restrictions expressed in the final environment Γ' that results from analyzing the function's body 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: [APP1] only makes sense when the type assumed for f is compatible with a functional type, whereas [APP2] 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 $\tau_1, \ldots, \tau_n$, and it poses additional constraints on the free variables (denoted by Γ') demanded by the function execution.

The [LET] rule is rather standard. The only difference is that we remove the constraints involving the bound variable x_1 from the type and the final environment, since this variable is not free in the **let** expression. For every judgement $\Gamma \vdash e : \tau, \Gamma'$ nonfree variables in e must have the same type in Γ and Γ', as the execution of e succeeds or fails regardless of their value.

In order to derive a type for a **case** or a **receive** expression, we have to derive one for each of its clauses. Figure 5 contains the type rules for clauses. In this rule 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). The notation $[X : \tau]$ stands for the environment that maps every variable in X to τ and the remaining ones to `any()`. The rule [CLS-TRUE] handles those cases in which the type of the discriminant is compatible with the type of the pattern and the type of the guard is compatible with the atom *true*. The remaining cases are dealt with by the rule [CLS-FALSE].

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

$$\frac{\Gamma_1 \vdash e : \tau, \Gamma_2 \qquad \tau \subseteq \tau'}{\Gamma_1 \vdash e : \tau', \Gamma_2} \text{ [SUB-T]} \qquad \frac{}{\Gamma_1 \sqcap [x : \texttt{none}()] \vdash e : \texttt{none}(), \bot} \text{ [NONE]}$$

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

$$\frac{\Gamma \vdash e_i : \tau_i, \Gamma_i}{\Gamma \vdash \{e_1, \ldots, e_n\} : \{\tau_1, \ldots, \tau_n\}, \bigsqcap_{i=1}^{n} \Gamma_n} \text{ [TUPLE]} \qquad \frac{\Gamma \vdash e_1 : \tau_1, \Gamma_1 \qquad \Gamma \vdash e_2 : \tau_2, \Gamma_2}{\Gamma \vdash [e_1 \mid e_2] : \texttt{nelist}(\tau_1, \tau_2), \Gamma_1 \sqcap \Gamma_2} \text{ [LIST]}$$

$$\frac{\Gamma \vdash e : \tau, \Gamma'}{\Gamma \vdash \mathbf{fun}(x_1, \ldots, x_n) \to e : \left((\Gamma'(x_1), \ldots, \Gamma'(x_n)) \xrightarrow{\Gamma'} \tau \right) \backslash \{x_1, \ldots, x_n\}, \Gamma} \text{ [ABS]}$$

$$\frac{\Gamma(f) \sqcap ((\texttt{any}(), \overset{n}{\cdots}, \texttt{any}()) \xrightarrow{[]} \texttt{any}()) = (\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma'} \tau}{\Gamma \vdash f(x_1, \ldots, x_n) : \tau, \Gamma \sqcap [f : (\texttt{any}(), \overset{n}{\cdots}, \texttt{any}()) \xrightarrow{[]} \texttt{any}(), x_1 : \tau_1, \ldots, x_n : \tau_n] \sqcap \Gamma'} \text{ [APP1]}$$

$$\frac{\Gamma(f) \sqcap ((\texttt{any}(), \overset{n}{\cdots}, \texttt{any}()) \xrightarrow{[]} \texttt{any}()) = \texttt{none}()}{\Gamma \vdash f(x_1, \ldots, x_n) : \texttt{none}(), \bot} \text{ [APP2]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma_1 \qquad \Gamma_1 \sqcap [x_1 : \tau_1] \vdash e_2 : \tau_2, \Gamma_2}{\Gamma \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 : \tau_2 \backslash x_1, \Gamma_2 \backslash x_1} \text{ [LET]}$$

$$\frac{\begin{array}{c} cls_i = \left(p_i\ \mathbf{when}\ e_i' \to e_i''\right) \\ \Gamma \Vdash_{\{x\}} cls_i : \tau_i, \Gamma_i \quad \tau = \tau_i \backslash vars(p_i) \quad \Gamma' = \Gamma_i \backslash vars(p_i) \end{array}}{\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ cls_1 \ldots cls_n : \tau, \Gamma'} \text{ [CASE]}$$

$$\frac{\begin{array}{c} cls_i = \left(p_i\ \mathbf{when}\ e_i' \to e_i''\right) \\ \Gamma \vdash e_t : \tau_t, \Gamma_t \quad \Gamma_t \vdash e : \tau, \Gamma' \quad \tau_t \sqcap \texttt{number}() \neq \texttt{none}() \quad \tau_t \sqcap \texttt{infinity} \neq \texttt{none}() \\ \Gamma_t \Vdash_{\varnothing} cls_i : \tau_i, \Gamma_i \quad \tau = \tau_i \backslash vars(p_i) \quad \Gamma' = \Gamma_i \backslash vars(p_i) \end{array}}{\Gamma \vdash \mathbf{receive}\ cls_1 \ldots cls_n\ \mathbf{after}\ e_t \to e : \tau, \Gamma'} \text{ [RECEIVE-1]}$$

$$\frac{\begin{array}{c} cls_i = \left(p_i\ \mathbf{when}\ e_i' \to e_i''\right) \\ \Gamma \vdash e_t : \tau_t, \Gamma_t \quad \tau_t \sqcap \texttt{number}() = \texttt{none}() \quad \tau_t \sqcap \texttt{infinity} \neq \texttt{none}() \\ \Gamma_t \Vdash_{\varnothing} cls_i : \tau_i, \Gamma_i \quad \tau = \tau_i \backslash vars(p_i) \quad \Gamma' = \Gamma_i \backslash vars(p_i) \end{array}}{\Gamma \vdash \mathbf{receive}\ cls_1 \ldots cls_n\ \mathbf{after}\ e_t \to e : \tau, \Gamma'} \text{ [RECEIVE-2]}$$

$$\frac{\Gamma \vdash e_t : \tau_t, \Gamma_t \qquad \tau_t \sqcap (\texttt{number}() \cup \texttt{infinity}) = \texttt{none}()}{\Gamma \vdash \mathbf{receive}\ cls_1 \ldots cls_n\ \mathbf{after}\ e_t \to e : \texttt{none}(), \bot} \text{ [RECEIVE-3]}$$

$$\frac{\Gamma \sqcap [\overline{x_j : \tau_j}] \vdash e_i : \tau_i', \Gamma_i' \qquad \tau_i' \backslash \{\overline{x_j}\} = \tau_i \qquad \bigsqcap_{i=1}^{n} \left(\Gamma_i' \backslash \{\overline{x_j}\}\right) \sqcap [\overline{x_j : \tau_j}] \vdash e : \tau, \Gamma'}{\Gamma \vdash \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e : \tau \backslash \{\overline{x_j}\}, \Gamma' \backslash \{\overline{x_j}\}} \text{ [LETREC]}$$

Figure 4: Typing rules for expressions.

$$\frac{\Gamma \vdash p : \tau_p, \Gamma_p \quad \Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g \quad \tau_g \sqcap \mathit{true} \neq \texttt{none()} \quad \Gamma_g \vdash e : \tau', \Gamma'}{\Gamma \Vdash_X p \textbf{ when } e_g \rightarrow e : \tau', \Gamma'} \text{ [CLS-TRUE]}$$

$$\frac{\Gamma \vdash p : \tau_p, \Gamma_p \quad \Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g \quad \tau_g \sqcap \mathit{true} = \texttt{none()}}{\Gamma \Vdash_X p \textbf{ when } e_g \rightarrow e : \texttt{none()}, \bot} \text{ [CLS-FALSE]}$$

$$\frac{\Gamma_1 \Vdash_X \mathit{cls} : \tau, \Gamma_2 \qquad \Gamma_2 \Vdash_X \mathit{cls} : \tau', \Gamma_3}{\Gamma_1 \Vdash_X \mathit{cls} : \tau', \Gamma_3} \text{ [CLS-TRANS]}$$

Figure 5: Typing rules for clauses.

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-2] handles those cases in which the type of e_t does not contain the atom `infinity`. In this case we disregard the type of the expression in the **after** guard. The rule [RECEIVE-3] handles the case in which e_t cannot evaluate to a number or to infinity. In this case the evaluation of **receive** always fails. The remaining cases are dealt with by [RECEIVE-1].

5 Examples

In order to keep the examples shorter, we will not show the use of the rules [CONST] and [VAR] because their use is trivial.

5.1 Using the rule [TRANS]

Our first example will be a function which takes a number and returns a tuple with the given number and its successor. We will use the variable Sum as the function `'erlang':'+'`, whose type will be given in the initial environment Γ_0. The code in our subset of Core Erlang is:

$$\textbf{fun}(V_0) \rightarrow \textbf{let } V_1 = 1 \textbf{ in } \{V_0, \mathit{Sum}(V_0, V_1)\}$$

The type `(number())` $\rightarrow$ `{number(),number()}` is obtained for this lambda abstraction with our derivation. We use the following rules, where Γ_0 contains the type for Sum:

$$\frac{\begin{array}{c}\Gamma_0 = [\mathit{Sum} : (\texttt{number()}, \texttt{number()}) \xrightarrow{[]} \texttt{number()}] \\ \Gamma_0(\mathit{Sum}) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{number()}, \texttt{number()}) \xrightarrow{[]} \texttt{number()}\end{array}}{\Gamma_0 \vdash \mathit{Sum}(V_0, V_1) : \texttt{number()}, (\Gamma_1 = \Gamma_0 \sqcap [V_0 : \texttt{number()}; V_1 : \texttt{number()}])} \text{ [APP1]}$$

$$\frac{\Gamma_0 \vdash V_0 : \texttt{any()}, \Gamma_0 \qquad \Gamma_0 \vdash \mathit{Sum}(V_0, V_1) : \texttt{number()}, \Gamma_1}{\Gamma_0 \vdash \{V_0, \mathit{Sum}(V_0, V_1)\} : \{\texttt{any()}, \texttt{number()}\}, \Gamma_1} \text{ [TUPLE]}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash \{V_0, \mathit{Sum}(V_0, V_1)\} : \{\texttt{any()}, \texttt{number()}\}, \Gamma_1 \\ \Gamma_1 \vdash \{V_0, \mathit{Sum}(V_0, V_1)\} : \{\texttt{number()}, \texttt{number()}\}, \Gamma_1\end{array}}{\Gamma_0 \vdash \{V_0, \mathit{Sum}(V_0, V_1)\} : \{\texttt{number()}, \texttt{number()}\}, \Gamma_1} \text{ [TRANS]}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash 1:1,\Gamma_0 \qquad \Gamma_0 \sqcap [V_1:1] \vdash \{V_0, Sum(V_0,V_1)\}: \\ \{\texttt{number()},\texttt{number()}\},\Gamma_0 \sqcap [V_0:\texttt{number()};V_1:1]\end{array}}{\Gamma_0 \vdash \textbf{let}\ V_1 = \ldots \textbf{in} \cdots : \{\texttt{number()},\texttt{number()}\},\Gamma_0 \sqcap [V_0:\texttt{number()}]} \text{ [LET]}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash \textbf{let}\ V_1 = \ldots \textbf{in} \cdots : \{\texttt{number()},\texttt{number()}\},\Gamma_0 \sqcap [V_0:\texttt{number()}] \\ (\Gamma_0 \sqcap [V_0:\texttt{number()}]) \backslash \{V_0\} = \Gamma_0 \\ (\{\texttt{number()},\texttt{number()}\}) \backslash \{V_0\} = (\{\texttt{number()},\texttt{number()}\})\end{array}}{\Gamma_0 \vdash \textbf{fun}(V_0) \rightarrow \cdots : (\texttt{number()}) \xrightarrow{\Gamma_0} \{\texttt{number()},\texttt{number()}\},\Gamma_0} \text{ [ABS]}$$

Sometimes in the middle of a derivation, some of the steps may be forced to reevaluate the previous rules in order to reach the same environments needed to apply the current rule. In this example when the [TRANS] is applied, the previous [LIST] and [APP1] rules are needed to be reevaluated since we are building a new judgement with a different assumption environment.

