Journal of Computer Languages 58 (2020) 100965

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.editorialmanager.com/cola/default.aspx

Deriving overloaded success type schemes in Erlang☆

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

DSIC, Universidad Complutense de Madrid, Calle Prof. José García Santesmases 9, 28040 Madrid, Spain

ARTICLE INFO

Keywords:
Polymorphic types
Type systems
Erlang
Success types
Program semantics

ABSTRACT

Erlang is a programming language which brings together the features of functional programming and actor-based concurrency. Although it is a dynamically-typed language, there exists a tool (*Dialyzer*) that analyses Erlang programs in order to detect type discrepancies at compile-time. This tool is based on the notion of *success types*, which are overapproximations to the actual semantics of expressions, so that the evaluation of an 'ill-typed' expression will eventually fail at runtime. Dialyzer allows programmers to provide their own type specifications. Although such specifications can be polymorphic and overloaded (i.e., reflecting different executing branches) for documentation purposes, the type analysis disregards the information provided by polymorphic type schemes and so does, in some cases, with overloaded types. In this paper we introduce: (1) a type system that allows us to obtain polymorphic overloaded success type schemes for programs, (2) a semantic definition of this kind of types, and (3) correctness results that prove that the adequacy of the obtained types w.r.t. the semantics of expressions.

1. Introduction

Erlang is a concurrent functional language which lies at the heart of the Erlang/OTP platform. This platform consists of the Erlang runtime system and a set of libraries. It is designed to build distributed, fault-tolerant, soft real-time, highly available systems, and it provides hot-swapping capabilities[1]

As a consequence of its pragmatic design philosophy, Erlang is arousing increasing interest in industry and academia because of its strength in producing robust, easy to build and maintain, scalable systems. This is why several technologies are being developed with Erlang —such as message-broker software (e.g. RabbitMQ, VerneMQ), XMPP application servers (e.g. ejabberd), or distributed databases (e.g. Riak, Apache CouchDB, Amazon SimpleDB)— including WhatsApp's message server [1,2].

Unlike the static typed discipline provided by Hindley–Milner type systems [3] used in some languages (e.g., Haskell), the dynamically-typed nature of Erlang makes it more flexible in the task of programming. However, this flexibility comes at a price: unintended type errors might go unnoticed until the program is executed. Multiple attempts to design compile-time type analyses for Erlang can be found in the literature [4–6]. Among them, *Dialyzer* [7–9] can be used to report type discrepancies in programs, and *Typer* [10] is able to infer the type of the functions defined in a program, both for documentation purposes and for detecting errors.

Both Typer and Dialyzer are based on the notion of *success types*, which are overapproximations to the sets of values to which expressions may evaluate. In this context, there is a type (called none()) that denotes the empty set of values, so if an expression is given type none() by the system, we can ensure that its evaluation will not result in a value, either because it does not terminate, or because the evaluation fails at some point. In this sense, expressions having a success type none() are the closest analogue to ill-typed expressions in standard type systems. But, unlike the latter systems, there are expressions having success types different from none() whose evaluation might not succeed. This approach to type analysis fits into the pragmatic philosophy of Erlang. One of the design principles of Dialyzer is that it should neither require type annotations from the programmer nor produce false positives. If Dialyzer reports a type error, that error is bound to occur at runtime[2]

Dialyzer is a great tool to find runtime errors, but exhibits some weaknesses. For example, it cannot infer polymorphic types for functions. Programmers can attach polymorphic type specifications to their

☆ Work partially supported by the Spanish MINECO project CAVI-ART-2 (TIN2017-86217-R) and by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union.

* Corresponding author.
E-mail addresses: fraguas@ucm.es (F.J. López-Fraguas), montenegro@fdi.ucm.es (M. Montenegro), gorka.suarez@ucm.es (G. Suárez-García).

[1] Hot swapping allows developers to change the code of an application while it is running.

[2] Assuming that the computation terminates.

https://doi.org/10.1016/j.cola.2020.100965
Received 8 November 2019; Received in revised form 28 February 2020; Accepted 5 April 2020
Available online 19 April 2020

programs [9], but the inference algorithm cannot deal with polymorphic type variables, so these are transformed into monomorphic types before the analysis. Also the tool is not designed to infer by itself overloaded functional type specifications, although these overloaded types may be given by the user. An overloaded functional type represents a collection of functional types that represents the different branches of execution of the function, where the input and output types differ among those branches.

For example, a simple implementation of the `map` function in Erlang is:

```
map(_, []) -> [];
map(F, [X|XS]) -> [F(X)|map(F, XS)].
```

But the type inferred by Typer is:

```
-spec map(any(), [any()]) -> [any()].
```

The type inferred uses the type any() —which represents all the possible values that can be represented by the language— almost for every element. The only information we obtain from our `map` function is that the second parameter and the result are lists. However, this function admits the following overloaded type: $(\text{any}(), []) \to []$ $\sqcup$ $\forall\ \alpha_1, \beta_1, \alpha, \beta.\ (\alpha_1 \to \beta_1, [\alpha]) \to [\beta]$ **when** $\alpha \subseteq \alpha_1, \beta \subseteq \beta_1$. We can see in the example that neither the polymorphic variables nor the overloaded function type have been inferred.

A first contribution to address that problem was made in [11], where given an Erlang program with user-given polymorphic type specifications, a new one is synthesized such that Dialyzer, when run on the transformed program, infers more precise types for expressions that use polymorphic functions. However, this approach is limited by its tight dependence of Dialyzer. Any change made to the tool could affect and even invalidate the transformation proposed. Moreover, proving any theoretical result relies on trusting on non rigorously proved properties of Dialyzer. This is a second relevant weakness of Dialyzer: the lack of a rigorous formalization and a well developed theoretical framework upon which one can justify the technical correctness of the proposals.

This paper is a step forward to our intended goal of developing a full type system with associated type checking and type inference mechanisms that follow the philosophy of success types, but coping appropriately with the issues of polymorphism and having at the same time rigorous theoretical foundations. In previous work [12] we introduced a polymorphic type system as a first step towards this aim. Concretely, we stated a set of typing rules for deriving polymorphic success types for programs written in a desugared version of Erlang, and we proved them correct with respect to a suitable semantics of programs. In this paper we extend this work in several ways. Firstly, we add support for overloaded functional types. This allows us to obtain more accurate results not only when describing the result of a function, but also when deriving types for their callers. A noteworthy consequence is that we treat calls to functions involving runtime type checking (such as `is_integer`, `is_atom`, etc.) as if they were ordinary function calls, without the need for specific typing rules reflecting their behaviour. Another difference with respect to our previous work is the way in which non-linear types (i.e. functional types in which a type variable occurs more than once) are dealt with. The semantics of types and constraints given in [12] was rather contrived due to the need for tracking and joining the information corresponding to the different occurrences of a type variable. In this work we take an explicit approach by introducing new kinds of constraints on types, which allow us to do without non-linear types (thus greatly simplifying the technical development) while maintaining the expressiveness of the type system.

The rest of the paper is organized as follows: Section 2 introduces success types and their particularities by means of examples. Section 3 describes the syntax and semantics of the language that we shall study. Sections 4 and 5 put forward the main definitions in the type system and the derivation rules, the latter of which are exemplified in Section 6. In Section 7 we discuss the correctness of the type system (its proof can be found in a separate Appendix). Finally, Sections 8 and 9 consider alternative approaches to success types, and Section 10 concludes.

2. Success types: an informal overview

In this section we shall introduce the Erlang language, success types [8], and their relation with Dialyzer. We also highlight some differences between success types and Hindley–Milner type systems.

2.1. Erlang's syntax in a nutshell

As in the vast majority of functional languages, an Erlang program consists of a series of function definitions. As an example, let us introduce the following function `halve` that divides its input by two:

```
halve(X) -> X / 2.
```

We can attach a *guard* to `halve` in order to restrict the set of input values it accepts. For example, if we want to ensure that `halve` is only applied to even integer numbers, we can rewrite[3] it as follows:

```
halve1(X) when X rem 2 == 0 -> X div 2.
```

where `div` and `rem` are infix operators that denote integer division and remainder, respectively. With this guard, the evaluation of an expression such as $\text{halve}_1(3)$ would fail at runtime.

An Erlang function definition may have several *clauses*, which are scanned in order each time the function is called. For example, we can extend the previous example with a new clause handling error cases:

```
halve2(X) when X rem 2 == 0 -> X div 2;
halve2(X) when X rem 2 == 1 -> not_even.
```

In case the given input is an odd number, the function returns `not_even`, which is an *atom*. In Erlang, atoms are symbolic constants. One may think that the guard `X rem 2 == 1` in the second clause is redundant and that we could remove it so as to get:

```
halve3(X) when X rem 2 == 0 -> X div 2;
halve3(X) -> not_even.
```

However, this definition is not equivalent to that of halve_2, since halve_3 can be applied to non-integer data such as the atom `foo`, so halve_3(foo) would be evaluated to `not_even` whereas the evaluation of halve_2(foo) would fail. If we want halve_3 to check for non-integer values as input without having to compute `X rem 2`, we could add the following guard to the second clause:

```
halve4(X) when X rem 2 == 0 -> X div 2;
halve4(X) when is_integer(X) -> not_even.
```

in which the guard `is_integer(X)` is satisfied whenever `X` is an integer value.

Besides numbers and atoms, Erlang also supports lists and tuples. The syntax of lists is similar to those of Prolog: `[]` denotes the empty list and `[X|Xs]` denotes the list whose head is `X` and whose tail is `Xs`. With respect to tuples, the expression $\{e_1, \ldots, e_n\}$ denotes an n-ary tuple. For example, let us extend halve_3 to report a tuple containing the value the function has been applied to, in case this value is not an even number:

```
halve5(X) when X rem 2 == 0 -> X div 2;
halve5(X) -> not_even, X.
```

[3] We use a subscript below the function name just for presentation purposes, so that we can refer to the corresponding definition later.

2.2. Success types

As it was mentioned in Section 1, success types overapproximate the sets of values to which an expression is evaluated, and also overapproximate the set of input-output pairs defining the behaviour of a function. For example, given the halve_1 function defined above, the type (integer()) → integer() would be a success type for this function. This type specifies that the function expects an integer argument; otherwise it shall fail or shall not terminate. In case it executes successfully, it returns an integer number. From a semantic point of view, this type represents all functions whose graphs (i.e. sets of input-output pairs) are of the form (*n, m*), where *n* and *m* are integer numbers. Therefore, the type is a overapproximation of the behaviour of halve_1, and this approximation is strict, since the graph of halve_1 does not contain, for example, the tuple (5, 0). Conversely, we know that any input-output pair having a non-integer value as its first component cannot belong to the graph of any function with type (integer()) → integer(), and hence it cannot belong to the graph of halve_1. Therefore we can say that a function application such as halve_1(foo) will certainly fail at runtime.

In this setting, every literal (such as an integer or an atom) constitutes by itself a *singleton type* which represents that literal. In particular, `not_even` is a type that represents only the `non_even` atom. Therefore, a success type for halve_2 would be (integer()) → integer() ∪ not_even. The right-hand side at the arrow contains a *union type* specifying that halve_2 may return an integer number or the atom `not_even`.

Success types can be sometimes rather counter-intuitive in comparison with Hindley–Milner types. At first sight, one would think that the success type obtained for halve_2 would be also applicable to halve_3, but it is not. In fact, the expression halve_3(foo) evaluates to `not_even`, so the pair (foo, not_even) is contained within the graph of halve_3, but is not represented by (integer()) → integer() ∪ not_even. We would need a more generic type for halve_3, such as (any()) → integer() ∪ not_even, where the type any() denotes all Erlang values. A function with this type allows any kind of input. If we still want to obtain the `integer()` type we would have to add an `is_integer` guard to the second clause, as we did in halve_4. But, assuming that a programmer chooses *not* to include such a guard (e.g., because they want halve_3 to accept any kind of input), we can still describe more accurately the behaviour of halve_3 by using *overloaded success types*. For example, the overloaded type (integer()) → integer()⊔(any()) → not_even denotes all functions that may return an integer when given an integer as input, and may return `not_even` in any case.

The type $\{\tau_1, ..., \tau_n\}$ denotes the set of *n*-ary tuples such that, for every $i \in \{1..n\}$, the *i*-th component has type τ_i. Thus the halve_5 function accepts the following success type: halve_5: (integer()) → integer()⊔(any()) → {not_even, any()}. If the only information we know about halve_5 is this success type, the most accurate success type we could infer for the expression halve_5(foo) is {not_even, any()}. *Polymorphic success types* allow us to improve upon this by specifying a relation between the input and the output of a function, which is realized by means of *polymorphic type variables*. For example, the type (integer()) → integer()⊔ ∀ α. (α) → {not_even, α} is a success type of halve_5, which allows us to derive the type {not_even, foo} for the expression halve_5(foo). Besides this, polymorphic variables can be constrained. For example, had we added a guard `when is_integer(X)` to the second clause of halve_5, a success type for this function would be (integer()) → integer()⊔ ∀ α. (α) →{not_even, α} **when** α ⊆ integer().