The rule [TRANS] is used in this derivation to update the final type of an expression when a new environment is reached. In the example when we evaluate V_0 with Γ_0 we receive `any()` as type for the variable, but after evaluating $Sum(V_0,V_1)$ we get some new information about that variable. Without the [TRANS] rule, the final type would be like: `(number())` $\rightarrow$ `{any(),number()}`, but with this rule we can reanalyze the expression $\{V_0, Sum(V_0,V_1)\}$ under an environment in which V_0 is known to have type `number()`.

5.2 Once upon a map

In our second example we will be using the function Map with an arithmetic function which takes a number and multiplies it by two. We will use the variable Mul as the function `'erlang':'*'`, which will be given in the initial environment Γ_0. The code in our subset of Core Erlang is:

letrec $Map = \textbf{fun}(V_1,V_2) \rightarrow$ **case** V_2 **of**

`[]` **when** `'true'` $\rightarrow$ `[]`

$[X \mid XS]$ **when** `'true'` $\rightarrow$ **let** $V_3 = V_1(X)$ **in let** $V_4 = Map(V_1,XS)$ **in** $[V_3 \mid V_4]$ **end**

$Foo = \textbf{fun}(V_0) \rightarrow$ **let** $V_5 = \textbf{fun}(V_6) \rightarrow$ **let** $V_7 = 2$ **in** $Mul(V_6,V_7)$ **in** $Map(V_5,V_0)$

in Foo

The type `([any()])` $\rightarrow$ `[any()]` is obtained for the lambda abstraction Foo with our derivation. The whole derivation can be found at an appendix at the end of the paper, but in this section can be seen an excerpt of the derivation with the most meaningful rules we have used, where Γ_0 contains the type for Mul:

$$\frac{\begin{array}{c}\Gamma_0 = [Mul:(\texttt{number()},\texttt{number()}) \xrightarrow{[]} \texttt{number()}] \\ \Gamma_1 = \Gamma_0 \sqcap [Map:(\texttt{any()},\texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]}; Foo:(\texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]}] \\ (1) \qquad \Gamma_1 \vdash \textbf{fun}(V_1,V_2) \rightarrow \cdots : (\texttt{any()},\texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]},\Gamma_1 \\ (2) \qquad \Gamma_1 \vdash \textbf{fun}(V_0) \rightarrow \cdots : (\texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]},\Gamma_1 \\ \Gamma_1 \vdash Foo:(\texttt{[any()]}) \xrightarrow{[]} \texttt{[any()]},\Gamma_1\end{array}}{\Gamma_0 \vdash \textbf{letrec}\ Map = \ldots Foo = \ldots \textbf{in} \cdots : (\texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]},\Gamma_0} \text{ [LETREC]}$$

$$\frac{\begin{array}{c}(3)\quad \Gamma_1 \vdash \textbf{case}\ V_2\ \textbf{of}\ldots\textbf{end} : \texttt{[any()]}, (\Gamma_2 = \Gamma_1 \sqcap [V_1 : \texttt{any()}; V_2 : \texttt{[any()]}]) \\ \Gamma_2 \backslash \{V_1, V_2\} = \Gamma_1 \qquad \texttt{[any()]} \backslash \{V_1, V_2\} = \texttt{[any()]} \\ \hline (1)\quad \Gamma_1 \vdash \textbf{fun}(V_1, V_2) \rightarrow \cdots : (\texttt{any()}, \texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]}, \Gamma_1\end{array}}{}\ [\text{ABS}]$$

$$\frac{\begin{array}{c}\Gamma_3 = \Gamma_1 \sqcap [V_1 : \texttt{any()}; V_2 : \texttt{[]}] \\ \Gamma_4 = \Gamma_1 \sqcap [V_1 : (\texttt{any()}) \xrightarrow{[]} \texttt{any()}; V_2 : \texttt{[any()]}; X : \texttt{any()}; XS : \texttt{[any()]}] \\ (4)\quad \Gamma_{1\{V_2\}} \Vdash \texttt{[]}\ \textbf{when}\ \texttt{'true'} \rightarrow \texttt{[]} : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \\ (5)\quad \Gamma_{1\{V_2\}} \Vdash \texttt{[}X \mid XS\texttt{]}\ \textbf{when}\ \texttt{'true'} \rightarrow \cdots : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \\ \hline (3)\quad \Gamma_1 \vdash \textbf{case}\ V_2\ \textbf{of}\ldots\textbf{end} : \texttt{[any()]}, \Gamma_2\end{array}}{}\ [\text{CASE}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \texttt{[]} : \texttt{[]}, \Gamma_1 \qquad \Gamma_1 \sqcap [V_2 : \texttt{[]}] \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_3 \\ \texttt{'true'} \sqcap true \neq \texttt{none()} \qquad \Gamma_3 \vdash \texttt{[]} : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \\ \hline (4)\quad \Gamma_{1\{V_2\}} \Vdash \texttt{[]}\ \textbf{when}\ \texttt{'true'} \rightarrow \texttt{[]} : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4\end{array}}{}\ [\text{CLS-T}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \texttt{[X|XS]} : \texttt{[any()|any()]}, \Gamma_1 \\ (\Gamma_1 \sqcap [V_2 : \texttt{[any()|any()]}] = \Gamma_5) \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_5 \\ \texttt{'true'} \sqcap true \neq \texttt{none()} \qquad \Gamma_5 \vdash \textbf{let}\ V_3 = \ldots\textbf{in}\cdots : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4 \quad (6) \\ \hline (5)\quad \Gamma_{1\{V_2\}} \Vdash \texttt{[}X \mid XS\texttt{]}\ \textbf{when}\ \texttt{'true'} \rightarrow \cdots : \texttt{[any()]}, \Gamma_3 \sqcup \Gamma_4\end{array}}{}\ [\text{CLS-T}]$$

$$\cdots$$

$$\frac{\begin{array}{c}(11)\quad \Gamma_1 \vdash \textbf{let}\ V_5 = \ldots\textbf{in}\cdots : \texttt{[any()]}, (\Gamma_9 = \Gamma_1 \sqcap [V_0 : \texttt{[any()]}]) \\ \Gamma_9 \backslash \{V_0\} = \Gamma_1 \qquad \texttt{[any()]} \backslash \{V_0\} = \texttt{[any()]} \\ \hline (2)\quad \Gamma_1 \vdash \textbf{fun}(V_0) \rightarrow \cdots : (\texttt{[any()]}) \xrightarrow{\Gamma_1} \texttt{[any()]}, \Gamma_1\end{array}}{}\ [\text{ABS}]$$

$$\frac{\begin{array}{c}(12)\quad \Gamma_1 \vdash \textbf{fun}(V_6) \rightarrow \cdots : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}, \Gamma_1 \\ (13)\quad (\Gamma_1 \sqcap [V_5 : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}] = \Gamma_{10}) \vdash Map(V_5, V_0) : \\ \texttt{[any()]}, (\Gamma_{11} = \Gamma_9 \sqcap [V_5 : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}]) \\ \hline (11)\quad \Gamma_1 \vdash \textbf{let}\ V_5 = \ldots\textbf{in}\cdots : \texttt{[any()]}, \Gamma_9\end{array}}{}\ [\text{LET}]$$

$$\cdots$$

$$\frac{\Gamma_{10}(Map) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{any()}, \texttt{[any()]}) \xrightarrow{\Gamma_0} \texttt{[any()]}}{(13)\quad \Gamma_{10} \vdash Map(V_5, V_0) : \texttt{[any()]}, \Gamma_{11}}\ [\text{APP1}]$$

In this example we show the derivation from the topmost expression downwards. Most of the judgements in the derivation are identified with a number, in order to make them easier to track throughout the whole derivation. In the [CASE] rule of the judgement (3) we can see that Γ_3 is the environment obtained from the first clause and Γ_4 from the second one, but to use the rule we must unify the final environments and the types. To make the unification possible we use the rules [SUB-2] on the expression of each clause and [SUB-T], in order to get the same type in both clauses.

5.3 Improving the obtained success types

In a monomorphic type system the success types we can obtain for functions like Map are a little far away from what would be desired. The cause of these unwanted results is that the input types are not connected with the output types in the function type. There are several solutions to improve this situation. One of them consists in reanalyzing the type of a function definition for each of its applications. In each of these analysis, we inject the types of the input arguments into the assumption environment.

Each time we use the rule [APP1] to analyze a function application, we can try to derive again the type of

the function definition, provided we have its code. In this new derivation, the assumption environment maps the input arguments to the types of the actual arguments in the function application. Thus we obtain a more precise signature for that function. In our Map example, a first analysis of Map would yield a final environment $\Gamma_2 = \Gamma_1 \sqcap [V_1 : \texttt{any()}, V_2 : [\texttt{any()}]]$ but inside the definition of Foo, the Map function is applied to V_5 (of type $(\texttt{number()}) \xrightarrow{[]} \texttt{number()}$) and V_0 (of type $\texttt{any()}$), so we would reanalyze the body of the definition of Map under the assumption environment $\Gamma_2 \sqcap [V_1 : (\texttt{number()}) \xrightarrow{[]} \texttt{number()}, V_2 : \texttt{any()}]$, so as to get the signature $((\texttt{number()}) \rightarrow \texttt{number()}, [\texttt{number()}]) \rightarrow [\texttt{number()}]$. Then we can update the type of the application $Map(V_5, V_0)$ to infer that V_0 must have type $[\texttt{number()}]$.

This approach, which is roughly equivalent to "inlining" the function definition in each application, changes the final type of the Foo function into $([\texttt{number()}]) \rightarrow [\texttt{number()}]$, instead of $([\texttt{any()}]) \rightarrow [\texttt{any()}]$ which is the success type we had reached in our derivation. The advantage of this refinement is that we reach more precise success types when using polymorphic functions, but its disadvantage is that its computational cost increases, since we have to reanalyze a function in each of its applications.