Success typing brings several particularities not occurring in standard Hindley–Milner systems. Let us highlight some of them:

- There are no ill-typed expressions, in a strict sense. Every expression has at least one success type: any().
- There is no algorithm that can, in general, compute all the success types for an expression. This is because the notion of success types is semantic, instead of being defined by set of rules. For instance, assume the following expression:

```
case b of
 true -> 1;
 false -> 2
end
```

  in which *b* is a convoluted expression which always evaluates to `true`. It turns out that `1` is a success type for this expression, but an algorithm inferring that type would need to know whether *b* is always evaluated to `true`, which is an undecidable problem.
- Some expressions do not have minimal success type. For example, assume the following function:

```
g() -> case rand:uniform(2) of
 1 -> 0;
 2 -> {ok, g()}
end.
```

  where `rand:uniform(2)` may be evaluated to `1` or `2`. There exists an infinitely decreasing chain of success types for the expression `g()`, namely: any() ⊇ 0 ∪ {ok, any()} ⊇0 ∪ {ok, 0 ∪ {ok, any()}} ⊇ ....
- There is a subtyping relation between types, also defined semantically. However, this relation is covariant in the arguments and the result of a functional type. According to [9], the type $(\tau) \to \tau'$ is a success type for a function *f* if and only if for every pair of values *v* and *v′*:

  $f(v)$ is evaluated to $v' \Longrightarrow v$ is of type $\tau \wedge v'$ is of type τ'

  Hence, in order to prove that $(\tau_1) \to \tau_1' \subseteq (\tau_2) \to \tau_2'$ it is enough to prove that $\tau_1 \subseteq \tau_1'$ and $\tau_2 \subseteq \tau_2'$. In a standard Hindley–Milner context, it turns out that if $(\tau) \to \tau'$ is a type for *f*, then:

  $f(v)$ is evaluated to $v' \wedge v$ is of type $\tau \Longrightarrow v'$ is of type τ'

  Given that the condition of *v* being of type τ is at the left-hand side of the implication, the sufficient condition of covariance $\tau_1 \subseteq \tau_1'$ shown before turns into a contravariant relation: $\tau_1' \subseteq \tau_1$.
- The notion of polymorphism is subtler in the context of success types than in Hindley–Milner type systems. In Hindley–Milner, any instance of a valid polymorphic type scheme for an expression is a valid type for that expression. For example, if $\forall\alpha.(\alpha) \to \alpha$ is a valid Hindley–Milner type for the identity function, then so are (bool()) → bool() and (int()) → int(). This is not true when considering success types, since these two monomorphic success types are incompatible for the same expression (they correspond to disjoint function graphs). In fact, the first monomorphic type would forbid the application of the identity function to an integer, which is an expression that always succeeds.

2.3. Dialyzer: a discrepancy analyzer for Erlang

As mentioned above, Dialyzer is a tool that infers success types for every expression in a program and detects whether a sub-expression admits the type `none()`, which would mean that its evaluation is bound to fail at runtime. In our examples, Dialyzer infers for halve_1 and halve_2 the success types stated in Section 2.2. As regards to halve_3, it would infer the success type (any()) → integer() ∪ not_even for this function, in contrast to the overloaded type (integer()) → integer()⊔(any()) → not_even. Although Dialyzer does not infer overloaded type specifications, the user still may specify them in their programs, and Dialyzer exploits this information in order to obtain more accurate results. For example, given the following definitions:

```
-spec f (0) - -> 1 [] (1) - -> 2
f(0) -> 1;
f(1) -> 2.
g() -> f(f(1)).
```

$$
\begin{array}{rcl}
var & ::= & \textsc{VariableName} \\
lit & ::= & \textsc{Atom} \mid \textsc{Integer} \mid \textsc{Float} \mid [] \\
pat & ::= & var \mid lit \mid [pat_1 \mid pat_2] \mid \{\overline{pat_i}^n\} \\
cls & ::= & pat \ \textbf{when} \ exp_1 \rightarrow exp_2 \\
fun & ::= & \textbf{fun}(\overline{var_i}^n) \rightarrow exp
\end{array}
\qquad
\begin{array}{rcl}
exp & ::= & var \mid lit \mid fun \mid [exp_1 \mid exp_2] \mid \{\overline{exp_i}^n\} \\
 & & \mid var(\overline{var_i}^n) \mid \textbf{let} \ var = exp_1 \ \textbf{in} \ exp_2 \\
 & & \mid \textbf{letrec} \ \overline{var_i = fun_i}^n \ \textbf{in} \ exp \\
 & & \mid \textbf{case} \ var \ \textbf{of} \ \overline{cls_i}^n \ \textbf{end} \\
 & & \mid \textbf{receive} \ \overline{cls_i}^n \ \textbf{after} \ var \rightarrow exp
\end{array}
$$

Fig. 1. Subset of the Core Erlang syntax.

With the given overloaded specification for `f`, Dialyzer reports that the evaluation of `g()` will fail at runtime, since the expression `f(1)` has `2` as a success type, which does not belong to the domain of `f`, so it cannot be used in the outer call to `f`. Without an user-given overloaded specification for `f`, Dialyzer would obtain the specification `-spec f(0 ∪ 1) → 1 ∪ 2`. As a result, the expression `f(1)` would obtain the type `1 ∪ 2`, and so would `f(f(1))`. No type clashes would be reported.

At the time of the writing, Dialyzer does not support overloaded specifications for functions in which the types of the domains are non-disjoint. Such specifications are ignored. In our halve_3 example, the specification (integer()) → integer()⊔(any()) → not_even would be disregarded, as integer() and any() do not denote disjoint sets of elements (in fact, there is a subtyping relation between them). However such an specification would be useful to obtain more accurate results in certain cases. For example, assume the following definition:

```
halve_plus_one(X) -> halve3(X) + 1.
```

A sound success type for `halve_plus_one` is (integer()) → integer(). The fact that the subexpression halve_3(X) occurs as an argument of the + operator forces the branch (any()) → not_even in the overloaded specification of halve_3 to be discarded, so `X` is bound to be an `integer()`. Without an overloaded type, `X` would have been inferred to have type `any()`. The typing rules that will be introduced in this paper allow us to derive the type (integer()) → integer() for `halve_plus_one`.

Regarding polymorphism, Dialyzer currently supports user-given specifications with type variables. For example, in halve_5 the user could specify the type (integer()) → integer()⊔ ∀ α. (α) → {not_even, α}. Unfortunately, although these specifications are supported if the programmer supplies them, the extra information provided by polymorphism is lost when dealing with function applications. The occurrences of α in the type of halve_5 are collapsed into any(), which results in halve_5: (integer()) → integer()⊔(any()) → {not_even, any()}, thus losing the connection between the input and the output.

Having polymorphism into account allows a type inference tool to anticipate at compile time some program mistakes that would have gone unnoticed if the tool had just collapsed type variables to `any()`. As an example, let us consider the identity function `id`, which has type ∀α.(α) → α. This is more precise than (any()) → any(), since the former would allow us to derive that the evaluation of `id(true) + 3` shall not succeed. Similarly, polymorphic variables allow us to detect whether the elements of a list given to the `map` function (see Section 1) are not disjoint from the domain of the function given to `map`. In [11] a program transformation is introduced so that Dialyzer, when run over the transformed program, uses a monomorphic instance suitable to each call. However, deriving polymorphic type schemes for functions is not trivial, since the meaning of polymorphic type variables is not so well-studied in the context of success types as it is in the context of Hindley–Milner type systems. In this paper we introduce a semantic definition of polymorphic types such that the typing rules allow us, on the one hand, to take polymorphic information into account when deriving types for expressions and, on the other hand, to derive polymorphic type schemes for function definitions.

3. Language

In this section we introduce the language we work with in our type system and our examples.

3.1. Syntax

Erlang has a rich and powerful syntax that can be translated into Core Erlang, a desugared version of the language whose full syntax can be found in [13]. This work is focused on a selected subset of Core Erlang shown in Fig. 1. The differences between this subset and Core Erlang are meant to simplify the typing rules without losing generality. This subset has literal values, variables, lists, tuples, lambda abstractions, **let** expressions to introduce new variables, **letrec** expressions to introduce new recursive functions, **case** expressions to branch the execution, **receive** expressions to branch the execution when a message is received, and function applications. The variable in the **after** clause of a **receive** expression can be an integer or the atom `'infinity'`. In this last case the **after** clause will never be reached.

The literal values in this language are: atoms, integer numbers, float numbers, and the empty list represented with []. Atoms are symbolic constants, enclosed in single quotes (e.g. `'true'`, `'ok'`, `'Earth'`).[4]

The first difference between our subset and Core Erlang is that we unify the three different ways to apply functions. The first one uses **apply** to invoke functions inside the current module or λ-abstractions bound to 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'`.[5] From the point of view of typing, these are equivalent, so we give them an uniform treatment to remove noise from our typing rules.

Core Erlang has tuples, written as $\{\overset{n}{\cdots}\}$, but to improve efficiency it also uses internally another data structure called sequence, which is written as $<\overset{n}{\cdots}>$, and is very similar to a tuple. 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 cannot be nested. In our subset of Core Erlang, since sequences are an optimization mechanism of the implementation of Erlang, we will use tuples instead of sequences because they are pretty much similar from the point of view of types.

Core Erlang also has some expressions that 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.

[4] A difference between Erlang and Core Erlang is that Erlang allows atoms to be unquoted when they begin with a lower-case letter, underscore (_), or @; and does not contains other characters than alphanumeric characters.

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

Exceptions are supported in Erlang, but the "let it crash" [14] philosophy of the language discourages its extensive use. Thus we choose to leave **try/catch** and exceptions as a future goal.

In the chosen subset, we only allow variables in **case** discriminants and application parameters in order to simplify the typing rules. This allows us to attach type information to the discriminant when typing the branches. We also assume that, in the context of function applications, the function being applied and the arguments are variables, so that their types can be stored in a typing environment when analysing a function application.[6]

3.2. Semantics

In previous work [11,12] the semantics of a closed expression is defined as a subset of **DVal**, where **DVal** represents all the possible values that can be reached with the language expressions. To represent functions inside **DVal** we use graphs, which are sets of tuples $((\overline{args}), value)$ that relate a sequence of values *args* (the arguments) to a result value. Due to the non-deterministic nature of concurrent Erlang, a tuple *args* may be related to more than one result inside a function graph. In this sense, the semantics of a function is a mathematical relation rather than a function in a strict sense. To represent data structures inside **DVal** we also use tuples $(ctor, \overline{args})$, where *ctor* is the constructor of the structure and *args* is a sequence of values taken by the constructor. The constructors we have in our language are:

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

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

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

The function *matches* receives a substitution θ, a value v, and a clause p **when** $e_g \to e$, and gives us the set of substitutions where the program variables $\overline{x_i}$ —inside a pattern expression p— are set to certain values $\overline{v_i}$, such that this new substitution $\theta[\overline{x_i/v_i}]$ with the value v exists in the semantics of the pattern p, and the substitution $\theta[\overline{x_i/v_i}]$ with the value '`true`' exists in the semantics of the guard expression e_g. This means that the function returns the modified substitutions of θ that match the semantics of the pattern and satisfy the guard of a clause.

Using *matches*, the **case** and the **receive** semantics remove the values not reached by the execution of these expressions. For instance:

```
case X of
  Y when 'true' → 'first'
  Z when 'true' → 'second'
end
```

Without the usage of the function *matches*, the semantics of the expression would be a set with the values '`first`' and '`second`'. But using *matches*, all the possible substitutions for the second clause are contained in the substitutions of the first one, and that is why we cannot reach the execution of the second clause.

[6] We do not lose generality by introducing these constraints, since we can replace complex expressions by let-bound variables.

4. Type system

The syntax of types is shown in Fig. 3. As explained before, the types none() and any() denote the absence and the totality of values, respectively. For every basic type B (such as integer() or atom()), we assume a semantic definition $\mathcal{B}[\![B]\!] \subseteq$ **DVal** containing the set of values denoted by this type. Given a literal c, the namesake type c stands for the singleton set $\{c\}$ with the corresponding value. The same applies to the empty list $[\,]$. A type can also be a type variable α. We assume that there is a set **TypeVar** of type variables and we use α, β, α_1, etc. to denote elements from this set. As introduced in Section 2, there are tuple types of the form $\{\overline{\tau_i}^n\}$. The type nelist(τ_1, τ_2) represents those non-empty lists containing elements in τ_1 and whose continuations belong to τ_2. For example, nelist(integer(), [ ]) represents all non-empty *proper* lists of integers, i.e., those having the empty list as its innermost tail. In the following we use $[\tau]$ to abbreviate nelist(τ, [ ]) $\cup$ [ ].

The set of values to which type variables are instantiated can be restricted via *constraints*. In particular, the type τ **when** C denotes the values of type τ, in which the type variables occurring in τ are instantiated to sets of values satisfying the constraints in C. The syntax of constraints is defined in the last line of Fig. 3. A constraint of the form $\alpha \subseteq \tau$ specifies that the values to which α is instantiated are contained within τ. By abuse of notation, we write $\alpha_1 = \alpha_2$ to denote the conjunction of $\alpha_1 \subseteq \alpha_2$ and $\alpha_1 \supseteq \alpha_2$. We also allow constraints in which the left-hand side of the $\subseteq$ relation is a literal c. A constraint of the form $\bowtie \{\alpha_1, \ldots, \alpha_n\}$ specifies that, whenever each variable α_i is instantiated to a set V_i, the intersection of all the V_i must be non-empty. For example, the type {nelist(α_1, [ ]), nelist(α_2, [ ])} **when** $\bowtie \{\alpha_1, \alpha_2\}$ denotes those pairs of non-empty lists having a common element. This kind of constraints has been introduced in our type system in order to maintain the expressiveness of the non-linear types of [12], in which a pair of lists with common elements are denoted by {nelist(α, [ ]), nelist(α, [ ])}. Lastly, we have constraints of the form $\tau \Leftarrow \alpha$, which is a stronger (i.e. more restrictive) variant of $\alpha \subseteq \tau$. The precise meaning and the rationale behind this kind of constraints will be explained later after we have introduced type instantiations and environments.