Another solution is reached by the tool *Dialyzer*, which is based on the notion of *refined success types* [3]. In our Map example, this refinement is applied depending on whether the Map function is exported or not outside the module. But even in the case where Map is not exported, the V_2 variable does not change its type from $[\texttt{any()}]$ to $[\texttt{number()}]$, so Dialyzer ends up inferring $([\texttt{any()}]) \rightarrow [\texttt{number()}]$ as the final type of Foo.

Even if this inlining technique yields more accurate results, it is still better to consider the use of polymorphic types, which connect the input types in the parameters with the output type, without any significant computational overhead. The extension of this type system to support polymorphic types is subject of future work.

6 Correctness results

Before giving a proof of the correctness of the type system, we shall give some auxiliary definitions and results that will be needed in it. Firstly, we say that a variable is *unrestricted* in a type environment Γ if it has type $\texttt{any()}$ in Γ and it is recursively unrestricted in the environments occurring in the functional types mapped to by Γ. That is, x is unrestricted in Γ if and only if $\Gamma \setminus x = \Gamma$. Similarly, we say that x is unrestricted in τ iff $\tau \setminus x = \tau$.

The first proposition states that, although the semantics of a type τ depends on a substitution θ, the unrestricted variables in τ are irrelevant in this substitution when determining the semantics of τ.

Proposition 1. *Given a substitution θ, a type τ, an environment Γ, and a finite set of pairs (x_1, v_1), ..., (x_n, v_n) such that the variables $x_1, \ldots, x_n$ are unrestricted in Γ and τ, it holds that:*

1. $\mathcal{T}_\theta [\![\tau]\!] = \mathcal{T}_{\theta[\overline{x_i / v_i}]} [\![\tau]\!]$.
2. $\theta \in \mathcal{T}_{Env} [\![\Gamma]\!] \Leftrightarrow \theta[\overline{x_i / v_i}] \in \mathcal{T}_{Env} [\![\Gamma]\!]$.

Proof. By induction on the structure of Γ and τ. The cases in which τ is $\texttt{none()}$, $\texttt{any()}$, a basic type, or a singleton type are trivial, since the substitution θ plays no role in the semantics of these types. If τ is a tuple type, a list type or an union type, the theorem follows directly from applying the induction hypothesis, so let us consider the case of functional types and environments.

If τ is a type of the form $(\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma} \tau'$ assume that $\theta \notin \mathcal{T}_{Env} [\![\Gamma]\!]$. Then, by induction hypothesis, we get $\theta[\overline{x_i / v_i}] \notin \mathcal{T}_{Env} [\![\Gamma]\!]$ so in this case $\mathcal{T}_\theta \left[\!\!\left[(\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma} \tau' \right]\!\!\right] = \{\emptyset\} = \mathcal{T}_{\theta[\overline{x_i / v_i}]} \left[\!\!\left[(\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma} \tau' \right]\!\!\right]$. On the other hand, if θ

belongs to $\mathcal{T}_{Env}[\![\Gamma]\!]$, then so does $\theta[\overline{x_i/v_i}]$ and the proposition follows from applying the induction hypothesis to each of the components $\tau_1,\ldots,\tau_n,\tau'$ of the functional type.

In the case of environments, we get:

$$
\begin{array}{ll}
& \theta \in \mathcal{T}_{Env}[\![\Gamma]\!] \\
\Leftrightarrow & \\
& \forall x \in Var : \theta(x) \in \mathcal{T}_{\theta}[\![\Gamma(x)]\!] \\
\Leftrightarrow & \quad \{ \text{ by induction hypothesis } \} \\
& \forall x \in Var : \theta(x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma(x)]\!] \\
\Leftrightarrow & \\
& \left(\forall x \in Var\backslash\{\overline{x_i}\} : \theta(x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma(x)]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}\} : \theta(x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma(x)]\!]\right) \\
\Leftrightarrow & \quad \{ \text{ since the } \overline{x_i} \text{ are unrestricted in } \Gamma \} \\
& \left(\forall x \in Var\backslash\{\overline{x_i}\} : \theta[\overline{x_i/v_i}](x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma(x)]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}\} : \theta(x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\texttt{any()}]\!]\right) \\
\Leftrightarrow & \quad \{ \text{ since every value belongs to the semantics of } \texttt{any()} \} \\
& \left(\forall x \in Var\backslash\{\overline{x_i}\} : \theta[\overline{x_i/v_i}](x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma(x)]\!]\right) \wedge \left(\forall x \in \{\overline{x_i}\} : \theta[\overline{x_i/v_i}](x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\texttt{any()}]\!]\right) \\
\Leftrightarrow & \\
& \forall x \in Var : \theta[\overline{x_i/v_i}](x) \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma(x)]\!] \\
\Leftrightarrow & \\
& \theta[\overline{x_i/v_i}] \in \mathcal{T}_{Env}[\![\Gamma]\!]
\end{array}
$$

□

The following result shows that, whenever we discard the types of some variables in a type or an environment via the notation $\tau\backslash\{x_1,\ldots,x_n\}$ or $\Gamma\backslash\{x_1,\ldots,x_n\}$, we obtain types or environments with equal or greater semantics.

Proposition 2. *Given a substitution θ and a finite set of pairs $(x_1,v_1),\ldots,(x_n,v_n)$:*

1. $\mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\tau]\!] \subseteq \mathcal{T}_{\theta}[\![\tau\backslash\{x_1,\ldots,x_n\}]\!]$ *for every type τ.*
2. $\theta[\overline{x_i/v_i}] \in \mathcal{T}_{Env}[\![\Gamma]\!] \Rightarrow \theta \in \mathcal{T}_{Env}[\![\Gamma\backslash\{x_1,\ldots,x_n\}]\!]$ *for every type environment Γ.*

Proof. By induction on the structure of τ and Γ. Again, all cases are straightforward except the cases of a functional type and an environment.

Assume that τ is of the form $(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau'$. If $\theta[\overline{x_i/v_i}] \notin \mathcal{T}_{Env}[\![\Gamma]\!]$ then $\mathcal{T}_{\theta[\overline{x_i/v_i}]}\left[\!\left[(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau'\right]\!\right]$ has a single element: the empty function $\emptyset$, which trivially belongs to $\mathcal{T}_{\theta}\left[\!\left[\left((\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma} \tau'\right)\backslash\{x_1,\ldots,x_n\}\right]\!\right]$. On the other hand,

if $\theta[\overline{x_i / v_i}]$ belongs to $\mathcal{T}_{Env}[\![\Gamma]\!]$, then by induction hypothesis it follows that $\theta \in \mathcal{T}_{Env}[\![\Gamma \backslash \{x_1, \ldots, x_n\}]\!]$ and therefore:

$$
\begin{array}{ll}
& \mathcal{T}_{\theta[\overline{x_i / v_i}]}\left[\!\left[(\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma} \tau'\right]\!\right] \\
= & \quad \{ \text{ since } \theta[\overline{x_i / v_i}] \in \mathcal{T}_{Env}[\![\Gamma]\!] \} \\
& \left\{F \mid F \subseteq \left\{((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathcal{T}_{\theta[\overline{x_i / v_i}]}[\![\tau_i]\!], v \in \mathcal{T}_{\theta[\overline{x_i / v_i}]}[\![\tau]\!]\right\}\right\} \\
\subseteq & \quad \{ \text{ by induction hypothesis } \} \\
& \left\{F \mid F \subseteq \left\{((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathcal{T}_{\theta}[\![\tau_i \backslash \{x_1, \ldots, x_n\}]\!], v \in \mathcal{T}_{\theta}[\![\tau \backslash \{x_1, \ldots, x_n\}]\!]\right\}\right\} \\
= & \quad \{ \text{ since } \theta \in \mathcal{T}_{Env}[\![\Gamma \backslash \{x_1, \ldots, x_n\}]\!] \} \\
& \mathcal{T}_{\theta}\left[\!\left[\left((\tau_1, \ldots, \tau_n) \xrightarrow{\Gamma} \tau'\right) \backslash \{x_1, \ldots, x_n\}\right]\!\right]
\end{array}
$$

In the case of an environment Γ, we get the following:

$$
\begin{array}{ll}
& \theta[\overline{x_i / v_i}] \in \mathcal{T}_{Env}[\![\Gamma]\!] \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathcal{T}_{Env}[\![\Gamma]\!] \} \\
& \forall x \in Var : \theta[\overline{x_i / v_i}](x) \in \mathcal{T}_{\theta[\overline{x_i / v_i}]}[\![\Gamma(x)]\!] \\
\Rightarrow & \quad \{ \text{ by induction hypothesis } \} \\
& \forall x \in Var : \theta[\overline{x_i / v_i}](x) \in \mathcal{T}_{\theta}[\![\Gamma(x) \backslash \{x_1, \ldots, x_n\}]\!] \\
\Leftrightarrow & \\
& \left(\forall x \in Var \backslash \{x_1, \ldots, x_n\} : \theta[\overline{x_i / v_i}](x) \in \mathcal{T}_{\theta}[\![\Gamma(x) \backslash \{x_1, \ldots, x_n\}]\!]\right) \\
& \wedge \left(\forall x \in \{x_1, \ldots, x_n\} : \theta[\overline{x_i / v_i}](x) \in \mathcal{T}_{\theta}[\![\Gamma(x) \backslash \{x_1, \ldots, x_n\}]\!]\right) \\
\Leftrightarrow & \quad \{ \text{ since the } x_i \text{ are unrestricted in } \Gamma(x) \backslash \{x_1, \ldots, x_n\} \} \\
& (\forall x \in Var \backslash \{x_1, \ldots, x_n\} : \theta(x) \in \mathcal{T}_{\theta}[\![\Gamma(x) \backslash \{x_1, \ldots, x_n\}]\!]) \\
& \wedge \left(\forall x \in \{x_1, \ldots, x_n\} : \theta[\overline{x_i / v_i}](x) \in \mathcal{T}_{\theta}[\![\texttt{any}()]\!]\right) \\
\Rightarrow & \quad \{ \text{ since } \texttt{any}() \text{ contains all the values } \} \\
& (\forall x \in Var \backslash \{x_1, \ldots, x_n\} : \theta(x) \in \mathcal{T}_{\theta}[\![\Gamma(x) \backslash \{x_1, \ldots, x_n\}]\!]) \\
& \wedge \left(\forall x \in \{x_1, \ldots, x_n\} : \theta(x) \in \mathcal{T}_{\theta}[\![\texttt{any}()]\!]\right) \\
\Rightarrow & \quad \{ \text{ since the } x_i \text{ are unrestricted in } \Gamma(x) \backslash \{x_1, \ldots, x_n\} \} \\
& \forall x \in Var : \theta(x) \in \mathcal{T}_{\theta}[\![\Gamma(x) \backslash \{x_1, \ldots, x_n\}]\!] \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathcal{T}_{Env}[\![\Gamma \backslash \{x_1, \ldots, x_n\}]\!] \} \\
& \theta \in \mathcal{T}_{Env}[\![\Gamma \backslash \{x_1, \ldots, x_n\}]\!]
\end{array}
$$

□

An important property of the typing rules is that, besides the constraints already defined in the assumption environment, the final environment does not pose further restrictions on those variables that not occur free in the expression being typed, unless the final environment is $\bot$.