A *polymorphic type scheme* σ has the form $\forall\ \overline{\alpha_j}.\ (\overline{\tau_i}^n) \to \tau$ and denotes the set of all n-ary functions that receive values in $\overline{\tau_i}^n$ and return a value of type τ. Al occurrences of the variables $\overline{\alpha_j}$ inside $(\overline{\tau_i}^n) \to \tau$ are said to be *bound*. A type of the form $\bigsqcup_{i=1}^m \sigma_i$ denotes an *overloaded* type scheme, which represents those functions resulting from joining functional values taken from each σ_i. For example the type $0 \to 1 \sqcup 1 \to 0$ contains the functions defined by the following graphs: $\emptyset$, {(0, 1)}, {(1, 0)}, {(0, 1), (1, 0)}. Notice the difference with the union type $0 \to 1 \cup 1 \to 0$. The latter denotes the graphs $\emptyset$, {(0, 1)}, {(1, 0)}, but not {(0, 1), (1, 0)}.

We say that an occurrence of a variable α inside a type τ is *free* if it is not bound. We denote by $ftv(\tau)$ the set of type variables occurring free in τ. We shall also consider that **when** has higher precedence than $\{\to, \cup, \sqcup\}$, so the type $(\alpha_1) \to \alpha_2$ **when** $\alpha_2 \subseteq \alpha_1$ is equivalent to $(\alpha_1) \to (\alpha_2$ **when** $\alpha_2 \subseteq \alpha_1)$.

4.1. Type instantiations

Having introduced the syntax, we have to give a meaning to each type. To that end we need a semantic function $\mathcal{T}[\![\_]\!]$ that maps every type to the set of values it denotes. This function would, for example, map the type integer() to the set of integer values, and map the type nelist(integer(), [ ]) to the set of all non-empty proper lists that can be built with integer values. However, things become more involved when we have to figure out the set of values corresponding to a type such as nelist(α, [ ]). When considering this type in isolation, it turns out that α could stand for any value, and hence the list $[3, 'true']$ belongs to the semantics of nelist(α, [ ]). However α may be restricted in a broader context, such as in the type nelist(α, [ ]) **when** $\alpha \subseteq$ integer(), which

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

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

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

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

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

$$\mathcal{E}\left[\!\left[\mathbf{case}\ x\ \mathbf{of}\ \overline{cls_i}^n\ \mathbf{end}\right]\!\right] = \bigcup_{i=1}^{n} \{(\theta, v) \mid \theta \in \mathbf{Subst}, \theta' \in matches(\theta, \theta(x), cls_i), (\theta', v) \in \mathcal{E}\,[\![e_i']\!],\ (\forall k < i : matches(\theta, \theta(x), cls_k) = \emptyset)\}$$
$$\text{where } \forall i \in \{1..n\} : cls_i = (p_i\ \mathbf{when}\ e_i \to e_i')$$

$$\mathcal{E}\left[\!\left[\mathbf{receive}\ \overline{cls_i}^n\ \mathbf{after}\ x_t \to e\right]\!\right] = \bigcup_{i=1}^{n} \{(\theta, v) \mid \theta \in \mathbf{Subst}, \theta(x_t) \in \texttt{integer()} \cup \{\texttt{'infinity'}\},\ v' \in \mathbf{DVal}, \theta' \in matches(\theta, v', cls_i), (\theta', v) \in \mathcal{E}\,[\![e_i']\!],\ (\forall k < i : matches(\theta, v', cls_k) = \emptyset)\}$$
$$\cup \{(\theta, v) \mid (\theta, v) \in \mathcal{E}\,[\![e]\!], \theta(x_t) \in \texttt{integer()}\}$$
$$\text{where } \forall i \in \{1..n\} : cls_i = (p_i\ \mathbf{when}\ e_i \to e_i')$$

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

$$matches(\theta, v, p\ \mathbf{when}\ e_g \to e) = \{\theta[\overline{x_i/v_i}] \mid \overline{v_i} \in \mathbf{DVal}, (\theta[\overline{x_i/v_i}], v) \in \mathcal{E}\,[\![p]\!] \wedge (\theta[\overline{x_i/v_i}], \texttt{'true'}) \in \mathcal{E}\,[\![e_g]\!]\}$$
$$\text{where } vars(p) = \{\overline{x_i}\}$$

Fig. 2. Denotational semantics of expressions.

$$\begin{array}{rcl} \mathbf{Type} \ni \tau & ::= & \texttt{none()} \mid \texttt{any()} \mid B \mid c \mid \texttt{[]} \mid \alpha \mid \{\overline{\tau_i}\} \mid \texttt{nelist}(\tau_1, \tau_2) \mid \tau_1 \cup \tau_2 \mid \bigsqcup_{i=1}^{m} \sigma_i \mid \tau\ \mathbf{when}\ C \\ \sigma & ::= & \forall \overline{\alpha_j}.(\overline{\tau_i}) \to \tau \\ C & ::= & \{\overline{\varphi_i}\} \\ \varphi & ::= & \alpha \subseteq \tau \mid c \subseteq \tau \mid \bowtie\{\alpha_1, \ldots, \alpha_n\} \mid \tau \Leftarrow \alpha \end{array}$$

Fig. 3. Syntax of types, type schemes, and constraints.

should exclude the list [3, 'true']. Therefore, in order to determine whether a value belongs to the semantics of a given type, we have to track the set of values to which α is instantiated for that value. In this example, α is instantiated to the set {3, 'true'} and we would determine, in the outer context, that this instantiation does not satisfy the constraint $\alpha \subseteq$ integer().

In a standard Hindley–Milner type system, instances of type variables are, in turn, types. In this work we adopt a slightly more generic approach and consider instead that a type variable is instantiated as a set of values. Thus we define a *type instantiation* as a function π: **TypeVar** → $\mathcal{P}$(**DVal**), and we denote by **TypeInst** the set of all type instantiations. We say that a variable α is *instantiated* by π if π(α) is non-empty. In the example above, we say that [3, 'true'] belongs to the semantics of nelist(α, [ ]) under any instantiation π such that π(α) = {3, 'true'}. Analogously, given the union type τ = {'ok', α} ∪ 'error', we say that {'ok', 5} belongs to the semantics of τ under every π such that π(α) = {5}, and that 'error' belongs to the semantics of τ under every π such that π(α) = ∅ (i.e. α is not instantiated in π).

As a consequence of the above, it turns out that our semantic function $\mathcal{T}[\![\_]\!]$ will determine, for each type τ, a set of pairs (*v*, π) instead of a set of values. The instantiation π specifies how the variables in τ are instantiated for that specific *v*. For example, we would say that ([3, 'true'], π) ∈ nelist(α, [ ]) for any π such that π(α) = {3, 'true'}. The occurrence of α in this type determines the value of π(α) in every pair (*v*, π) belonging to $\mathcal{T}[\![$nelist(α, [ ])$]\!]$. However, there may be some occurrences of a type variable α which still, despite restricting the corresponding π(α), they do it in a weaker way. As an example, consider the occurrence of α inside the constraint $\alpha \subseteq$ integer(). There are many choices for π(α) such that π satisfies this constraint. Therefore, it will be useful to distinguish the occurrences of α which strongly constraint the corresponding π(α) from those that are just meant to check some bounds on α. Given a variable $\alpha \in ftv(\tau)$, we say that a type variable is in an *instantiable position* of a type if it appears outside a constraint, or in an instantiable position in the left-hand side of a constraint of the form $\tau \Leftarrow \alpha$. We denote by *itv*(τ) the set of free type variables in τ that occur in the instantiable positions of τ.

Let us introduce some notation on type instantiations. We denote by [ ] the instantiation that maps every type variable to the empty set, and by $[\overline{\alpha_i \mapsto V_i}^n]$ the instantiation mapping each α_i to its corresponding set V_i and any other variable different from $\alpha_1, \ldots, \alpha_n$ to the empty set. Given two instantiations π_1, π_2 and a set *X* of type variables we say that $\pi_1 \equiv \pi_2$ (modulo *X*) iff $\pi_1(\alpha) = \pi_2(\alpha)$ for every $\alpha \in X$. Given an instantiation π and a set *X* of type variables, we denote by $\pi \backslash X$ the instantiation π' such that π' ≡ [ ] (modulo *X*) and π' ≡ π (modulo **TypeVar***X*). That is, π' is the substitution that behaves like π but leaving the variables in *X* uninstantiated. We also lift the set operator ∪ to instantiations and use the notation $\pi_1 \cup \pi_2$ to denote the π such that $\pi(\alpha) = \pi_1(\alpha) \cup \pi_2(\alpha)$ for every $\alpha \in$ **TypeVar**. This will allow us to decompose an instantiation π into a number of instantiations $\pi_1, \ldots, \pi_n$, each one typing a component of a given value (for instance, an element of a given list). Sometimes the decomposition should only apply to a

given subset of variables. Thus we define a notion of decomposition which is more restrictive than the $\cup$ operator. For any set Π of instantiations and type τ, we define the set $Dcp(\Pi, \tau)$ as follows:

$$\pi \in Dcp(\Pi, \tau) \stackrel{\text{def}}{\iff} \pi = \bigcup \Pi \text{ and } \forall\, \pi' \in \Pi.$$
$$\pi \equiv \pi' \text{ (modulo } \mathbf{TypeVar} \setminus itv(\tau))$$

For example, when Π is $\{[\alpha\mapsto\{1\}, \beta\mapsto\{4, 5\}], [\alpha\mapsto\{2\}, \beta\mapsto\{4, 5\}]\}$ and τ is nelist(α, [ ]), we say that $[\alpha\mapsto\{1, 2\}, \beta\mapsto\{4, 5\}] \in Dcp(\Pi, \tau)$, but there is no π such that $\pi \in Dcp(\{[\alpha\mapsto\{1\}, \beta\mapsto\{4\}], [\alpha\mapsto\{2\}, \beta\mapsto\{5\}]\}, \tau)$, since β is not in an instantiable position of τ, thus we shall not allow its decomposition into the subsets $\{4\}$ and $\{5\}$.

It is worth noting that, according to the semantic definition that will be given below, if there are multiple occurrences of the same type variable in different instantiable positions, then this variable will be instantiated to the same set in all these occurrences. For example, the type $\{\alpha, \alpha\}$ denotes the sets of tuples whose components are equal, and $([\alpha]) \to [\alpha]$ denotes those functions that, given a list, return another list containing the same elements as the input, possibly by rearranging the order of the elements, or by adding duplicates, or by discarding duplicates from the input list. This is different from what one would expect in a Hindley–Milner type system, in which the parametricity property [15,16] allows the elements of the output list to be a proper subset of those in the input list. If this is what we want to express in our type system, we would have to use the type $([\alpha]) \to [\alpha']$ **when** $\alpha' \subseteq \alpha$.

4.2. Semantics of types and constraints

In this section we give a precise meaning to types by defining the semantic function $\mathcal{T}[\![\_]\!]$ mentioned above. The definition is shown in Fig. 4. The semantics of none() is empty, since it denotes the absence of values. With respect to the semantics of any(), singleton types (literals and the empty list) and basic types, each value is paired with every possible instantiation, since no constraints are imposed on type variables. In the case of a type variable α, its semantics allows any possible value v, but now α becomes instantiated to the singleton set $\{v\}$.

Regarding tuple types $\{\overline{\tau_i}^n\}$, these represent n-ary tuples $(\{\cdot^n\cdot\}, v_1, ..., v_n)$ such that each v_i belongs to the semantics of its corresponding τ_i. The instantiations associated with each component are required to be all equal. Analogously, a list type nelist(τ, τ') denotes the set of non-empty lists, each one associated with an instantiation π. Unlike tuple types, the instantiation π can be decomposed into several π_i, one for each element of the list. In order to motivate the use of $\pi \in Dcp(\{\overline{\pi_i}^n\}, \tau)$ instead of $\pi = \bigcup_{i=1}^n \pi_i$ consider the type $[(0 \textbf{ when } \alpha \subseteq \text{int}()) \cup (1 \textbf{ when } \text{'true'} \subseteq \alpha)]$. Clearly the two constraints are incompatible, so it is sensible to exclude the list $[0, 1]$ from this type. However, we get that $(0, [\alpha \mapsto \{3\}]) \in \mathcal{T}[\![0 \textbf{ when } \alpha \subseteq \text{integer}()]\!]$ and $(1, [\alpha \mapsto \{\text{'true'}\}]) \in \mathcal{T}[\![1 \textbf{ when } \text{'true'} \subseteq \alpha]\!]$. If we used standard union to join all the instantiations we would obtain that $([0, 1], [\alpha \mapsto \{3, \text{'true'}\}])$ belongs to the semantics of the list type, which is not what is intended, especially when $\{3, \text{'true'}\}$ does not satisfy any of the constraints in that type.

The values denoted by a type of $\tau_1 \cup \tau_2$ is the union of the values denoted by each constituent. However, if we take a pair (v, π) from $\mathcal{T}[\![\tau_i]\!]$, we must ensure that all the variables in $itv(\tau_2)$ but not in $itv(\tau_1)$ become uninstantiated and vice versa. For example, the type nelist(α, [ ]) $\cup$ [ ] contains the empty list, but this value has to be paired with an instantiation π such that $\pi(\alpha) = \emptyset$. In order to understand the rationale behind this restriction, assume a function f of type $\forall\, \alpha_1, \alpha_2.\ ([\alpha_1]) \to [\alpha_2]$ **when** $\alpha_2 \subseteq \alpha_1$. We expect that f, when given a list, yields another list whose elements are a subset of those in the input list. As a consequence, we expect f to map the empty list only to another empty list. If we do not force α_1 to be uninstantiated in the case in which the input given to f is the empty list, then α_2 would be able to contain any element to which α_1 is instantiated, thus allowing f([ ]) to be evaluated to some other list rather than [ ].

The semantics of an overloaded type of the form $\bigsqcup_{i=1}^n \sigma_i$ is more involved. First let us concentrate on the semantics of a simple functional type $(\overline{\tau_i}^n) \to \tau$, defined by the $\mathcal{F}[\![\_]\!]$ function. This semantics denotes graphs of n-ary functions. The function that always fails (i.e. the function with an empty graph) belongs to every functional type. In this case none of the variables in the instantiable positions of $(\overline{\tau_i}^n) \to \tau$ would be actually instantiated. Now we assume that there is a pair $(f, \pi) \in \mathcal{T}[\![(\overline{\tau_i}^n) \to \tau]\!]$ such that the graph of f is non-empty. We decompose π into as many instantiations as tuples contained in the graph of f. For every input-output tuple $w = ((\overline{v_i}^n), v)$ in this graph associated with its corresponding instantiation π_w, each argument v_i has to belong to the type τ_i of the corresponding parameter, and the result v has to be contained within the type of the result (i.e. τ), all of them with the same π_w. For example, assume the type $([\alpha]) \to \text{integer}()$. We have, on the one hand, that $([1, 5], [\alpha \mapsto \{1, 5\}]) \in \mathcal{T}[\![[\alpha]]\!]$ and that $(0, [\alpha \mapsto \{1, 5\}]) \in \mathcal{T}[\![\text{integer}()]\!]$. On the other hand, we get that $([7, \text{'b'}, \text{'false'}], [\alpha \mapsto \{7, \text{'b'}, \text{'false'}\}]) \in \mathcal{T}[\![[\alpha]]\!]$ and that $(1, [\alpha \mapsto \{7, \text{'b'}, \text{'false'}\}]) \in \mathcal{T}[\![\text{integer}()]\!]$. Therefore, we get that the function

$$\mathcal{T}[\![\texttt{none()}]\!] = \emptyset \qquad \mathcal{T}[\![c]\!] = \{(c, \pi) \mid \pi \in \mathbf{TypeInst}\}$$
$$\mathcal{T}[\![\texttt{any()}]\!] = \{(v, \pi) \mid v \in \mathbf{DVal}, \pi \in \mathbf{TypeInst}\} \qquad \mathcal{T}[\![\texttt{[]}]\!] = \{(\texttt{[]}, \pi) \mid \pi \in \mathbf{TypeInst}\}$$
$$\mathcal{T}[\![\alpha]\!] = \{(v, \pi) \mid \pi \in \mathbf{TypeInst}, \pi(\alpha) = \{v\}\} \qquad \mathcal{T}[\![B]\!] = \{(v, \pi) \mid v \in \mathcal{B}[\![B]\!], \pi \in \mathbf{TypeInst}\}$$
$$\mathcal{T}[\![\{\overline{\tau_i}^n\}]\!] = \left\{ \left(\left(\{\cdot^n\cdot\}, \overline{v_i}^n \right), \pi \right) \;\middle|\; \forall i \in \{1..n\}.\, (v_i, \pi) \in \mathcal{T}[\![\tau_i]\!] \right\}$$
$$\mathcal{T}[\![\texttt{nelist}(\tau, \tau')]\!] = \{(([\texttt{\_|\_}], \overline{v_i}^n, v'), \pi) \mid n \geq 1, \forall i \in \{1..n\}.\, (v_i, \pi_i) \in \mathcal{T}[\![\tau]\!], \pi \in Dcp(\{\overline{\pi_i}^n\}, \tau), (v', \pi) \in \mathcal{T}[\![\tau']\!]\}$$
$$\mathcal{T}[\![\tau_1 \cup \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] \setminus (itv(\tau_2) \setminus itv(\tau_1)) \cup \mathcal{T}[\![\tau_2]\!] \setminus (itv(\tau_1) \setminus itv(\tau_2))$$
$$\text{where } \mathcal{T}[\![\tau]\!] \setminus X = \{(v, \pi \setminus X) \mid (v, \pi) \in \mathcal{T}[\![\tau]\!]\}$$
$$\mathcal{T}\left[\!\!\left[\bigsqcup_{i=1}^n \sigma_i\right]\!\!\right] = \left\{ \left(\bigcup_{i=1}^n f_i, \pi \right) \;\middle|\; \forall i \in \{1..n\}.(f_i, \pi) \in \mathcal{S}[\![\sigma_i]\!] \right\}$$
$$\mathcal{T}[\![\tau \textbf{ when } C]\!] = \{(v, \pi) \mid (v, \pi) \in \mathcal{T}[\![\tau]\!], \pi \vDash C\}$$

$$\mathcal{S}[\![\forall \overline{\alpha_j}^m.\, (\overline{\tau_i}^n) \to \tau]\!] = \{(f, \pi') \mid (f, \pi) \in \mathcal{F}[\![(\overline{\tau_i}^n) \to \tau]\!], \pi \equiv \pi' (\text{modulo } \mathbf{TypeVar} \setminus \{\overline{\alpha_j}^m\})\}$$

$$\mathcal{F}[\![(\overline{\tau_i}^n) \to \tau]\!] = \{(\emptyset, \pi) \mid \pi \in \mathbf{TypeInst}, \forall \alpha \in itv((\overline{\tau_i}^n) \to \tau).\, \pi(\alpha) = \emptyset\} \cup$$
$$\{(f|_1, \pi') \mid \emptyset \subset f \subseteq \{(((\overline{v_i}^n), v), \pi) \mid \forall i \in \{1..n\}.\, (v_i, \pi) \in \mathcal{T}[\![\tau_i]\!], (v, \pi) \in \mathcal{T}[\![\tau]\!]\},$$
$$\pi' \in Dcp(f|_2, (\overline{\tau_i}^n) \to \tau)\}$$
$$\text{where } f|_1 = \{w \mid (w, \pi) \in f\},\ f|_2 = \{\pi \mid (w, \pi) \in f\}$$

Fig. 4. Semantics of types and type schemes.

with graph {(([1, 5]), 0), (([7, 'b', 'false']), 1)} belongs to the semantics of $([\alpha]) \to \text{integer}()$ associated with the instantiation $[\alpha \mapsto \{1, 5, 7, 'b', 'false'\}]$.

Now let us consider the semantics of a functional type scheme $\forall\, \overline{\alpha_i}.\, (\overline{\tau_i}) \to \tau$, given by the function $\mathcal{S}[\![\_]\!]$. For every pair $(f, \pi) \in \mathcal{F}[\![(\overline{\tau_i}) \to \tau]\!]$ we have to "erase" from π the information relative to the bound type variables $\overline{\alpha_i}$, since it is irrelevant outside the context of the quantified type scheme. That is why we can replace the instantiation π by any other π' provided they agree in the variables that are not quantified in the scheme.

The semantics of a type of the form τ **when** C contains those pairs (v, τ) belonging to the semantics of τ but provided π satisfies all the constraints in C. We denote the latter condition by $\pi \vDash C$. The satisfiability relation is defined in Fig. 5. A constraint $\alpha \subseteq \tau$ is satisfied by π if the values of $\pi(\alpha)$ are contained within the semantics of τ, each one with an instantiation consistent with values of π. In this definition the $\subseteq$ relation on sets is lifted to type instantiations in the usual way. For example, we say that $\pi = [\alpha \mapsto \{[1, 2]\}, \beta \mapsto \{1, 2, 3\}]$ satisfies $\alpha \subseteq [\beta]$, but $\pi' = [\alpha \mapsto \{[1, 2, 5]\}, \beta \mapsto \{1, 2, 3\}]$ does not, since there is no π'' such that $([1, 2, 5], \pi'') \in \mathcal{T}[\![[\beta]]\!]$ and $\pi''(\beta) \subseteq \{1, 2, 3\}$. Satisfiability on constraints of the form $c \subseteq \tau$ is defined similarly. With respect to joinability constraints $\bowtie\{\overline{\alpha_i}^n\}$, the satisfiability relation holds for π if either all the variables $\overline{\alpha_i}^n$ are uninstantiated in π, or all of them are instantiated in π, but the corresponding sets to which they are instantiated have at least one element in common. Constraints of the form $\tau \Leftarrow \alpha$ are a stronger (i.e. more restrictive) version of subset constraints and they will be useful in normalization of type environments, as it will be explained at the end of the next section. As an example, let us consider the difference between $\pi \vDash \alpha \subseteq [\beta]$ and $\pi \vDash [\beta] \Leftarrow \alpha$. Assume that $\pi(\alpha)$ contains a single value, which is the list $[0, 4]$. The constraint $\alpha \subseteq \beta$ allows $\pi(\beta)$ to be any superset of $\{0, 4\}$, whereas the constraint $[\beta] \Leftarrow \alpha$ forces $\pi(\beta)$ to be *exactly equal* to $\{0, 4\}$. In general, if we get $\pi \vDash \tau \Leftarrow \alpha$ and $\pi(\alpha) = \{v\}$, then it must hold that $(v, \pi) \in \mathcal{T}[\![\tau]\!]$. In the case in which $\pi(\alpha) = \{v_1, ..., v_n\}$, then the v_i have to be contained within $\mathcal{T}[\![\tau]\!]$, each one possibly with a different instantiation, but the union of all these instantiations has to be exactly π. Back to the previous example, if $\pi(\alpha) = \{[0, 4], [1, 'true']\}$ and $\pi \vDash [\beta] \Leftarrow \alpha$, then it must hold that $\pi(\beta) = \{0, 4, 1, 'true'\}$.

4.3. Environments and annotated types

As usual in type systems, in order to typecheck a given expression, we have to track the types of the variables occurring in it. A *type environment* Γ is a pair (γ, C), where γ is a function that maps each program variable to its corresponding type and C is a set of constraints that pose restrictions to the type variables in γ. By abuse of notation, we shall use $\Gamma(x)$ instead of $\gamma(x)$, and use $.\Gamma|_C$ to denote the set of constraints in the right-hand side of Γ. The environment $[\,]$ denotes the mapping of all variables to any() whereas $\bot$ denotes the environment that maps all variables to none(). We use the notation $\Gamma[\overline{x_i : \tau_i}^n | C]$ to denote the environment Γ' such that $\Gamma'(x_i) = \tau_i$, for each $i \in \{1..n\}$, $\Gamma'(z) = \Gamma(z)$ for any other $z \notin \{\overline{x_i}^n\}$, and $\Gamma'|_C = \Gamma|_C \cup C$. As a particular case, we use $[\overline{x_i : \tau_i}^n | C]$ for abbreviating $[\,][\overline{x_i : \tau_i}^n | C]$.

The semantics of a type environment is the set of all substitutions that map variables to values belonging to their respective types. More precisely, given an environment Γ and an instantiation π, we define $\mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ as follows:

$$\mathcal{T}^{\pi}_{Env}[\![\Gamma]\!] = \{\theta \mid \forall\, x \in \mathbf{Var}.\ (\theta(x), \pi) \in \mathcal{T}[\![\Gamma(x)]\!],\ \pi \vDash \Gamma|_C\}$$

Notice that the instantiation used for each variable has to be the same as the one used to check the constraints in Γ. In case the instantiation is not relevant, we use $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ to denote that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ for some π.

The rules of the type system that will be described in Section 5 allow us to derive not only a success type for an expression, but also an environment describing the necessary conditions in order to evaluate this expression. Moreover, the success type of the expression may restrict the type variables occurring in this environment and vice versa. Therefore, it is convenient to group a type and an environment in a single pair with its own semantic definition. Moreover, if an expression contain **case** distinctions or calls a function with an overloaded type scheme, the type system will allows us to derive a sequence of such pairs. An *annotated type* is a sequence of pairs $\langle\tau_1; \Gamma_1\rangle; ...; \langle\tau_n; \Gamma_n\rangle$. The syntax and semantics of annotated types is given in Fig. 6. The semantics of an annotated type is a set of pairs (θ, v). We say that (θ, v) belongs to the semantics of $\langle\tau; \Gamma\rangle$ if there is an instantiation π such that v belongs to the semantics of τ and θ belongs to the semantics of Γ.