Proposition 3. *Assume a judgement $\Gamma \vdash e : \tau, \Gamma'$. For every variable x, if x is unrestricted in Γ and it does not appear free in e, then either Γ' is $\bot$ or x is unrestricted in τ and Γ'. The same applies to judgements of the form $\Gamma \Vdash_X cls : \tau, \Gamma'$ for those variables distinct from the pattern of cls, unrestricted in Γ and which do not appear free in the body of the clause.*

Proof. By induction on the typing derivation. In the case of the subtyping rules, if x is unrestricted in Γ, so will be in every environment $\Gamma'' \supseteq \Gamma$ and in every type $\tau'' \supseteq \tau$. In the case of the [NONE], [APP-2] and [RECEIVE-3] rules, we have that $\Gamma' = \bot$. In the remaining cases, the result follows directly from applying the induction hypothesis on the subderivations and from the fact that removing the **let**-bound variables and pattern variables from types and environments makes them unrestricted. □

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, and θ is contained within the final environment Γ'. A similar result can be stated for $\Vdash$-judgements. Both facts must be proved simultaneously, due to the mutual dependence between both set of rules.

Theorem 1. *Let us assume typing environments Γ and Γ', a type τ, an expression e, a clause cls and a set X of variables. Then the following two statements hold:*

1. *If $\Gamma \vdash e : \tau, \Gamma'$, then $\mathcal{E}[\![e]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$.*
2. *If $\Gamma \Vdash_X cls : \tau, \Gamma'$, then*

$$\{(\theta, v) \in \mathcal{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$$

for any $V \subseteq DVal$.

Proof. Given some environments Γ and Γ' an expression e, and a type τ, assume that $\Gamma \vdash e : \tau, \Gamma'$. We prove that $\mathcal{E}[\![e]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$ by induction on the size of the type derivation. We distinguish cases on the last rule applied.

- **Case [SUB-1]**

  We know that there exists some Γ'' such that $\Gamma \subseteq \Gamma''$ and $\Gamma'' \vdash e : \tau, \Gamma'$. Therefore:

$$\begin{array}{lll} & \mathcal{E}[\![e]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathcal{E}[\![e]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]} & \{ \text{ since } \Gamma \subseteq \Gamma'' \} \\ \subseteq & \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \{ \text{ by induction hypothesis } \} \end{array}$$

- **Case [SUB-2]**

  In this case we have some Γ'' such that $\Gamma'' \subseteq \Gamma'$ and $\Gamma \vdash e : \tau, \Gamma''$. So we get:

$$\begin{array}{lll} & \mathcal{E}[\![e]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma'']\!]} & \{ \text{ by induction hypothesis } \} \\ \subseteq & \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \{ \text{ since } \Gamma'' \subseteq \Gamma' \} \end{array}$$

- **Case [SUB-T]**

  There exists some τ' such that $\tau' \subseteq \tau$ and $\Gamma \vdash e : \tau', \Gamma'$. Therefore:

$$\begin{array}{lll} & \mathscr{E}[\![e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathscr{T}[\![\tau']\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]} & \{\text{ by induction hypothesis }\} \\ \subseteq & \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]} & \{\text{ since } \tau' \subseteq \tau\ \} \end{array}$$

- **Case [NONE]**

  Given any environment Γ such that $\Gamma(x) = \texttt{none()}$ for some $x \in \mathit{Var}$, we know that $\mathscr{T}_{Env}[\![\Gamma]\!] = \emptyset$. In particular, this is what would happen if we applied the [NONE] rule, since $\Gamma = \Gamma' \sqcap [x : \texttt{none()}]$ for some $x \in \mathit{Var}$ and therefore $\Gamma(x) = \Gamma'(x) \sqcap \texttt{none()} = \texttt{none()}$, which makes $\mathscr{T}_{Env}[\![\Gamma]\!]$ empty. Therefore:

$$\mathscr{E}[\![e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} = \emptyset = \mathscr{T}[\![\texttt{none()}]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\bot]\!]}$$

- **Case [TRANS]**

  We know that there exist an intermediate environment Γ'' and an intermediate type τ' such that $\Gamma \vdash e : \tau', \Gamma''$ and $\Gamma'' \vdash e : \tau, \Gamma'$. Now, let us assume a substitution θ and a value v such that $(\theta, v) \in \mathscr{E}[\![e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. That is, $(\theta, v) \in \mathscr{E}[\![e]\!]^\star$ where $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. By induction hypothesis we can ensure that $\theta \in \mathscr{T}_{Env}[\![\Gamma'']\!]$. Therefore, $(\theta, v) \in \mathscr{E}[\![e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma'']\!]}$ and we can apply induction hypothesis again in order to obtain $(\theta, v) \in \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$.

- **Case [CONST]**

  In this case we get:

$$\begin{array}{lll} & \mathscr{E}[\![c]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ = & \{(\theta, c) \mid \theta \in \mathbf{Subst}\} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ = & \{(\theta, v) \mid \theta \in \mathbf{Subst}, v \in \{c\}\} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ = & \left\{(\theta, v) \mid \theta \in \mathbf{Subst}, v \in \mathscr{T}_\theta[\![c]\!]\right\} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ = & \mathscr{T}[\![c]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \end{array}$$

- **Case [VAR]**

  We obtain:

$$\begin{array}{lll} & \mathscr{E}[\![x]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \\ = & \{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], v = \theta(x)\} & \\ \subseteq & \quad \{\text{ since } \theta \in \mathscr{T}_{Env}[\![\Gamma]\!] \text{ implies that } \theta(x) \in \mathscr{T}_\theta[\![\Gamma(x)]\!]\ \} & \\ & \left\{(\theta, v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], v \in \mathscr{T}_\theta[\![\Gamma(x)]\!]\right\} & \\ = & \mathscr{T}[\![\Gamma(x)]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} & \end{array}$$

- **Case [TUPLE]**

For each $i \in \{1..n\}$, we know that there exists a derivation of $\Gamma \vdash e_i : \tau_i ; \Gamma_i$ for some τ_i and Γ_i.

$$
\begin{array}{ll}
 & \mathscr{E}[\![\{e_1,\ldots,e_n\}]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\
= & \left\{(\theta,(\{.\overset{n}{.}.\},v_1,\ldots,v_n)) \mid \forall i \in \{1..n\}.\theta \in \mathscr{T}_{Env}[\![\Gamma]\!], (\theta,v_i) \in \mathscr{E}[\![e_i]\!]^\star\right\} \\
\subseteq & \qquad \{ \text{by i.h: } \mathscr{E}[\![e_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]} \} \\
 & \left\{(\theta,(\{.\overset{n}{.}.\},v_1,\ldots,v_n)) \mid \forall i \in \{1..n\}.\theta \in \mathscr{T}_{Env}[\![\Gamma_i]\!], (\theta,v_i) \in \mathscr{T}[\![\tau_i]\!]^\star\right\} \\
= & \qquad \{ \text{since } \bigcap_{i=1}^n \mathscr{T}_{Env}[\![\Gamma_i]\!] = \mathscr{T}_{Env}\left[\!\left[\prod_{i=1}^n \Gamma_i\right]\!\right] \text{ by Proposition } \textbf{??} \} \\
 & \left\{(\theta,(\{.\overset{n}{.}.\},v_1,\ldots,v_n)) \mid \theta \in \mathscr{T}_{Env}\left[\!\left[\prod_{i=1}^n \Gamma_i\right]\!\right], \forall i \in \{1..n\}.(\theta,v_i) \in \mathscr{T}[\![\tau_i]\!]^\star\right\} \\
= & \left\{(\theta,v) \mid \theta \in \mathscr{T}_{Env}\left[\!\left[\prod_{i=1}^n \Gamma_i\right]\!\right], (\theta,v) \in \mathscr{T}[\![\{\tau_1,\ldots,\tau_n\}]\!]^\star\right\} \\
= & \mathscr{T}[\![\{\tau_1,\ldots,\tau_n\}]\!]^\star \restriction_{\mathscr{T}_{Env}\left[\!\left[\prod_{i=1}^n \Gamma_i\right]\!\right]}
\end{array}
$$

- **Case [LIST]**

  Similarly to the [TUPLE] case, we assume that $e = [e_1 \mid e_2]$ for some expressions e_1 and e_2, and there exists a type derivation of each subexpression: $\Gamma \vdash e_1 : \tau_1, \Gamma_1$ and $\Gamma \vdash e_2 : \tau_2, \Gamma_2$.

$$
\begin{array}{ll}
 & \mathscr{E}[\![[e_1 \mid e_2]]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\
= & \left\{(\theta,([\_\mid\_],v_1,v_2)) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], (\theta,v_1) \in \mathscr{E}[\![e_1]\!]^\star, (\theta,v_2) \in \mathscr{E}[\![e_2]\!]^\star\right\} \\
\subseteq & \qquad \{ \text{by i.h: } \mathscr{E}[\![e_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]} \} \\
 & \left\{(\theta,([\_\mid\_],v_1,v_2)) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_1]\!], \theta \in \mathscr{T}_{Env}[\![\Gamma_2]\!], (\theta,v_1) \in \mathscr{T}[\![\tau_1]\!]^\star, (\theta,v_2) \in \mathscr{T}[\![\tau_2]\!]^\star\right\} \\
= & \qquad \{ \text{since } \mathscr{T}_{Env}[\![\Gamma_1]\!] \cap \mathscr{T}_{Env}[\![\Gamma_2]\!] = \mathscr{T}[\![\Gamma_1 \sqcap \Gamma_2]\!]^\star \text{ by Proposition } \textbf{??} \} \\
 & \left\{(\theta,([\_\mid\_],v_1,v_2)) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!], (\theta,v_1) \in \mathscr{T}[\![\tau_1]\!]^\star, (\theta,v_2) \in \mathscr{T}[\![\tau_2]\!]^\star\right\} \\
= & \left\{(\theta,v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!], (\theta,v) \in \mathscr{T}[\![nelist(\tau_1,\tau_2)]\!]^\star\right\} \\
= & \mathscr{T}[\![nelist(\tau_1,\tau_2)]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!]}
\end{array}
$$