We define a pre-order relation on typing environments as follows: $\Gamma_1 \subseteq \Gamma_2$ if and only if $\mathcal{T}_{Env}[\![\Gamma_1]\!] \subseteq \mathcal{T}_{Env}[\![\Gamma_2]\!]$. We use $\Gamma_1 \approx \Gamma_2$ to denote that Γ_1 and Γ_2 are semantically equivalent. We can define analogous relations for annotated types. In some of the typing rules we shall make use of greatest lower bounds on environments. That is, given Γ_1 and Γ_2, we want to find some Γ such that $\mathcal{T}_{Env}[\![\Gamma]\!] = \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$. This requirement motivates the need for constraints of the form $\tau \Leftarrow \alpha$. As an example, consider the following pair of environments:

$$\Gamma_1 = [X: \text{nelist}(\alpha_1, [\,]),\ Y: \alpha_2 | \alpha_2 \subseteq \alpha_1] \qquad \Gamma_2 = [X: \alpha_3,\ Z: \alpha_4 | \alpha_3 = \alpha_4]$$

The first one specifies that X is bound to a list, and that Y is one of its elements. The second one specifies that X and Z are bound to the same value. A candidate for the greatest lower bound of these two environments would be $\Gamma = [X: \alpha_3,\ Y: \alpha_2,\ Z: \alpha_4 | \alpha_3 = \alpha_4,\ \alpha_3 \subseteq \text{nelist}(\alpha_1, [\,]),\ \alpha_2 \subseteq \alpha_1]$ but, unfortunately, the semantics of Γ is strictly greater than $\mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$. In fact, the substitution $[X/[1],\ Y/2,\ Z/[1]]$ belongs to $\mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ (if we choose $\pi = [\alpha_1 \mapsto \{1, 2\}, \alpha_2 \mapsto \{2\} \alpha_3 \mapsto \{[1]\}, \alpha_4 \mapsto \{[1]\}]$) but does not belong to $\mathcal{T}_{Env}[\![\Gamma_1]\!]$. The reason for this loss of accuracy is that α_1 appears in an instantiable position of Γ_1, but not in Γ. Hence whenever we find some $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma_1]\!]$, it holds that $\pi(\alpha_1)$ contains exactly the elements of the list $\theta(X)$, but, in Γ, the type variable α_1 can instantiated to any superset containing the elements of $\theta(X)$, thus breaking the requirement that Y is one of the elements of X. In order to avoid this, the greatest lower bound should be $[X: \alpha_3,\ Y: \alpha_2,\ Z: \alpha_4 | \alpha_3 = \alpha_4,\ \text{nelist}(\alpha_1, [\,]) \Leftarrow \alpha_3,\ \alpha_2 \subseteq \alpha_1]$, which represents $\mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$ without loss of precision.

In order to obtain a explicit form of the greatest lower bound of two environments and annotated types we need to *normalize* them beforehand. The key idea to normalize an environment is to map each program variable to a fresh type variable, adding new constraints to relate

$$\begin{array}{rcl}
\pi \vDash \alpha \subseteq \tau & \iff & \pi(\alpha) \subseteq \{v \mid (v, \pi') \in \mathcal{T}[\![\tau]\!], \pi' \subseteq \pi\} \\
\pi \vDash c \subseteq \tau & \iff & c \in \{v \mid (v, \pi') \in \mathcal{T}[\![\tau]\!], \pi' \subseteq \pi\} \\
\pi \vDash \bowtie\{\alpha_1, \ldots, \alpha_n\} & \iff & \bigcup_{i=1}^{n} \pi(\alpha_i) = \emptyset \vee \bigcap_{i=1}^{n} \pi(\alpha_i) \neq \emptyset \\
\pi \vDash \tau \Leftarrow \alpha & \iff & \exists (\pi_v)_{v \in \pi(\alpha)} : \pi = \bigcup_{v \in \pi(\alpha)} \pi_v \wedge (\forall v \in \pi(\alpha) : (v, \pi_v) \in \mathcal{T}[\![\tau]\!]) \\
\\
\pi \vDash \{\varphi_1, \ldots, \varphi_n\} & \iff & \forall i \in \{1..n\} : \pi \vDash \varphi_i
\end{array}$$

Fig. 5. Constraint satisfiability.

$$\rho ::= \langle \tau; \Gamma \rangle \mid \rho; \rho \qquad \mathcal{T}[\![\langle \tau; \Gamma \rangle]\!] = \{(\theta, v) \mid \theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!], (v, \pi) \in \mathcal{T}[\![\tau]\!]\}$$
$$\mathcal{T}[\![\rho_1; \rho_2]\!] = \mathcal{T}[\![\rho_1]\!] \cup \mathcal{T}[\![\rho_2]\!]$$

Fig. 6. Syntax and semantics of annotated types.

these new type variables to the types in the unnormalized environment:

$$norm([\overline{x_i{:}\ \tau_i} \mid C]) = [\overline{x_i{:}\ \alpha_i} \mid C \cup \{\overline{\tau_i \Leftarrow \alpha_i}\}] \text{ where } ftv([\overline{x_i{:}\ \tau_i} \mid C]) \cap \{\overline{\alpha_i}\} = \varnothing$$

It is easy to prove that $norm(\Gamma) \approx \Gamma$. Now we define the greatest lower bound operator ($\sqcap$) between environments in order to combine the information provided by several environments into a single one:

$$\Gamma_1 \sqcap \Gamma_2 = [\overline{x_i{:}\ \alpha_i} | C_1 \cup C_2] \text{ where } norm(\Gamma_1) = [\overline{x_i{:}\ \alpha_i} | C_1] \text{ and } norm(\Gamma_2) = [\overline{x_i{:}\ \alpha_i} | C_2] \text{ and } ftv(\Gamma_1) \cap ftv(\Gamma_2) = \varnothing$$

To join the sets of constraints C_1 and C_2 we ensure that no type variable is shared with the exception of the $\overline{\alpha_i}$, by renaming type variables if necessary. It is straightforward to prove that $\mathcal{T}_{Env}[\![\Gamma_1 \sqcap \Gamma_2]\!] = \mathcal{T}_{Env}[\![\Gamma_1]\!] \cap \mathcal{T}_{Env}[\![\Gamma_2]\!]$ for every Γ_1, Γ_2.

Finally, we define a $\otimes$ operator that, when applied to a sequence of pairs $\langle \tau_1; \Gamma_1 \rangle, \ldots, \langle \tau_n; \Gamma_n \rangle$, it joins all the τ_i into a tuple type while computing the greatest lower bound on all the Γ_i:

$$\langle \tau_1; \Gamma_1 \rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n \rangle = \langle \{\tau_1, \ldots, \tau_n\}; [\overline{x_i{:}\ \alpha_i} | C_1 \cup \cdots \cup C_n] \rangle$$
$$\text{where } norm(\Gamma_j) = [\overline{x_i{:}\ \alpha_i} | C_j] \text{ for all } j \in \{1..n\}$$
$$\text{and } ftv(\langle \tau_j; \Gamma_j \rangle) \cap ftv(\langle \tau_k; \Gamma_k \rangle) = \varnothing \text{ for all } j, k \in \{1..n\}, j \neq k$$
$$\text{and } ftv(\langle \tau_j; \Gamma_j \rangle) \cap \{\overline{\alpha_i}\} = \varnothing \text{ for all } j \in \{1..n\}$$

We can extend this operator to annotated types (i.e. sequences of pairs) by distributing $\otimes$ over the ; operator that composes annotated types:

$$\rho_1 \otimes \cdots \otimes (\rho_i; \rho_i') \otimes \cdots \otimes \rho_n = (\rho_1 \otimes \cdots \otimes \rho_i \otimes \cdots \otimes \rho_n); (\rho_1 \otimes \cdots \otimes \rho_i' \otimes \cdots \otimes \rho_n)$$

The $\otimes$ operator satisfies the following property, which the typing rule for tuples and lists will be based on:

Proposition 1. *For every substitution θ, values $v_1, \ldots, v_n$, and annotated types $\rho_1, \ldots, \rho_n$ such that $(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]$ for each $i \in \{1..n\}$ we get $(\theta, (\{\overset{n}{\ldots}\}, v_1, \ldots, v_n)) \in \mathcal{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!]$.*

Proof. See Appendix. □

5. Typing judgements

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

With the type rules shown in this section we shall obtain an annotated type for each expression e. However, it will be convenient to add to our judgements an initial environment which will reflect some (already known) assumptions on the free variables of the expression e. Therefore, our judgements will be of the form $\Gamma \vdash e{:}\ \rho$, with the following meaning: assuming that the values of the free variables in e (given by a substitution θ) are contained within their corresponding types in Γ, if e is evaluated to a value v, then the pair (θ, v) belongs to the semantics of ρ. More precisely, if $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ and $(\theta, v) \in \mathcal{E}[\![e]\!]$ then $(\theta, v) \in \mathcal{T}[\![\rho]\!]$. This can be expressed more succinctly as $\mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\rho]\!]$. In the following we use the terms *assumption environment* to refer to Γ and *final environment* to refer to any Γ' inside a pair in the sequence represented by ρ.

The typing rules are shown in Fig. 7. The first two are auxiliary rules to strengthen or weaken elements in our judgements. While [SUB1] specifies that we can replace the assumption environment Γ' by a stronger (i.e. more restrictive) one, [SUB2] allow us to weaken the annotated type ρ.

The [CNS] and [VAR] rules specifies that the final environment inside the pair that types the expression poses no further constraints besides those in the assumption environment. The [TPL] rule merges the annotated types of each subexpression with the operator $\otimes$, whose definition was given in Section 4.3, in order to obtain a tuple type as a result. The [LST] rule does the same for list constructions, but converting the tuple type constructor into a nelist type constructor for every pair in the resulting annotated type.

With respect to the [ABS] rule, the final environment is the same as the assumption environment, since the evaluation of a λ-abstraction always succeeds. We use the type variables $\overline{\alpha_i}$ to denote the types of the free variables $\overline{y_i}$ in the λ-abstraction. After analysing the body of the λ-abstraction, we receive an annotated type. Each pair in the annotated type will contain a result type τ_j and a type $\tau_{j,i}$ for each parameter x_i inside a final environment. We demand each final environment in the annotated type to be equal to the assumption environment, except for the types of the parameters $\overline{x_i}^n$. With each pair we will build a functional type, which has to be generalized using the operator $\overline{\forall}$. This operator takes all the instantiable type variables inside the functional type and binds them. Finally, we obtain a collection of functional types schemes that we will use to build the overloaded functional type, using the operator $\sqcup$. Each scheme might contain free type variables that also appear in Γ, but these have to appear directly bound to program variables in Γ.

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 result consists of an annotated type containing as many pairs as overloads in the type of f. Each pair contains a type τ_j for the result and a modification of the assumption environment, where new constraints have been added to change the types we know for the variables $\overline{x_i}^n$. To be able to apply this rule, the initial environment Γ_0 we obtained from the assumption environment must satisfy certain conditions: the first condition is that $\Gamma_0(f)$ has to be a (possibly overloaded) functional type, and the second is that all the arguments $\overline{x_i}^n$ must be associated with type variables $\overline{\alpha_{x_i}}^n$.

In order to avoid clashes between variables when applying the [APP1] rule, we have to create a group of *renamings* $\overline{\mu_j}^m$ for each overload. A renaming μ is a (partial) injective mapping from type variables to type variables. We assume the existence of a function *freshRenaming* that, given a set of type variables, it returns a renaming mapping those variables to fresh ones. The notation $\tau\mu$ denotes a type with the same structure as τ in which the type variables have been renamed according to μ. The final environment Γ_j shown in [APP1] adds the following constraints to Γ_0: each α_{x_i} variable corresponding to the i-th argument should match the type of the corresponding parameter $\tau_{j,i}$; and for each instantiable type variable β inside the functional type scheme, the fresh variable generated by the renaming has to be a subset of the original one.

Let us illustrate the rule [APP1] with the following example: $F(Z)$

$$\frac{\Gamma' \vdash e : \rho \qquad \Gamma \subseteq \Gamma'}{\Gamma \vdash e : \rho}\ [\text{SUB1}] \qquad \frac{\Gamma \vdash e : \rho \qquad \rho \subseteq \rho'}{\Gamma \vdash e : \rho'}\ [\text{SUB2}] \qquad \frac{}{\Gamma \vdash c : \langle c; \Gamma \rangle}\ [\text{CNS}]$$

$$\frac{}{\Gamma \vdash x : \langle \Gamma(x); \Gamma \rangle}\ [\text{VAR}] \qquad \frac{\Gamma \vdash e_i : \rho_i}{\Gamma \vdash \{\overline{e_i}^n\} : \rho_1 \otimes \cdots \otimes \rho_n}\ [\text{TPL}]$$

$$\frac{\Gamma \vdash e_1 : \rho_1 \qquad \Gamma \vdash e_2 : \rho_2 \qquad \rho_1 \otimes \rho_2 = \overline{\langle \{\tau_i, \tau_i'\}; \Gamma_i \rangle}^n}{\Gamma \vdash [e_1 \,|\, e_2] : \langle \mathtt{nelist}(\tau_i, \tau_i'); \Gamma_i \rangle}\ [\text{LST}]$$

$$\frac{\Gamma = \Gamma_0 \left[\overline{x_i : \mathtt{any}()}^n, \overline{y_i : \alpha_i}^l\right] \qquad \Gamma \vdash e : \overline{\langle \tau_j; \Gamma\,[\overline{x_i : \tau_{j,i}}^n] \rangle}^m \qquad \forall j \in \{1..m\} : ftv\left(\overline{\forall}\,(\overline{\tau_{j,i}}^n) \to \tau_j\right) \subseteq \{\overline{\alpha_i}^l\}}{\Gamma \vdash \mathbf{fun}(\overline{x_i}^n) \to e : \left\langle \bigsqcup_{j=1}^m \overline{\forall}\,(\overline{\tau_{j,i}}^n) \to \tau_j; \Gamma \right\rangle}\ [\text{ABS}]$$

where $\overline{\forall}\tau \stackrel{\text{def}}{=} \forall\overline{\alpha_i}.\tau$ and $\{\overline{\alpha_i}\} = itv(\tau)$.