- **Case [ABS]**

  We assume that the expression being typed is of the form $\textbf{fun}(x_1,\ldots,x_n) \to e_f$, and that we have a derivation for $\Gamma \vdash e_f : \tau' : \Gamma''$ for some type τ' and environment Γ''. Let us assume that $(\theta,F) \in \mathscr{E}[\![\textbf{fun}(x_1,\ldots,x_n) \to e_f]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. This implies that $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$ and

$$F = \left\{((v_1,\ldots,v_n),v) \mid v_1,\ldots,v_n \in DVal, (\theta[\overline{x_i/v_i}],v) \in \mathscr{E}[\![e]\!]^\star\right\} \tag{2}$$

  We assume that $x_1,\ldots,x_n$ are unrestricted in Γ. Otherwise we could have applied renaming in order to obtain distinct bound variables throughout the whole expression. From this assumption it follows by Proposition 1 that $\theta[\overline{x_i/v_i}] \in \mathscr{T}_{Env}[\![\Gamma]\!]$, which leads to:

$$(\theta[\overline{x_i/v_i}],v) \in \mathscr{E}[\![e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau']\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma'']\!]}$$

  where the latter inclusion is given by induction hypothesis. We can rewrite (2) as follows,

$$F \subseteq \left\{((v_1,\ldots,v_n),v) \mid v_1,\ldots,v_n \in DVal, (\theta[\overline{x_i/v_i}],v) \in \mathscr{T}[\![\tau']\!]^\star\right\}$$

and it holds that $\theta[\overline{x_i/v_i}] \in \mathcal{T}_{Env}[\![\Gamma'']\!]$, this implies that $v_j \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma''(x_j)]\!]$ for each $j \in \{1..n\}$. Therefore:

$$F \subseteq \left\{((v_1,\ldots,v_n),v) \mid \forall j.v_j \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\Gamma''(x_j)]\!], v \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}[\![\tau']\!]\right\}$$

By the definition of the semantics of a functional type, it holds that

$$F \in \mathcal{T}_{\theta[\overline{x_i/v_i}]}\left[\!\!\left[(\Gamma''(x_1),\ldots,\Gamma''(x_n)) \xrightarrow{\Gamma''} \tau'\right]\!\!\right]$$

or, equivalently,

$$(\theta[\overline{x_i/v_i}],F) \in \mathcal{T}\left[\!\!\left[(\Gamma''(x_1),\ldots,\Gamma''(x_n)) \xrightarrow{\Gamma''} \tau'\right]\!\!\right]^\star.$$

We apply Proposition 2,

$$(\theta,F) \in \mathcal{T}\left[\!\!\left[\left((\Gamma''(x_1),\ldots,\Gamma''(x_n)) \xrightarrow{\Gamma''} \tau'\right)\setminus\{x_1,\ldots,x_n\}\right]\!\!\right]^\star$$

and, since we know that $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$,

$$(\theta,F) \in \mathcal{T}\left[\!\!\left[\left((\Gamma''(x_1),\ldots,\Gamma''(x_n)) \xrightarrow{\Gamma''} \tau'\right)\setminus\{x_1,\ldots,x_n\}\right]\!\!\right]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$$

- **Case [APP1]**

  Assume that $e = f(x_1,\ldots,x_n)$, that $\Gamma(f) \sqcap ((\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \to \texttt{any}()) = (\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma''} \tau'$ for some $\tau_1,\ldots,\tau_n,\tau'$ and Γ''. This rule specifies that $\Gamma' = \Gamma \sqcap [f : (\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \to \texttt{any}(), x_1 : \tau_1,\ldots,x_n : \tau_n] \sqcap \Gamma''$. We have:

  $$\mathcal{E}[\![f(x_1,\ldots,x_n)]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} = \left\{(\theta,v) \mid \theta \in \mathcal{T}_{Env}[\![\Gamma]\!], ((\theta(x_1),\ldots,\theta(x_n)),v) \in \theta(f)\right\}$$

  Assume a substitution θ and a value v such that $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ and $(\theta,v) \in \mathcal{E}[\![e]\!]^\star$. Since $\theta(f)$ is an n-ary function (otherwise there would not be any v such that (θ,v) belongs to the above mentioned set) we know that $\theta(f) \in \mathcal{T}_\theta\left[\!\!\left[(\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \xrightarrow{[]} \texttt{any}()\right]\!\!\right]$. Together with the fact that $\theta(f) \in \mathcal{T}_\theta[\![\Gamma(f)]\!]$, it holds that $\theta(f) \in \mathcal{T}_\theta\left[\!\!\left[(\tau_1,\ldots,\tau_n) \xrightarrow{\Gamma''} \tau\right]\!\!\right]$. Moreover, since $((\theta(x_1),\ldots,\theta(x_n)),v) \in \theta(f)$, it holds that:

  1. $\theta \in \mathcal{T}_{Env}[\![\Gamma'']\!]$. Otherwise $\theta(f)$ would be empty.
  2. For every $i \in \{1..n\}$, $\theta(x_i) \in \mathcal{T}_\theta[\![\tau_i]\!]$, so $\theta \in \mathcal{T}_{Env}[\![[x_i : \tau_i]]\!]$.
  3. $\theta \in \mathcal{T}_{Env}\left[\!\!\left[f : (\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \xrightarrow{[]} \texttt{any}()\right]\!\!\right]$.
  4. $v \in \mathcal{T}_\theta[\![\tau]\!]$.

  Therefore $\theta \in \mathcal{T}_{Env}\left[\!\!\left[\Gamma \sqcap [f : (\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \xrightarrow{[]} \texttt{any}(), x_1 : \tau_1,\ldots,x_n : \tau_n] \sqcap \Gamma''\right]\!\!\right] = \mathcal{T}_{Env}[\![\Gamma']\!]$ and $(\theta,v) \in \mathcal{T}[\![\tau]\!]^\star$, from which it follows that:

  $$(\theta,v) \in \mathcal{T}[\![\tau]\!]^\star \restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]}$$

- **Case [APP2]**

Again, we have

$$\mathscr{E}[\![f(x_1,\ldots,x_n)]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} = \{(\theta,v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], ((\theta(x_1),\ldots,\theta(x_n)),v) \in \theta(f)\}$$

We prove by contradiction that this set is empty. Assume that $(\theta,v) \in \mathscr{E}[\![f(x_1,\ldots,x_n)]\!]^\star$ for some $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. The existence of a tuple in this set implies the existence of a tuple $((\theta(x_1),\ldots,\theta(x_n)),v)$ in $\theta(f)$, so the latter is an n-ary function. We obtain, on the one hand, that $\theta(f) \in \mathscr{T}_\theta[\![\Gamma(f)]\!]$ and, on the other hand, we know that $\theta(f) \in \mathscr{T}_\theta\left[\!\!\left[(\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \xrightarrow{[]} \texttt{any}()\right]\!\!\right]$. Therefore:

$$\begin{aligned} \theta(f) &\subseteq \Gamma(f) \\ \theta(f) &\subseteq (\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \xrightarrow{[]} \texttt{any}() \\ \texttt{none}() &\subsetneq \theta(f) \end{aligned}$$

contradicting the fact that `none()` is the greatest lower bound of $\Gamma(f)$ and $(\texttt{any}(),\overset{n}{\cdots},\texttt{any}()) \xrightarrow{[]} \texttt{any}()$, as the assumption in [APP2] demands.

Now that we have proved that $\mathscr{E}[\![f(x_1,\ldots,x_n)]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$ is empty, the theorem follows trivially:

$$\mathscr{E}[\![f(x_1,\ldots,x_n)]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} = \emptyset \subseteq \mathscr{T}[\![\texttt{none}()]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\bot]\!]}$$

- **Case [LET]**

  In this case we know that $e = \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$ for some x_1, e_1, and e_2 and that $\Gamma \vdash e_1 : \tau_1, \Gamma_1$ and $\Gamma_1 \sqcap [x_1 : \tau_1] \vdash e_2 : \tau_2, \Gamma_2$ for some Γ_1, τ_1, Γ_2, τ_2, so $\tau = \tau_2 \backslash x_1$ and $\Gamma' = \Gamma_2 \backslash x_1$.

$$\begin{aligned} & \mathscr{E}[\![\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\ = \quad & \\ & \{(\theta,v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma]\!], (\theta,v_1) \in \mathscr{E}[\![e_1]\!]^\star, (\theta[x_1/v_1],v) \in \mathscr{E}[\![e_2]\!]^\star\} \\ \subseteq \quad & \{\text{ by i.h. }\} \\ & \{(\theta,v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_1]\!], (\theta,v_1) \in \mathscr{T}[\![\tau_1]\!]^\star, (\theta[x_1/v_1],v) \in \mathscr{E}[\![e_2]\!]^\star\} \\ \subseteq \quad & \{\text{ by Proposition 3 and Proposition 1 }\} \\ & \{(\theta,v) \mid \theta[x_1/v_1] \in \mathscr{T}_{Env}[\![\Gamma_1]\!], v_1 \in \mathscr{T}_{\theta[x_1/v_1]}[\![\tau_1]\!], (\theta[x_1/v_1],v) \in \mathscr{E}[\![e_2]\!]^\star\} \\ = \quad & \{\text{ since } x_1 \text{ has type } \texttt{any}() \text{ in } \Gamma_1 \} \\ & \{(\theta,v) \mid \theta[x_1/v_1] \in \mathscr{T}_{Env}[\![\Gamma_1 \sqcap [x_1 : \tau_1]]\!], (\theta[x_1/v_1],v) \in \mathscr{E}[\![e_2]\!]^\star\} \\ \subseteq \quad & \{\text{ by i.h. }\} \\ & \{(\theta,v) \mid \theta[x_1/v_1] \in \mathscr{T}_{Env}[\![\Gamma_2]\!], (\theta[x_1/v_1],v) \in \mathscr{T}[\![\tau_2]\!]^\star\} \\ \subseteq \quad & \{\text{ by Proposition 2 }\} \\ & \{(\theta,v) \mid \theta \in \mathscr{T}_{Env}[\![\Gamma_2 \backslash x_1]\!], (\theta,v) \in \mathscr{T}[\![\tau_2 \backslash x_1]\!]^\star\} \\ = \quad & \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]} \end{aligned}$$

- **Case [CASE]**

  We get $e = \textbf{case}\ x\ \textbf{of}\ cls_1 \ldots cls_n$ for some variable x and clauses $cls_1,\ldots,cls_n$. Moreover, we assume that each

clause cls_i ($i = \{1..n\}$) has the form p_i **when** $e_i \to e_i'$, where p_i is a pattern and e_i, e_i' are expressions. We get:

$$\begin{array}{rl} & \mathscr{E}[\![\textbf{case } x \textbf{ of } cls_1 \ldots cls_n]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \\ = & \bigcup_{i=1}^{n} \{(\theta, v) \mid v_1, \ldots, v_m \in DVal, (\theta[\overline{x_{ij}/v_j}], v) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}}, \\ & \qquad (\forall k < i. \forall \overline{v_j}'. \forall v'. (\theta[\overline{x_{kj}/v_j'}], v') \notin \mathscr{C}[\![cls_k]\!]_{\{\theta(x)\}})\} \\ \subseteq & \bigcup_{i=1}^{n} \left\{(\theta, v) \mid v_1, \ldots, v_m \in DVal, (\theta[\overline{x_{ij}/v_j}], v) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}}\right\} \end{array}$$