$$\frac{\begin{array}{c}\Gamma \sqcap \left[f : \left(\overline{\mathtt{any}()}^n\right) \to \mathtt{any}()\right] \subseteq \Gamma_0 \\ \Gamma_0(f) = \bigsqcup_{j=1}^m (\forall\overline{\alpha_{j,i}}.\,(\overline{\tau_{j,i}}^n) \to \tau_j) \qquad \forall i \in \{1..n\} : \Gamma_0(x_i) = \alpha_{x_i} \\ \forall j \in \{1..m\} : \mu_j = \mathit{freshRenaming}\,(itv\,((\overline{\tau_{j,i}}^n) \to \tau_j) \cup \{\overline{\alpha_{j,i}}\}) \\ \forall j \in \{1..m\} : \Gamma_j = \Gamma_0\,[\{\overline{\tau_{j,i}\mu_j \Leftarrow \alpha_{x_i}}^n\} \cup \{\beta\mu_j \subseteq \beta \mid \beta \in itv\,(\forall\overline{\alpha_{j,i}}.\,(\overline{\tau_{j,i}}^n) \to \tau_j)\}]\end{array}}{\Gamma \vdash f(\overline{x_i}^n) : \overline{\langle \tau_j\mu_j; \Gamma_j \rangle}^m}\ [\text{APP1}]$$

$$\frac{\Gamma \sqcap \left[f : \left(\overline{\mathtt{any}()}^n\right) \to \mathtt{any}()\right] \approx \bot}{\Gamma \vdash f(\overline{x_i}^n) : \langle \mathtt{none}(); \bot \rangle}\ [\text{APP2}] \qquad \frac{\forall i \in \{1..n\} : \Gamma\,[x : \alpha] \Vdash_\alpha cls_i : \rho_i}{\Gamma\,[x : \alpha] \vdash \mathbf{case}\ x\ \mathbf{of}\ \overline{cls_i}^n : \overline{\rho_i}^n}\ [\text{CAS}]$$

$$\frac{\begin{array}{c}cls_i = (p_i\ \mathbf{when}\ e_i \to e_i') \qquad \mathit{fresh}\,(\alpha) \\ \Gamma \sqcap [x_t : \mathtt{integer}() \cup \text{'}\mathtt{infinity}\text{'}] \Vdash_\alpha cls_i : \rho_i \\ \Gamma \sqcap [x_t : \mathtt{integer}()] \vdash e : \rho\end{array}}{\Gamma \vdash \mathbf{receive}\ \overline{cls_i}^n\ \mathbf{after}\ x_t \to e : \overline{\rho_i}^n; \rho}\ [\text{RCV}] \qquad \frac{\begin{array}{c}\Gamma\,[\overline{x_i : \alpha_i}] \vdash p : \tau \qquad \{\overline{x_i}\} = vars(p) \\ \Gamma\,[\overline{x_i : \alpha_i} \mid \tau \Leftarrow \alpha] \vdash e_g : \overline{\langle \tau_i'; \Gamma_i' \rangle}^n \\ \forall i \in \{1..n\} : \Gamma_i'\,[\mathtt{true} \subseteq \tau_i'] \vdash e : \rho_i\end{array}}{\Gamma\,[x_i : \mathtt{any}()] \Vdash_\alpha p\ \mathbf{when}\ e_g \to e : \overline{\rho_i \backslash \{\overline{x_i}\}}^n}\ [\text{CLS}]$$

$$\frac{\Gamma\,[x : \mathtt{any}()] \vdash e_1 : \overline{\langle \tau_i; \Gamma_i \rangle}^n \qquad \forall i \in \{1..n\} : \Gamma_i\,[x : \tau_i] \vdash e_2 : \rho_i}{\Gamma\,[x : \mathtt{any}()] \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \overline{\rho_i \backslash \{x\}}^n}\ [\text{LET}] \qquad \frac{\forall i \in \{1..n\} : \Gamma\,[\overline{x_i : \tau_i}^n] \vdash f_i : \langle \tau_i; \Gamma\,[\overline{x_i : \tau_i}^n] \rangle \qquad \Gamma\,[\overline{x_i : \tau_i}^n] \vdash e : \rho}{\Gamma\left[\overline{x_i : \mathtt{any}()}^n\right] \vdash \mathbf{letrec}\ \overline{x_i = f_i}^n\ \mathbf{in}\ e : \rho \backslash \{\overline{x_i}^n\}}\ [\text{LRC}]$$

$$\frac{}{\Gamma \vdash c : c}\ [\text{LIT}_P] \qquad \frac{}{\Gamma\,[x : \alpha_x] \vdash x : \alpha_x}\ [\text{VAR}_P]$$

$$\frac{\Gamma \vdash p_1 : \tau_1 \qquad \Gamma \vdash p_2 : \tau_2}{\Gamma \vdash [p_1 \,|\, p_2] : \mathtt{nelist}(\tau_1, \tau_2)}\ [\text{LST}_P] \qquad \frac{\forall i \in \{1..n\} : \Gamma \vdash p_i : \tau_i}{\Gamma \vdash \{\overline{p_i}^n\} : \{\overline{\tau_i}^n\}}\ [\text{TPL}_P]$$

Fig. 7. Typing rules for expressions and clauses.

where F is bound to the function $\mathbf{fun}(X) \to \{X, X + Y\}$. The environment Γ_0 maps F to the type $\forall\,\alpha.\,(\alpha) \to \{\alpha, \text{number}()\}$ **when** $\alpha \subseteq \text{number}()$, $\beta \subseteq \text{number}()$, Y to β, and Z to α_Z. The renaming μ is created with *freshRenaming*, and we obtain $\mu = [\alpha/\alpha']$. Then, to build Γ_1 we add the constraint $\alpha' \Leftarrow \alpha_Z$ to Γ_0. Since we do not have free instantiable type variables in the type of F, we do not add further constraints. As result we obtain for $F(Z)$ the type $\langle \{\alpha', \text{number}()\}$ **when** $\alpha' \subseteq \text{number}(), \beta \subseteq \text{number}();\ \Gamma_0[\alpha' \Leftarrow \alpha_Z] \rangle$, which adds new information to the program variables Z and Y.

Our second example of the rule [APP1] is $F(X)$ when F has the type $(\alpha) \to \text{bool}()$. This happens, for example, when F is predicate function received as parameter (e.g., as in a *filter* function). Assume that the variable X has type β in Γ_0, and μ is $[\alpha/\alpha']$. To build Γ_1 we add the constraint $\alpha' \Leftarrow \beta$ to Γ_0, and —since we have a free instantiable type variable in the type for F— we also add $\alpha' \subseteq \alpha$ to connect the renamed variable with the original variable that occurs free in the functional type. As result we obtain for $F(X)$ the type $\langle \text{bool}();\ \Gamma_0[\alpha' \Leftarrow \beta,\ \alpha' \subseteq \alpha] \rangle$. This implies that X has to be bound to a value contained within the domain of F.

In order to derive a type for a **case** or a **receive** expression, we have to derive a type for each one of its clauses. With the [CLS] rule we obtain judgements of the form $\Gamma \Vdash_\alpha cls: \rho$; where α is a type variable, which is the type of the discriminant of a **case** expression or a fresh type variable in the case of a **receive** expression. The rule [CLS] demands the type of the discriminant to be compatible with the type of the pattern and the type of the guard to contain the atom 'true'. In order to satisfy the first requirement, we bind the variables appearing in the pattern p_i to fresh type variables, we derive the type of the pattern (by using rules [LIT_P], [VAR_P], [LST_P], and [TPL_P]), and finally we add a matching constraint between the type of the discriminant and the type of the pattern. The resulting environment is used to analyse the guard, and we obtain an annotated type $\overline{\langle \tau_i'; \Gamma_i' \rangle}^n$. For each pair in the annotated type of the guard, we will use as assumption environment the Γ_i' with an extra constraint that checks if τ_i' contains the atom 'true'. Then we derive the annotated type ρ_i for the body of the clause. Finally, we concatenate all the obtained ρ_i, disregarding the information relative to pattern variables. The notation $\rho \backslash \{\overline{x_i}\}$ returns an annotated type that results from replacing all the environments Γ in ρ by $\Gamma[\overline{x_i}: \text{any}()]$.

The rule [RCV] is similar to [CAS] but —in order to typecheck the clauses— demands the variable x_t to have a type inside integer() ∪ 'infinity', and —in order to typecheck the body of the **after** expression— demands the variable x_t not to contain 'infinity', since in that case the **after** clause would not be evaluated at runtime.

In the [LET] rule we obtain an annotated type $\overline{\langle \tau_i; \Gamma_i \rangle}^n$ for the bound expression e_1. This allow us to take Γ_i, override the type for x with τ_i, and use this modified environment as assumption environment for the main expression e_2. For each pair obtained analysing the bound expression, we will obtain an annotated type ρ_i for the main expression. Finally, we annotate the whole **let** expression with the concatenation of every ρ_i, removing the information regarding the bound variable x.

The [LRC] rule works like the [LET] rule with one main difference: to analyse the bound expressions f_i, we bind $\overline{x_i}^n$ to the types $\overline{\tau_i}^n$ in the assumption environment to analyse each bound expression. Since each f_i is a λ-abstraction, we know that the result will be a pair with a type and the same environment we use as assumption environment, and we also demand that the type inferred for each f_i is the type τ_i as initially

assumed.

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

Proposition 2. *Given any expression e and initial environment* Γ, *there exists an annotated type* ρ *such that* Γ⊢*e:* ρ. *In particular:* Γ⊢*e*: ⟨any(); [ ]⟩.

Proof. Straightforward, by inspection of the typing rules. Side-conditions involving the inclusion relation ⊆ between environments or annotated types can always be satisfied by choosing [ ] and ⟨any(); [ ]⟩ respectively on the right-hand side of these conditions. If the side-condition of [APP1] involving *f* to be a scheme type does not hold, then the rule [APP2] can be applied. The side-conditions of [APP1] involving $x_1, \ldots, x_n$ always hold, because in any environment if *x* is related to a type τ, a fresh α can be assigned to *x*, and the type can be moved to the constraints of the environment with the constraint τ⇐α. The remaining side-condition of [ABS], involving the free variables in the type scheme, depends on the annotated type obtained for *e*, which can be modified using [SUB2] to build a bigger type with less constraints if needed.

Once we prove have derived the judgement Γ⊢*e:* ρ for some ρ, and given that $\rho \subseteq \langle \text{any}(); [\,] \rangle$, we can use rule [SUB2] in order to obtain Γ⊢*e*: ⟨any(); [ ]⟩. □

6. Examples

In this section we will introduce some examples and the types obtained for each with our type system. To keep the examples shorter, the use of [CNS], [VAR], [CNS_P], [VAR_P], [TPL_P], and [LST_P] is not discussed because the use of these rules is trivial.

6.1. Using functions in guards

In this example we use a function is_atom with the following type:

$$\tau_{\text{is\_atom}} = (\text{atom}()) \to \text{'true'} \sqcup (\text{any}()) \to \text{'false'}$$

This type will be given in the initial environment.[7] Assume the following expression:

fun(*D*) → **case** *D* **of**
  M **when** is_atom(*M*) → 'plastic'
  T **when** 'true' → {'love', *T*}
end

The type (atom()) → 'plastic' ⊔ ∀ α. (α) → {'love', α} is obtained with our typing rules. Some of its judgements are shown at Fig. 8. Let us discuss the application of the [SUB2] rule marked with a (*) sign. This rule starts with the environment $\Gamma_0 = [D{:}\ \alpha, M{:}\ \beta | \beta \Leftarrow \alpha, \text{atom}() \Leftarrow \beta, \text{'true'} \subseteq \text{'true'}]$, where *D* is the parameter of the function and *M* is the variable introduced by the first clause of the **case** expression. Under Γ_0, the atom 'plastic' is typed by the rule [CNS] with the pair ⟨'plastic'; Γ_0⟩. This pair can be transformed using [SUB2], where the first step is to remove the 'true' ⊆ 'true' constraint since it holds that Γ['true' ⊆ 'true'] ≈ Γ for every Γ, so the constraint is irrelevant. After removing the constraint we know that α is used in the environment only to type the variable *D*, and we also have the constraint β⇐α, so we can replace α with β since it holds that Γ[*x*: α|α⇐β] ≈ Γ[*x*: β] for each Γ and *x* when α does not appear in Γ. After the substitution we have the pair ⟨'plastic'; [*D*: β, *M*: β|atom() ⇐ β]⟩. We can then apply the fact that Γ[*x*: β|τ⇐β] ⊆ Γ[*x*: τ|τ⇐β] for any *x* and Γ, thus obtaining ⟨'plastic'; [*D*: atom(), *M*: atom()|atom() ⇐ β]⟩. We have lost the connection between *D* and *M*, but *M* will be removed from the environment after exiting the scope of the clause anyway. Finally, we can remove the constraint by using the fact that Γ[τ⇐β] ≈ Γ when β does not occur in Γ.

In the judgement (**) we transform the environment before typing the tuple, thus removing the 'true' ⊆ 'true' constraint obtained from the result of the guard. We substitute α for β because of the constraint β⇐α, and after the substitution we rename β with the type variable α, which did not occur in the environment at that time. As result we obtain the pair ⟨{'love', α}; [*D*: α, *T*: α]⟩, where we can see the variable *D* connected to the result of the function by means of the type variable α.