Assume a pair (θ, v) belonging to $\mathscr{E}[\![\textbf{case } x \textbf{ of } cls_1 \ldots cls_n]\!] \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. Then $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$ and there exists a $i \in \{1..n\}$ such that $(\theta[\overline{x_{ij}/v_j}], v) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}}$ for some values $\overline{v_j}$, where $\overline{x_{ij}}$ are the variables of the i-th pattern. We know that there exists a derivation of $\Gamma \Vdash_{\{x\}} cls_i : \tau_i, \Gamma_i$ for some τ_i and Γ_i. The induction hypothesis specifies that:

$$\left\{(\theta, v) \in \mathscr{C}[\![cls_i]\!]_V \mid \forall y \in \{x\}, \theta(y) \in V\right\} \subseteq \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$$

for any $V \subseteq DVal$. In particular, for $V = \{\theta(x)\}$ the condition $\forall y \in \{x\}, \theta(y) \in V$ can be rewritten as $\theta(x) = \theta(x)$ which holds trivially, so we can rewrite the induction hypothesis as follows:

$$\mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$$

Without loss of generality, we can assume that the pattern variables $\{\overline{x_{ij}}\}$ have type `any()` in Γ, since otherwise we could rename them in the corresponding clause. Therefore $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$ implies $\theta[\overline{x_{ij}/v_j}] \in \mathscr{T}_{Env}[\![\Gamma]\!]$, so $(\theta[\overline{x_{ij}/v_j}], v) \in \mathscr{C}[\![cls_i]\!]_{\{\theta(x)\}} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$ and we get $(\theta[\overline{x_{ij}/v_j}], v) \in \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$. If we denote by τ_i' (respectively Γ_i') the type $\tau_i \setminus \{\overline{x_{ij}}\}$ (respectively the environment $\Gamma' \setminus \{\overline{x_{ij}}\}$) we get, by Proposition 2 that $(\theta, v) \in \mathscr{T}[\![\tau_i']\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i']\!]}$. Finally, since it holds that

$$\tau_i' \subseteq \bigsqcup_{i=1}^{n} \tau_i' = \tau \quad \text{and} \quad \Gamma_i' \subseteq \bigsqcup_{i=1}^{n} \Gamma_i' = \Gamma',$$

we get $(\theta, v) \in \mathscr{T}[\![\tau']\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$, which proves the theorem.

- **Case [RECEIVE-1]**

  In this case we have an expression of the form **receive** $cls_1 \ldots cls_n$ **after** $e_t \to e$ for some clauses $cls_1, \ldots, cls_n$ and expressions e_t and e. Given (θ, v) in $\mathscr{E}[\![\textbf{receive } cls_1 \ldots cls_n \textbf{ after } e_t \to e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$ it holds that $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$, and we can distinguish cases:

  - **Case 1**. There exist some $v_1, \ldots, v_n, v_t \in DVal$ and some $i \in \{1..n\}$ such that the pair $(\theta[\overline{x_{ij}/v_i}], v) \in \mathscr{C}[\![cls_i]\!]_{DVal}$, $(\theta, v_t) \in \mathscr{E}[\![e_t]\!]^\star$ and $v_t \in Integer \cup \{infinity\}$, where $\overline{x_{ij}}$ are the variables in the pattern p_i of the clause cls_i. Since we have applied the [RECEIVE-1] rule, we have derived the judgements $\Gamma \vdash e_t : \tau_t, \Gamma_t$ and $\Gamma \Vdash_\emptyset cls_i : \tau_i, \Gamma_i$ for some τ_i and Γ_i. By applying the induction hypothesis to the first one it holds that $(\theta, v_t) \in \mathscr{T}[\![\tau_t]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]}$, so $\theta \in \mathscr{T}_{Env}[\![\Gamma_t]\!]$. The induction hypothesis corresponding to the second judgement takes the following form

    $$\left\{(\theta, v) \in \mathscr{C}[\![cls_i]\!]_{DVal} \mid \forall x \in \emptyset, \theta(x) \in DVal\right\} \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]} \subseteq \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$$

which is equivalent as the following statement:

$$\mathscr{C}[\![cls_i]\!]_{DVal} \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]} \subseteq \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$$

Moreover, we can assume that the pattern variables $\overline{x_{ij}}$ have type `any()` in Γ, since otherwise we could rename the clause. Thus $\theta[\overline{x_{ij}/v_i}] \in \mathscr{T}_{Env}[\![\Gamma_t]\!]$, and hence $(\theta[\overline{x_{ij}/v_i}], v) \in \mathscr{C}[\![cls_i]\!]_{DVal} \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]}$, from which it follows that $(\theta[\overline{x_{ij}/v_i}], v) \in \mathscr{T}[\![\tau_i]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i]\!]}$ and, by Proposition 2, (θ, v) belongs to the set $\mathscr{T}[\![\tau_i \setminus \{\overline{x_{ij}}\}]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_i \setminus \{\overline{x_{ij}}\}]\!]}$.

– **Case 2**. In this case $(\theta, v) \in \mathscr{E}[\![e]\!]^\star$ and there exists some $v_t \in \mathit{Integer}$ such that $(\theta, v_t) \in \mathscr{E}[\![e_t]\!]^\star$. As in the previous case, from applying the induction hypothesis to the judgement $\Gamma \vdash e_t : \tau_t : \Gamma_t$ it follows that $(\theta, v_t) \in \mathscr{T}[\![\tau_t]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]}$ and hence $\theta \in \mathscr{T}_{Env}[\![\Gamma_t]\!]$. Now we can apply again the induction hypothesis to the judgement $\Gamma_t \vdash e : \tau, \Gamma'$ so as to get $(\theta, v) \in \mathscr{E}[\![e]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]} \subseteq \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$.

- **Case [RECEIVE-2]**

  As in the previous case, we have to distinguish cases on the set to which (θ, v) belongs. But now the **Case 2** above cannot hold, since it assumes the existence of a pair $(\theta, v_t) \in \mathscr{T}[\![\tau_t]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_t]\!]}$ where $v_t \in \mathit{Integer}$ which contraducts the assumption, demanded by the rules [RECEIVE-2] of τ_t and `integer()` being disjoint. As a consequence, the only case to consider in this rule is **Case 1** above. The proof in this case does not depend on the existence of a judgement $\Gamma \vdash e : \tau, \Gamma'$, which is the only absent in [RECEIVE-2], so the theorem follows in this case.

- **Case [RECEIVE-3]**

  A necessary condition for (θ, v) belonging to $\mathscr{E}[\![\textbf{receive } cls_1 \ldots cls_n \textbf{ after } e_t \to e]\!]^\star$ is that the value v_t that results from the evaluation of e_t belongs to $\mathit{Integer} \cup \{\mathit{infinity}\}$. However, the assumptions in [RECEIVE-3] contradict this assumption, so we get that $\mathscr{E}[\![\textbf{receive } cls_1 \ldots cls_n \textbf{ after } e_t \to e]\!]^\star = \emptyset$ in this case, from which the theorem follows trivially.

- **Case [CLS-TRUE]**

  In this case we know that $\Gamma \Vdash_X cls : \tau, \Gamma'$ and have to prove that:

$$\{(\theta, v) \in \mathscr{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]} \subseteq \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$$

  for any $V \subseteq \mathit{Var}$. So, let us assume a set V of variables and a pair (θ, v) such that $(\theta, v) \in \mathscr{C}[\![cls]\!]_V$ and $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. If cls_i has the form p **when** $e_g \to e$, we know that $(\theta, v') \in \mathscr{E}[\![p]\!]^\star$ for every $v' \in V$, $(\theta, \mathit{true}) \in \mathscr{E}[\![e_g]\!]^\star$ and $(\theta, v) \in \mathscr{E}[\![e]\!]^\star$. By applying the induction hypothesis to the judgement $\Gamma \vdash p : \tau_p, \Gamma_p$, it holds that $v' \in \mathscr{T}[\![\tau_p]\!]^\star$ for every $v' \in V$ and $\theta \in \mathscr{T}_{Env}[\![\Gamma_p]\!]$. In other words, $V \subseteq \mathscr{T}[\![\tau_p]\!]^\star$. But, since for every variable $x \in X$ it holds that $\theta(x) \in V \subseteq \mathscr{T}[\![\tau_p]\!]^\star$, then it follows that $\theta \in \mathscr{T}_{Env}[\![[X : \tau_p]]\!]$, so $\theta \in \mathscr{T}_{Env}[\![\Gamma_p \sqcap [X : \tau_p]]\!]$ and we can apply, again, the induction hypothesis with the judgement $\Gamma_p \sqcap [X : \tau_p] \vdash e_g : \tau_g, \Gamma_g$ so as to get $(\theta, \mathit{true}) \in \mathscr{T}[\![\tau_g]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_g]\!]}$ which implies $\theta \in \mathscr{T}_{Env}[\![\Gamma_g]\!]$. Lastly, the induction hypothesis on the last judgement $\Gamma \vdash e : \tau, \Gamma'$ allows us to prove that $(\theta, v) \in \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$.

- **Case [CLS-FALSE]**

Similarly as in the previous case, we assume a set $V \subseteq Var$ and a pair $(\theta, v) \in \mathscr{C}[\![cls]\!]$. The latter fact leads to contradiction, since it would imply the existence of a tuple $(\theta, true) \in \mathscr{E}[\![e_g]\!]^\star \subseteq \mathscr{T}[\![\tau_g]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma_g]\!]}$, but the assumption in rule [CLS-FALSE] specifies that *true* and τ_g must be disjoint. Thus the assumption $(\theta, v) \in \mathscr{C}[\![cls]\!]$ cannot hold, and hence $\mathscr{C}[\![cls]\!]$ is empty and the theorem follows trivially.

- **Case [CLS-TRANS]**

  Assume a pair (θ, v) in $\{(\theta, v) \in \mathscr{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. There exist a type τ' and an environment Γ'' such that $\Gamma \Vdash_X cls : \tau', \Gamma''$ and $\Gamma'' \Vdash_X cls : \tau, \Gamma'$. From the induction hypothesis applied to the first judgement it follows that $(\theta, v) \in \mathscr{T}[\![\tau']\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma'']\!]}$ and hence $\theta \in \mathscr{T}_{Env}[\![\Gamma'']\!]$. Now we apply again the induction hypothesis in order to get:

  $$(\theta, v) \in \{(\theta, v) \in \mathscr{C}[\![cls]\!]_V \mid \forall x \in X, \theta(x) \in V\} \restriction_{\mathscr{T}_{Env}[\![\Gamma'']\!]} \subseteq \mathscr{T}[\![\tau]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$$

- **Case [LETREC]**

  In this case we assume that $e = \textbf{letrec } \overline{f_i = \textbf{fun}(\overline{x_{ij}}) \to e_i} \textbf{ in } e'$. However, for the sake of simplicity let us assume that we have a single recursive definition. That is, $e = \textbf{letrec } f = \textbf{fun}(\overline{y_i}) \to e_f \textbf{ in } e'$. The proof can be extended to several mutually recursive definitions in a straightforward way. Given a substitution θ, we define the function $F_\theta : DVal \to DVal$ as follows:

  $$\begin{aligned} &F_\theta(v) = v' \\ &\text{where } \{v'\} = \left\{ v'' \mid (\theta[f/v], v'') \in \mathscr{E}[\![\textbf{fun}(\overline{y_i}) \to e_f]\!]^\star \right\} \end{aligned}$$

  Since we have applied the [LETREC] rule, we must have the following judgement:

  $$\Gamma \sqcap [f : \tau_f] \vdash \textbf{fun}(\overline{y_i}) \to e_f : \tau'_f, \Gamma'_f$$

  for some τ_f, τ'_f and Γ'_f such that $\tau_f = \tau'_f \backslash f$, from which it follows the induction hypothesis:

  $$\mathscr{E}[\![\textbf{fun}(\overline{y_i}) \to e_f]\!]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma \sqcap [f:\tau_f]]\!]} \subseteq \mathscr{T}\left[\!\left[\tau'_f\right]\!\right]^\star \restriction_{\mathscr{T}_{Env}\left[\!\left[\Gamma'_f\right]\!\right]} \tag{3}$$

  Moreover, we know that τ'_f must be a functional type or a supertype of a functional type, so we assume that there exists a functional type $\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r$ such that $\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r = \tau'_f \sqcap (\overline{\texttt{any()}} \to \texttt{any()})$. If $\emptyset$ denotes the function that does not return any value (that is, the function with an empty graph), we shall prove the following for each natural number $m \geq 0$:

  $$F_\theta^m(\emptyset) \in \mathscr{T}_\theta\left[\!\left[\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r\right]\!\right] \tag{4}$$

  provided $\theta \in \mathscr{T}_{Env}[\![\Gamma]\!]$. In other words, if we iterate the application of F_θ to the empty function $\emptyset$ for a given number m of times, we get a value contained within the semantics of $\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r$. Let us prove (4) by induction on m:

  - **Base case ($m = 0$):** The function with the empty graph is contained within the semantics of every functional type, so (4) holds trivially.

– **Induction step ($m \geq 1$):** We get $F_\theta^m(\emptyset) = F_\theta(F_\theta^{m-1}(\emptyset))$, where $F_\theta^{m-1}(\emptyset)$ belongs to the semantics of $\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r$ by induction hypothesis, so it also belongs to $\mathscr{T}_\theta \left[\!\left[\tau'_f \right]\!\right] \subseteq \mathscr{T}_\theta \left[\!\left[\tau'_f \backslash f \right]\!\right] = \mathscr{T}_\theta \left[\!\left[\tau_f \right]\!\right]$. If we denote by w the function value $F_\theta^{m-1}(\emptyset)$, it holds that $\theta[f/w] \in \mathscr{T}_{Env} \left[\!\left[\Gamma \sqcap [f : \tau_f] \right]\!\right]$ and by (3) we get $(\theta[f/w], F_\theta(w)) \in \mathscr{T} \left[\!\left[\tau'_f \right]\!\right]^\star$ or, equivalently, $F_\theta(w) \in \mathscr{T}_{\theta[f/w]} \left[\!\left[\tau'_f \right]\!\right] \subseteq \mathscr{T}_\theta \left[\!\left[\tau_f \right]\!\right]$, the last step justified by Proposition 2. But, since $F_\theta(w)$ is also a function value, we get that $F_\theta(w)$ belongs to $\mathscr{T}_\theta \left[\!\left[(\overline{\texttt{any()}}) \rightarrow \texttt{any()} \right]\!\right]$ as well, so we get $F_\theta(w) \in \mathscr{T}_\theta \left[\!\left[\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r \right]\!\right]$.

Now let us prove that $lfp\, F_\theta \in \mathscr{T}_\theta \left[\!\left[\tau_f \right]\!\right]$. Since the function F_θ is monotone, we have the following ascending chain of functional values:

$$\emptyset \subseteq F_\theta(\emptyset) \subseteq F_\theta^2(\emptyset) \subseteq F_\theta^3(\emptyset) \subseteq \ldots$$

Moreover, we know that $\bigcup_{i=1}^{\infty} F_\theta^i(\emptyset) = lfp\, F_\theta$ by Kleene's fixed point theorem. If every function in the ascending chain shown above is the function with the empty graph $\emptyset$, then $lfp\, F_\theta = \emptyset$, which trivially belongs to the semantics of τ_f. Otherwise, there must be a value w in the ascending chain with a nonempty graph such that $w \in \mathscr{T}_\theta \left[\!\left[\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r \right]\!\right]$. This means that $\theta \in \mathscr{T}_{Env} \left[\!\left[\Gamma_f \right]\!\right]$, as otherwise we would obtain that $\mathscr{T}_\theta \left[\!\left[\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r \right]\!\right] = \{\emptyset\}$, contradicting the fact that the graph of w is not empty. Therefore, by definition of the semantics of a functional type, we know that:

$$\mathscr{T}_\theta \left[\!\left[\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r \right]\!\right] = \left\{ G \mid G \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathscr{T}_\theta \left[\!\left[\tau_i \right]\!\right], v \in \mathscr{T}_\theta \left[\!\left[\tau_r \right]\!\right] \right\} \right\}$$

By applying (4) we get an upper bound to the elements in the ascending chain:

$$\emptyset \subseteq F_\theta(\emptyset) \subseteq F_\theta^2(\emptyset) \subseteq F_\theta^3(\emptyset) \subseteq \ldots \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathscr{T}_\theta \left[\!\left[\tau_i \right]\!\right], v \in \mathscr{T}_\theta \left[\!\left[\tau_r \right]\!\right] \right\}$$

Therefore:

$$lfp\, F_\theta = \bigcup_{i=1}^{\infty} F_\theta^i(\emptyset) \subseteq \left\{ ((v_1, \ldots, v_n), v) \mid \forall i. v_i \in \mathscr{T}_\theta \left[\!\left[\tau_i \right]\!\right], v \in \mathscr{T}_\theta \left[\!\left[\tau_r \right]\!\right] \right\}$$

and

$$lfp\, F_\theta \in \mathscr{T}_\theta \left[\!\left[\overline{\tau_i} \xrightarrow{\Gamma_f} \tau_r \right]\!\right] \subseteq \mathscr{T}_\theta \left[\!\left[\tau'_f \right]\!\right] \subseteq \mathscr{T}_\theta \left[\!\left[\tau'_f \backslash f \right]\!\right] = \mathscr{T}_\theta \left[\!\left[\tau_f \right]\!\right] \tag{5}$$

Let us denote the least fixed point of F_θ by w, and assume that the pair (θ, v) is contained within the set $\mathscr{E} \left[\!\left[\textbf{letrec } f = \textbf{fun}(\overline{y_i}) \rightarrow e_f \textbf{ in } e' \right]\!\right]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma]\!]}$. Then it holds that $(\theta[f/w], v) \in \mathscr{E} \left[\!\left[e' \right]\!\right]^\star$ and $\theta \in \mathscr{T}_{Env} \left[\!\left[\Gamma \right]\!\right]$. Together with the fact that $w \in \mathscr{T}_\theta \left[\!\left[\tau_f \right]\!\right]$ (by (5)), it follows that $\theta[f/w] \in \mathscr{T}_{Env} \left[\!\left[\Gamma \sqcap [f : \tau_f] \right]\!\right]$. Moreover, since w is a fixed point of F_θ, it holds that $(\theta[f/w], w) \in \mathscr{E} \left[\!\left[\textbf{fun}(\overline{y_i}) \rightarrow e_f \right]\!\right]^\star$, so we can apply the induction hypothesis shown in (3) so as to get that $\theta[f/w] \in \mathscr{T}_{Env} \left[\!\left[\Gamma'_f \right]\!\right]$ or, equivalently, $\theta[f/w] \in \mathscr{T}_{Env} \left[\!\left[\left(\Gamma'_f \backslash f \right) \sqcap [f : \tau_f] \right]\!\right]$. By applying the induction hypothesis on the judgement $\left(\Gamma'_f \backslash f \right) \sqcap [f : \tau_f] \vdash e' : \tau', \Gamma''$ we get that:

$$(\theta[f/w], v) \in \mathscr{E} \left[\!\left[e' \right]\!\right]^\star \restriction_{\mathscr{T}_{Env}[\![(\Gamma'_f \backslash f) \sqcap [f:\tau_f]]\!]} \subseteq \mathscr{T} \left[\!\left[\tau' \right]\!\right]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma'']\!]}$$

And, by Proposition 2, we get $(\theta, v) \in \mathscr{T} \left[\!\left[\tau' \backslash f \right]\!\right]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma'' \backslash f]\!]} = \mathscr{T} \left[\!\left[\tau \right]\!\right]^\star \restriction_{\mathscr{T}_{Env}[\![\Gamma']\!]}$, which proves the theorem.

□

The following corollary shows that, in the particular case of closed exprerssions, our rules derive indeed success types in the sense of [7], which was in turn a generalization of [6].

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

7 Conclusions

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 (monomorphic) 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. When the rules are applied to closed expressions, they derive success types, i.e, overapproximations of the semantics. This is ensured by a corollary of the main correctness result proved in the paper.

Our plan for the next future is to extend our system so that polymorphic success types can be derived, starting by providing a suitable notion of polymorphic success type scheme. As far as we know, the first definition for that was given in [7], but it was too complex and presents some inconveniences in practice. Just to give a hint of why polymorphism is subtler here than in Hindley-Milner-like settings, we recall that in Hindley-Milner any instance of a valid polymorphic type scheme for an expression is a valid type for the expression. For instance, if $\forall a.a \to a$ is a valid type for an expression, then $bool \to bool$ or $int \to int$ must be also valid types. This cannot be true when considering success types, since those two monomorphic success types are incompatible for the same expression (they correspond to disjoint function graphs).

Whether or not we finally succeed in developing a sensible framework for polymorphic success types, we feel that our work here will be of great help, since we have contributed a good bunch of novel conceptual tools that should be directly applicable or at least inspiring for the enhanced setting, and that we summarize here:

- A denotational semantics for open Erlang expressions, presented in a relational way.
- A notion of types that reflects conditions over free variables in functional abstractions.
- Mutually recursive semantics for types and type environments.
- A type derivation system, containing many non-standard details like, e.g., the rules (TRANS) to retype expressions or (ABS), that takes care of decorating functional types with a type environment.
- Correctness results, showing that the whole framework is technically well devised.