Without support for overloaded functional types, the type for is_atom would be (any()) → bool() and we could not get any useful information from the guard in this example. To solve this issue, the type system would need a specific rule to deal with the application of is_atom. However, with overloaded functional types we can use the [APP1] rule to obtain the type information we need. For each functional type inside the type of is_atom, we obtain the annotated type made of the pairs ⟨'true'; [*D*: α, *M*: β|β ⇐ α, atom() ⇐ β]⟩ and ⟨'false'; [*D*: α, *M*: β|β ⇐ α, any() ⇐ β]⟩. Both pairs will be used to find a derivation for the body of the clause —in this example the atom 'plastic'—. In the case of the second pair, we obtain an environment with the constraint 'false' ⊆ 'true'. It holds that Γ['false' ⊆ 'true'] ≈ ⊥ for any Γ and that ρ; ⊥ ≈ ρ for any ρ, so the second branch of the annotated type is cancelled out when applying the [SUB2] rule.

6.2. The Map function

The next example will be a derivation of the function *Map*, which receives a function *F* and a list *L*, and returns a new list where each element is the result of applying *F* to the element at the same position in *L*. The code of *Map* is the following:

letrec *Map* = **fun**(*F*, *L*) → **case** *L* **of**
  [] **when** 'true' → []
  [*X*|*XS*] **when** 'true' → [*F*(*X*)|*Map*(*F*, *XS*)]
end in *Map*

The type (any(), []) → []⊔ ∀ α, α′, β, β′. ((α) → β, nelist(α′, [])) → nelist(β′, []) **when** α′ ⊆ α, β′ ⊆ β can be obtained for this *Map* function with our typing rules. This type represent those functions that:

- given any value and an empty list, return the empty list, and
- given a function with one parameter and a nonempty list whose elements are in the domain of that function, return a list whose elements must be in the range of function passed as argument.

Some of the judgements involved in the derivation are shown in Fig. 9, where τ_{Map} is the type obtained for *Map* (described above), and the initial environment is empty when the **letrec** rule is applied. We start from the application of the [ABS] rule since the application of the [LRC] rule is trivial for this example.

Using the tool *Typer* [10] from the Erlang distribution, the type we obtain is ((any()) → any(), [any()]) → [any()], which overapproximates the type obtained by our derivation, but does not take polymorphism into account.

6.2.1. Constraints transformation

The most remarkable step of the derivation is the transformation of the annotated type obtained from [LST] by applying the rule [SUB2] in the judgement (*) of Fig. 9. In this section we will discuss some of the transformations used to obtain the annotated type for that judgement.

One of the steps in the transformation is the conjunction of the constraints that define the type variable α_F. In the first pair of the annotated type we get (α) → β⇐α_F and any() ⇐ α_F. Every instantiation satisfies the latter constraint, so it can be discarded from the environment to obtain an equivalent one. In the second pair the conjunction is between (α) → β⇐α_F and (α″) → β″⇐α_F, in this case since both are functional types with type variables inside we can remove (α″) → β″

[7] For the sake of brevity, we do not show the binding of is_atom in the environments of the judgements shown in this example.

$$
\begin{array}{rl}
\text{[ABS]:} & [\,] \vdash \mathbf{fun}(D) \to \cdots : \langle (\texttt{atom()}) \to \texttt{'plastic'} \sqcup \forall\alpha.\,(\alpha) \to \{\texttt{'love'}, \alpha\}; [\,]\rangle \\
\text{[SUB1]:} & [\,] \vdash \mathbf{case}\ D\ \mathbf{of}\ \cdots : \langle \texttt{'plastic'}; [D : \texttt{atom()}]\rangle\,; \langle \{\texttt{'love'}, \alpha\}; [D : \alpha]\rangle \\
\text{[CAS]:} & [D : \alpha] \vdash \mathbf{case}\ D\ \mathbf{of}\ \cdots : \langle \texttt{'plastic'}; [D : \texttt{atom()}]\rangle\,; \langle \{\texttt{'love'}, \alpha\}; [D : \alpha]\rangle \\
\text{[CLS]:} & [D : \alpha] \Vdash_{\alpha}\ M\ \mathbf{when}\ \cdots : \langle \texttt{'plastic'}; [D : \texttt{atom()}]\rangle \\
\text{[APP1]:} & [D : \alpha, M : \beta \mid \beta \Leftarrow \alpha] \vdash \texttt{is\_atom}(M) : \langle \texttt{'true'}; [D : \alpha, M : \beta \mid \beta \Leftarrow \alpha, \texttt{atom()} \Leftarrow \beta]\rangle\,; \\
& \langle \texttt{'false'}; [D : \alpha, M : \beta \mid \beta \Leftarrow \alpha, \texttt{any()} \Leftarrow \beta]\rangle \\
(*)\ \text{[SUB2]:} & [D : \alpha, M : \beta \mid \beta \Leftarrow \alpha, \texttt{atom()} \Leftarrow \beta, \texttt{'true'} \subseteq \texttt{'true'}] \vdash \texttt{'plastic'} : \\
& \langle \texttt{'plastic'}; [D : \texttt{atom()}, M : \texttt{atom()}]\rangle \\
\text{[SUB2]:} & [D : \alpha, M : \beta \mid \beta \Leftarrow \alpha, \texttt{atom()} \Leftarrow \beta, \texttt{'true'} \subseteq \texttt{'false'}] \vdash \texttt{'plastic'} : \langle \texttt{none()}; \bot\rangle \\
\text{[CLS]:} & [D : \alpha] \Vdash_{\alpha}\ T\ \mathbf{when}\ \cdots : \langle \{\texttt{'love'}, \alpha\}; [D : \alpha]\rangle \\
(**)\ \text{[SUB1]:} & [D : \alpha, T : \beta \mid \beta \Leftarrow \alpha, \texttt{'true'} \subseteq \texttt{'true'}] \vdash \{\texttt{'love'}, T\} : \langle \{\texttt{'love'}, \alpha\}; [D : \alpha, T : \alpha]\rangle \\
\text{[TPL]:} & [D : \alpha, T : \alpha] \vdash \{\texttt{'love'}, T\} : \langle \{\texttt{'love'}, \alpha\}; [D : \alpha, T : \alpha]\rangle
\end{array}
$$

Fig. 8. Some of the judgements of the derivation in the example of Section 6.1.

$$
\begin{array}{rl}
\text{[ABS]:} & [\mathit{Map} : \tau_{\mathit{Map}}] \vdash \mathbf{fun}(F, L) \to \cdots : \langle \tau_{\mathit{Map}}; [\mathit{Map} : \tau_{\mathit{Map}}]\rangle \\
\text{[SUB1]:} & [\mathit{Map} : \tau_{\mathit{Map}}] \vdash \mathbf{case}\ L\ \mathbf{of}\ \cdots : \langle \texttt{[]}; [\mathit{Map} : \tau_{\mathit{Map}}, F : \texttt{any()}, L : \texttt{[]}]\rangle\,; \\
& \langle \texttt{nelist}(\beta', \texttt{[]})\ \mathbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; [\mathit{Map} : \tau_{\mathit{Map}}, F : (\alpha) \to \beta, L : \texttt{nelist}(\alpha', \texttt{[]})]\rangle \\
\text{[CAS]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L] \vdash \mathbf{case}\ L\ \mathbf{of}\ \cdots : \langle \texttt{[]}; [\mathit{Map} : \tau_{\mathit{Map}}, F : \texttt{any()}, L : \texttt{[]}]\rangle\,; \\
& \langle \texttt{nelist}(\beta', \texttt{[]})\ \mathbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; [\mathit{Map} : \tau_{\mathit{Map}}, F : (\alpha) \to \beta, L : \texttt{nelist}(\alpha', \texttt{[]})]\rangle \\
\text{[CLS]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L] \Vdash_{\alpha_L}\ \texttt{[]}\ \mathbf{when}\ \cdots : \langle \texttt{[]}; [\mathit{Map} : \tau_{\mathit{Map}}, F : \texttt{any()}, L : \texttt{[]}]\rangle \\
\text{[SUB2]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L \mid \texttt{[]} \Leftarrow \alpha_L, \texttt{'true'} \subseteq \texttt{'true'}] \vdash \texttt{[]} : \\
& \langle \texttt{[]}; [\mathit{Map} : \tau_{\mathit{Map}}, F : \texttt{any()}, L : \texttt{[]}]\rangle \\
\text{[CLS]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L] \Vdash_{\alpha_L}\ \texttt{[}X \mid XS\texttt{]}\ \mathbf{when}\ \cdots : \\
& \langle \texttt{nelist}(\beta', \texttt{[]})\ \mathbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; [\mathit{Map} : \tau_{\mathit{Map}}, F : (\alpha) \to \beta, L : \texttt{nelist}(\alpha', \texttt{[]})]\rangle \\
(*)\ \text{[SUB2]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L] \vdash \texttt{[}F(X) \mid \mathit{Map}(F, XS)\texttt{]} : \\
& \langle \texttt{nelist}(\beta', \texttt{[]})\ \mathbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma_4\rangle\,; \langle \texttt{nelist}(\beta', \texttt{[]})\ \mathbf{when}\ \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma_5\rangle \\
& \text{where } \Gamma_4 = [\mathit{Map} : \tau_{\mathit{Map}}, F : (\alpha) \to \beta, L : \texttt{nelist}(\alpha', \texttt{[]}), X : \alpha', XS : \texttt{[]}] \\
& \text{where } \Gamma_5 = [\mathit{Map} : \tau_{\mathit{Map}}, F : (\alpha) \to \beta, L : \texttt{nelist}(\alpha', \texttt{[]}), X : \alpha', XS : \texttt{nelist}(\alpha', \texttt{[]})] \\
\text{[LST]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L] \\
& \vdash \texttt{[}F(X) \mid \mathit{Map}(F, XS)\texttt{]} : \langle \texttt{nelist}(\beta', \texttt{[]}); \Gamma_4\rangle\,; \langle \texttt{nelist}(\beta', \texttt{nelist}(\beta''', \texttt{[]})); \Gamma_5\rangle \\
& \text{where } \Gamma_4 = [\mathit{Map} : \alpha_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid \tau_{\mathit{Map}} \Leftarrow \alpha_{\mathit{Map}}, (\alpha) \to \beta \Leftarrow \alpha_F, \texttt{any()} \Leftarrow \alpha_F, \\
& \quad \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L, \alpha' \Leftarrow \alpha_X, \texttt{[]} \Leftarrow \alpha_{XS}, \alpha' \subseteq \alpha, \beta' \subseteq \beta] \\
& \text{where } \Gamma_5 = [\mathit{Map} : \alpha_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid \tau_{\mathit{Map}} \Leftarrow \alpha_{\mathit{Map}}, (\alpha) \to \beta \Leftarrow \alpha_F, (\alpha'') \to \beta'' \Leftarrow \alpha_F, \\
& \quad \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L, \alpha' \Leftarrow \alpha_X, \texttt{nelist}(\alpha''', \texttt{[]}) \Leftarrow \alpha_{XS}, \alpha' \subseteq \alpha, \alpha''' \subseteq \alpha'', \beta' \subseteq \beta, \beta''' \subseteq \beta''] \\
\text{[APP1]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L] \vdash F(X) : \\
& \langle \beta'; [\mathit{Map} : \tau_{\mathit{Map}}, F : (\alpha) \to \beta, L : \alpha_L, X : \alpha', XS : \alpha_{XS} \mid \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L, \alpha' \Leftarrow \alpha_X, \alpha' \subseteq \alpha, \beta' \subseteq \beta]\rangle \\
\text{[APP1]:} & [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid \texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L] \vdash \mathit{Map}(F, XS) : \\
& \langle \texttt{[]}; [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid C_2]\rangle\,; \\
& \langle \texttt{nelist}(\beta''', \texttt{[]})\ \mathbf{when}\ \alpha''' \subseteq \alpha'', \beta''' \subseteq \beta''; [\mathit{Map} : \tau_{\mathit{Map}}, F : \alpha_F, L : \alpha_L, X : \alpha_X, XS : \alpha_{XS} \mid C_3]\rangle \\
& \text{where } C_2 = \{\texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L, \texttt{any()} \Leftarrow \alpha_F, \texttt{[]} \Leftarrow \alpha_{XS}\} \\
& \text{where } C_3 = \{\texttt{nelist}(\alpha_X, \alpha_{XS}) \Leftarrow \alpha_L, (\alpha'') \to \beta'' \Leftarrow \alpha_F, \texttt{nelist}(\alpha''', \texttt{[]}) \Leftarrow \alpha_{XS}\}
\end{array}
$$

Fig. 9. Some of the judgements involved in the derivation of *Map*.

adding the following constraints: $\alpha = \alpha''$ and $\beta = \beta''$. This is justified by the fact that whenever $(\theta, v) \in \mathcal{T}^{\pi}_{Env}[\![\Gamma[(\alpha) \to \beta \Leftarrow \alpha_F, (\alpha') \to \beta' \Leftarrow \alpha_F]]\!]$ holds, then and $\pi(\beta) = \pi(\beta')$. This transformation can also be applied to other structures, such as lists or tuples.

Other of the steps consists in joining two type variables into a supertype under certain conditions. For example, in the second branch of the annotated type obtained after applying [LST], we have the type nelist(β', nelist(β'', [])) and we know that $\beta' \subseteq \beta$ and $\beta''' \subseteq \beta''$. But the transformation explained previously introduced the constraint $\beta = \beta''$, so $\beta''' \subseteq \beta''$ can be transformed into $\beta''' \subseteq \beta$, and hence β' and β''' are subsets of β. Let us introduce a variable $\beta^{\star}$ defined as $\beta' \cup \beta'''$. We can replace $\beta' \subseteq \beta$ and $\beta''' \subseteq \beta$ with $\beta^{\star} \subseteq \beta$, and we can also replace nelist(β', nelist(β'', [])) with nelist($\beta^{\star}$, nelist($\beta^{\star}$, [])), which can be over-approximated with nelist($\beta^{\star}$, []). After finishing all those changes, and remove the unused constraints, we can rename $\beta^{\star}$ to β' since the latter was removed previously from the pair. This step is also applied to join α' and α'''.