References

[1] 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.

[2] 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.

[3] Miguel Jimenez, Tobias Lindahl, and Konstantinos F. Sagonas. A language for specifying type contracts in erlang and its interaction with success typings. In Thompson and Fredlund [10], pages 11–17.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Simon J. Thompson and Lars-Åke Fredlund, editors. *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007*. ACM, 2007.

A The map example

Here is the complete derivation from the example at Section 5.2:

$$\dfrac{\begin{array}{c}\Gamma_0 = [Mul : (\texttt{number}(), \texttt{number}()) \xrightarrow{[]} \texttt{number}()] \\ \Gamma_1 = \Gamma_0 \sqcap [Map : (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()]; Foo : ([\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()]] \\ (1) \quad \Gamma_1 \vdash \mathbf{fun}(V_1, V_2) \rightarrow \cdots : (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1 \\ (2) \quad \Gamma_1 \vdash \mathbf{fun}(V_0) \rightarrow \cdots : ([\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1 \\ \Gamma_1 \vdash Foo : ([\texttt{any}()]) \xrightarrow{[]} [\texttt{any}()], \Gamma_1\end{array}}{\Gamma_0 \vdash \mathbf{letrec}\ Map = \ldots Foo = \ldots \mathbf{in} \cdots : ([\texttt{any}()]) \xrightarrow{\Gamma_0} [\texttt{any}()], \Gamma_0}\ [\text{LETREC}]$$

$$\dfrac{\begin{array}{c}(3) \quad \Gamma_1 \vdash \mathbf{case}\ V_2\ \mathbf{of} \ldots \mathbf{end} : [\texttt{any}()], (\Gamma_2 = \Gamma_1 \sqcap [V_1 : \texttt{any}(); V_2 : [\texttt{any}()]]) \\ \Gamma_2 \backslash \{V_1, V_2\} = \Gamma_1 \qquad [\texttt{any}()] \backslash \{V_1, V_2\} = [\texttt{any}()] \\ (1) \quad \Gamma_1 \vdash \mathbf{fun}(V_1, V_2) \rightarrow \cdots : (\texttt{any}(), [\texttt{any}()]) \xrightarrow{\Gamma_1} [\texttt{any}()], \Gamma_1\end{array}}{}\ [\text{ABS}]$$

$$\dfrac{\begin{array}{c}\Gamma_3 = \Gamma_1 \sqcap [V_1 : \texttt{any}(); V_2 : []] \\ \Gamma_4 = \Gamma_1 \sqcap [V_1 : (\texttt{any}()) \xrightarrow{[]} \texttt{any}(); V_2 : [\texttt{any}()]; X : \texttt{any}(); XS : [\texttt{any}()]] \\ (4) \quad \Gamma_{1\{V_2\}} \Vdash [] \ \mathbf{when}\ \texttt{'true'} \rightarrow [] : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4 \\ (5) \quad \Gamma_{1\{V_2\}} \Vdash [X \mid XS] \ \mathbf{when}\ \texttt{'true'} \rightarrow \cdots : [\texttt{any}()], \Gamma_3 \sqcup \Gamma_4 \\ (3) \quad \Gamma_1 \vdash \mathbf{case}\ V_2\ \mathbf{of} \ldots \mathbf{end} : [\texttt{any}()], \Gamma_2\end{array}}{}\ [\text{CASE}]$$

$$\frac{\Gamma_1 \vdash [] : [], \Gamma_1 \qquad \Gamma_1 \sqcap [V_2 : []] \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_3 \qquad \texttt{'true'} \sqcap true \neq \texttt{none()} \qquad \Gamma_3 \vdash [] : [\texttt{any()}], \Gamma_3 \sqcup \Gamma_4}{(4) \quad \Gamma_{1\{V_2\}} \Vdash [] \textbf{ when } \texttt{'true'} \rightarrow [] : [\texttt{any()}], \Gamma_3 \sqcup \Gamma_4} \text{ [CLS-T]}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash [\texttt{X} \mid \texttt{XS}] : [\texttt{any()} \mid \texttt{any()}], \Gamma_1 \\ (\Gamma_1 \sqcap [V_2 : [\texttt{any()} \mid \texttt{any()}]] = \Gamma_5) \vdash \texttt{'true'} : \texttt{'true'}, \Gamma_5 \\ \texttt{'true'} \sqcap true \neq \texttt{none()} \qquad \Gamma_5 \vdash \textbf{let } V_3 = \ldots \textbf{ in } \cdots : [\texttt{any()}], \Gamma_3 \sqcup \Gamma_4 \quad (6)\end{array}}{(5) \quad \Gamma_{1\{V_2\}} \Vdash [X \mid XS] \textbf{ when } \texttt{'true'} \rightarrow \cdots : [\texttt{any()}], \Gamma_3 \sqcup \Gamma_4} \text{ [CLS-T]}$$

$$\frac{\begin{array}{l}(7) \quad \Gamma_5 \vdash V_1(X) : \texttt{any()}, (\Gamma_6 = \Gamma_5 \sqcap [V_1 : (\texttt{any()}) \xrightarrow{[]} \texttt{any()}; X : \texttt{any()}]) \\ (8) \quad \Gamma_6 \sqcap [V_3 : \texttt{any()}] \vdash \textbf{let } V_4 = \ldots \textbf{ in } \cdots : [\texttt{any()}], \Gamma_4 \sqcap [V_3 : \texttt{any()}]\end{array}}{(6) \quad \Gamma_5 \vdash \textbf{let } V_3 = \ldots \textbf{ in } \cdots : [\texttt{any()}], \Gamma_4} \text{ [LET]}$$

$$\frac{\Gamma_5(V_1) \sqcap ((\texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{any()}) \xrightarrow{[]} \texttt{any()}}{(7) \quad \Gamma_5 \vdash V_1(X) : \texttt{any()}, \Gamma_6} \text{ [APP1]}$$

$$\frac{\begin{array}{l}(9) \quad \Gamma_6 \sqcap [V_3 : \texttt{any()}] \vdash Map(V_1, XS) : [\texttt{any()}], (\Gamma_7 = \Gamma_6 \sqcap \\ \qquad [V_3 : \texttt{any()}; V_1 : \texttt{any()}; XS : [\texttt{any()}]] = \Gamma_4 \sqcap [V_3 : \texttt{any()}]) \\ (10) \quad (\Gamma_7 \sqcap [V_4 : [\texttt{any()}]] = \Gamma_8) \vdash [V_3 \mid V_4] : [\texttt{any()}], \Gamma_8\end{array}}{(8) \quad \Gamma_6 \sqcap [V_3 : \texttt{any()}] \vdash \textbf{let } V_4 = \ldots \textbf{ in } \cdots : [\texttt{any()}], \Gamma_4 \sqcap [V_3 : \texttt{any()}]} \text{ [LET]}$$

$$\frac{\Gamma_5(Map) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{any()}, [\texttt{any()}]) \xrightarrow{\Gamma_0} [\texttt{any()}]}{(9) \quad \Gamma_6 \sqcap [V_3 : \texttt{any()}] \vdash Map(V_1, XS) : [\texttt{any()}], \Gamma_7} \text{ [APP1]}$$

$$\frac{\Gamma_8 \vdash V_3 : \texttt{any()}, \Gamma_8 \qquad \Gamma_8 \vdash V_4 : [\texttt{any()}], \Gamma_8}{(10) \quad \Gamma_8 \vdash [V_3 \mid V_4] : [\texttt{any()}], \Gamma_8} \text{ [LIST]}$$

$$\frac{\begin{array}{l}(11) \quad \Gamma_1 \vdash \textbf{let } V_5 = \ldots \textbf{ in } \cdots : [\texttt{any()}], (\Gamma_9 = \Gamma_1 \sqcap [V_0 : [\texttt{any()}]]) \\ \qquad \Gamma_9 \backslash \{V_0\} = \Gamma_1 \qquad [\texttt{any()}] \backslash \{V_0\} = [\texttt{any()}]\end{array}}{(2) \quad \Gamma_1 \vdash \textbf{fun}(V_0) \rightarrow \cdots : ([\texttt{any()}]) \xrightarrow{\Gamma_1} [\texttt{any()}], \Gamma_1} \text{ [ABS]}$$

$$\frac{\begin{array}{l}(12) \quad \Gamma_1 \vdash \textbf{fun}(V_6) \rightarrow \cdots : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}, \Gamma_1 \\ (13) \quad (\Gamma_1 \sqcap [V_5 : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}] = \Gamma_{10}) \vdash Map(V_5, V_0) : \\ \qquad [\texttt{any()}], (\Gamma_{11} = \Gamma_9 \sqcap [V_5 : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}])\end{array}}{(11) \quad \Gamma_1 \vdash \textbf{let } V_5 = \ldots \textbf{ in } \cdots : [\texttt{any()}], \Gamma_9} \text{ [LET]}$$

$$\frac{\begin{array}{l}(14) \quad \Gamma_1 \vdash \textbf{let } V_7 = \ldots \textbf{ in } \cdots : \texttt{number()}, (\Gamma_{12} = \Gamma_1 \sqcap [V_6 : \texttt{number()}]) \\ \qquad \Gamma_{12} \backslash \{V_6\} = \Gamma_1 \qquad \texttt{number()} \backslash \{V_6\} = \texttt{number()}\end{array}}{(12) \quad \Gamma_1 \vdash \textbf{fun}(V_6) \rightarrow \cdots : (\texttt{number()}) \xrightarrow{\Gamma_1} \texttt{number()}, \Gamma_1} \text{ [ABS]}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash 2 : 2, \Gamma_1 \qquad \Gamma_{13} = \Gamma_1 \sqcap [V_7 : \texttt{number()}] \\ (15) \quad \Gamma_{13} \vdash Mul(V_6, V_7) : \texttt{number()}, (\Gamma_{14} = \Gamma_{12} \sqcap [V_7 : \texttt{number()}])\end{array}}{(14) \quad \Gamma_1 \vdash \textbf{let } V_7 = \ldots \textbf{ in } \cdots : \texttt{number()}, \Gamma_{12}} \text{ [LET]}$$

$$\frac{\Gamma_{13}(Mul) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{number()}, \texttt{number()}) \xrightarrow{[]} \texttt{number()}}{(15) \quad \Gamma_{13} \vdash Mul(V_6, V_7) : \texttt{number()}, \Gamma_{14}} \text{ [APP1]}$$

$$\frac{\Gamma_{10}(Map) \sqcap ((\texttt{any()}, \texttt{any()}) \xrightarrow{[]} \texttt{any()}) = (\texttt{any()}, [\texttt{any()}]) \xrightarrow{\Gamma_0} [\texttt{any()}]}{(13) \quad \Gamma_{10} \vdash Map(V_5, V_0) : [\texttt{any()}], \Gamma_{11}} \text{ [APP1]}$$