We have mentioned in Section 6.1 the substitution of type variables involved in match constraints and the renaming of type variables inside a pair of an annotated type. Another trivial transformation consists in moving constraints from the environment to the right-hand side of a **when** type and vice versa. That is, $\langle \tau; \Gamma[C] \rangle \approx \langle \tau\ \mathbf{when}\ C; \Gamma \rangle$.

6.2.2. Using the Map function

Now we will show some use cases where *Map* is applied to valid and non valid arguments. The first use case is *Map*(*F, L*) under an environment Γ that maps *F* to (number()) → number(), *L* to [integer()], and *Map* to the type obtained before:

$\Gamma \vdash Map(F, L)$: $\langle []; \Gamma[F{:}\ \alpha_F, L{:}\ \alpha_L|C_1]\rangle$;
$\langle \text{nelist}(\beta', []) \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma[F{:}\ \alpha_F, L{:}\ \alpha_L|C_2]\rangle$
where $C_1 = \{(\text{number}()) \to \text{number}() \Leftarrow \alpha_F, [\text{integer}()] \Leftarrow \alpha_L,$
$\text{any}() \Leftarrow \alpha_F, [] \Leftarrow \alpha_L\}$
where $C_2 = \{(\text{number}()) \to \text{number}() \Leftarrow \alpha_F, [\text{integer}()] \Leftarrow \alpha_L,$
$(\alpha) \to \beta \Leftarrow \alpha_F, \text{nelist}(\alpha', []) \Leftarrow \alpha_L\}$

The application has given us two pairs. The first pair forces L to contain an empty list, F remains as it is, and the result is the empty list. The second pair can be transformed so as to obtain that $\text{number}() \Leftarrow \alpha$, $\text{number}() \Leftarrow \beta$, and $\text{integer}() \Leftarrow \alpha'$. The given result in the second pair is a non-empty list of type number(). The annotated type we obtain with these transformations and the rule [SUB2] is:

$\Gamma \vdash Map(F, L)$: $\langle []; \Gamma[F{:}\ (\text{number}()) \to \text{number}(), L{:}\ []]\rangle$;
$\langle \text{nelist}(\text{number}(), []); \Gamma[F{:}\ (\text{number}()) \to \text{number}(), L : \text{nelist}(\text{integer}(), [])]\rangle$

The second use case is *Map(F, L)* under a Γ defined as before, but now F is mapped to the type $(\text{bool}()) \to \text{bool}()$, and L to $[\text{integer}()]$:

$\Gamma \vdash Map(F, L)$: $\langle []; \Gamma[F{:}\ \alpha_F, L{:}\ \alpha_L|C_1]\rangle$;
$\langle \text{nelist}(\beta', []) \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta; \Gamma[F : \alpha_F, L{:}\ \alpha_L|C_2]\rangle$
where $C_1 = \{(\text{bool}()) \to \text{bool}() \Leftarrow \alpha_F, [\text{integer}()] \Leftarrow \alpha_L,$
$\text{any}() \Leftarrow \alpha_F, [] \Leftarrow \alpha_L\}$
where $C_2 = \{(\text{bool}()) \to \text{bool}() \Leftarrow \alpha_F, [\text{integer}()] \Leftarrow \alpha_L,$
$(\alpha) \to \beta \Leftarrow \alpha_F, \text{nelist}(\alpha', []) \Leftarrow \alpha_L\}$

This time the application of rule [APP] yields two pairs, the second of which is equivalent to $\langle \text{none}(); \bot\rangle$. This is because the environment $\Gamma[F{:}\ \alpha_F, L{:}\ \alpha_L|C_2]$ is equivalent to $[F{:}\ \alpha_F, L{:}\ \alpha_L|C_2 \cup \{\text{bool}() \Leftarrow \alpha, \text{bool}() \Leftarrow \beta, \text{integer}() \Leftarrow \alpha'\}]$, so every substitution θ belonging to the semantics of this environment has to be obtained with an instantiation π in which $\pi \vDash \text{bool}() \Leftarrow \alpha$ and $\pi \vDash \text{integer}() \Leftarrow \alpha'$. But there is no value v such that (v, π) belongs to the semantics of the type $\text{nelist}(\beta', [])$ **when** $\alpha' \subseteq \alpha$, $\beta' \subseteq \beta$, since π cannot satisfy the constraint $\alpha' \subseteq \alpha$. After applying the rule [SUB2] the annotated type we obtain is:

$\Gamma \vdash Map(F, L)$: $\langle []; \Gamma[F{:}\ (\text{bool}()) \to \text{bool}(), L{:}\ []]\rangle$;

This means that the only chance of success to execute this application is that program variable L contains an empty list.

6.3. Higher-order & list-related functions

In this section we will show the types obtained for some functions involving lists, such as *Foldl, Reverse, Filter*, and *Nth*; some of them are higher-order functions. With the *Typer* [10] tool, the types obtained are:

Foldl : $((\text{any}(), \text{any}()) \to \text{any}(), \text{any}(), [\text{any}()]) \to \text{any}()$
Filter : $((\text{any}()) \to \text{any}(), [\text{any}()]) \to [\text{any}()]$
Reverse : $([\text{any}()]) \to [\text{any}()]$
Nth : $(\text{pos\_integer}(), \text{nonempty\_maybe\_improper\_list}()) \to \text{any}()$

With our type system we can derive the following types instead:

Foldl : $\forall\, \beta.\ (\text{any}(), \beta, []) \to \beta \sqcup \forall\, \alpha, \alpha', \beta, \beta', \gamma, \gamma'.((\alpha, \beta) \to \gamma, \beta', \text{nelist}(\alpha', [])) \to \gamma'$ **when** $\alpha' \subseteq \alpha$, $\beta' \subseteq \beta$, $\gamma' \subseteq \gamma$
Filter : $(\text{any}(), []) \to [] \sqcup \forall\, \alpha, \beta, \beta'.\ ((\alpha) \to \text{bool}(), \text{nelist}(\beta, [])) \to \text{nelist}(\beta', [])$ **when** $\beta \subseteq \alpha$, $\beta' \subseteq \beta$
Reverse : $\forall\, \alpha.\ ([\alpha]) \to [\alpha]$
Nth : $\forall\, \alpha, \beta.\ (\text{number}(), \text{nelist}(\alpha, \text{any}())) \to \beta$ **when** $\beta \subseteq \alpha$

The types given by *Typer* for *Foldl, Filter*, and *Reverse*, are supertypes of the types we have derived with our type system. Only in the case of the type of *Nth* we find that *Typer* has found a more accurate type in the first argument of the function. To improve our derivations with arithmetic operations we would need overloaded types for arithmetic operators, so that they distinguish cases according to the sign of their arguments.

In the following sections we explain how these types have been obtained.

6.3.1. The Foldl function

The *Foldl* function takes a function, an accumulated value, and a list; it is used to reduce a list into a single accumulated value by applying the given function to every element of the list (from left to right) while maintaining an accumulator that is propagated throughtout all applications. The code of this function is as follows:

```
letrec Foldl = fun(F, A, L) -> case L of
  [] when 'true' -> A
  [X|XS] when 'true'->
    let B = F(X, A) in Foldl(F, B, XS)
end in Foldl
```

The type obtained for *Foldl* is $\forall\, \beta.\ (\text{any}(), \beta, []) \to \beta \sqcup \forall\, \alpha, \alpha', \beta, \beta', \gamma, \gamma'.\ ((\alpha, \beta) \to \gamma, \beta', \text{nelist}(\alpha', [])) \to \gamma'$ **when** $\alpha' \subseteq \alpha$, $\beta' \subseteq \beta$, $\gamma' \subseteq \gamma$ from a derivation with our type system. In the first overload of the given type, the first parameter F can be any value, the second parameter A is also the result type of the functional type, and the third parameter L is an empty list. In the second overload, the first parameter F is the function that mixes the received accumulated value in the second parameter A with the head of the third parameter L. For that reason the type for A is contained in the type of the second parameter of F and the type for L is contained in the type of the first parameter of F, and the result of the functional type is the result type of F.

The type $\forall\, \alpha, \alpha', \beta, \beta', \gamma, \gamma'.\ ((\alpha, \beta) \to \gamma, \beta', [\alpha']) \to \gamma'$ **when** $\alpha' \subseteq \alpha$, $\beta' \subseteq \beta$, $\gamma' \subseteq \gamma$ is not a success type of this function, because when L is an empty list, the parameter A need not be related to the type of F's result, since the accumulating function is not going to be called. For this reason, when L is an empty list, the result obtained from the [CLS] rule is the pair $\langle \beta; [F{:}\ \text{any}(), A{:}\ \beta, L{:}\ []]\rangle$ whereas in the clause handling non-empty lists, we obtain $\langle \gamma' \textbf{ when } \alpha' \subseteq \alpha, \beta' \subseteq \beta, \gamma' \subseteq \gamma; [F{:}\ (\alpha, \beta) \to \gamma, A{:}\ \beta', L{:}\ \text{nelist}(\alpha', [])]\rangle$.

6.3.2. The Filter function

The *Filter* function takes a predicate function and a list, and returns a new list without the elements for which the predicate returns `'false'`. The code of the function is as follows:

```
letrec Filter = fun(P, L) -> case L of
    [] when 'true' -> []
    [X|XS] when 'true'->
      let B = P(X) in let L2 = Filter(P, XS)
      in case B of
          'true' when 'true' -> [X|L2]
          'false' when 'true' -> L2
      end
end in Filter
```

The type $(\text{any}(), []) \to [] \sqcup \forall\, \alpha, \beta, \beta'.\ ((\alpha) \to \text{bool}(), \text{nelist}(\beta, [])) \to \text{nelist}(\beta', [])$ **when** $\beta \subseteq \alpha$, $\beta' \subseteq \beta$ can be derived for *Filter*. In the first overload of that type, the first parameter P can be any value, the second parameter L is an empty list, and the result type is the empty list. In the second functional type, the first parameter P is a predicate function, the second parameter a list L whose elements are contained by the type of the only parameter of P, and the result type is a subtype of the list L.

Similarly as in the previous example, the type

$\forall\ \alpha.\ \forall\ \beta.\ ((\alpha) \to \text{bool}(), [\beta]) \to [\beta']$ **when** $\beta \subseteq \alpha, \beta' \subseteq \beta$ cannot be a success type for this function, because when L is an empty list, P is not called, so it is not necessarily a function. When L is not an empty list, we know that the type of P is $(\alpha) \to \alpha'$ and variable B ends up with the following restrictions: $\alpha' \Leftarrow \alpha_B$ and $'\text{true}' \cup '\text{false}' \Leftarrow \alpha_B$. Since α' can be instantiated to a single value, the output of the predicate function must contain the bool() type to succeed in the innermost case distinction.

6.3.3. The Reverse function

The *Reverse* function takes a list and returns a new list with the same elements in reverse order. The code of this function is the following:

```
letrec
    Reverse = fun(L) → let K = [] in R2(L, K)
    R2 = fun(LT, A) → case LT of
        [] when 'true' → A
        [X|XS] when 'true'→
            let B = [X|A] in R2(XS, B)
    end
in Reverse
```

The type $\forall\ \alpha.\ ([\alpha]) \to [\alpha]$ is obtained for *Reverse* from a derivation with our type system, where the only parameter L is a list. The list reversal is done through an auxiliary function *R2* with two parameters, where the first parameter *LS* is a list and the second parameter *A* is an accumulator. Since the type we obtain for *R2* is $\forall\ \beta.\ ([], \beta) \to \beta \sqcup \forall\ \alpha, \beta.\ (\text{nelist}(\alpha, []), \beta) \to \text{nelist}(\alpha, \beta)$, a derivation for $R2(L, [])$ yields the following annotated type as result: $\langle \beta; \Gamma[L\colon \alpha_L, K\colon \alpha_K | [] \Leftarrow \alpha_K, [] \Leftarrow \alpha_L, \beta \Leftarrow \alpha_K] \rangle; \langle \text{nelist}(\alpha, \beta); \Gamma[L : \alpha_L, K\colon \alpha_K | [] \Leftarrow \alpha_K, \text{nelist}(\alpha, []) \Leftarrow \alpha_L, \beta \Leftarrow \alpha_K] \rangle$, where Γ contains the definitions for *Reverse* and *R2*. We can apply rule [SUB2] because of the following

$$\langle []; \Gamma[L\colon [], K\colon []] \rangle; \langle \text{nelist}(\alpha, []); \Gamma[L\colon \text{nelist}(\alpha, []), K\colon []] \rangle \subseteq \langle [\alpha]; \Gamma[L\colon [\alpha], K\colon []] \rangle;$$

Therefore, after simplifying the annotated type into one pair, the parameter L and the result of *Reverse* is $[\alpha]$. The occurrence of the same α in both input and output entails that the set of elements in the output list is the same as the set of elements in the input list.

6.3.4. The Nth function

The *Nth* function takes a number and a list, and returns the element from that list at the position specified by the number. Its code is as follows:

```
letrec Nth = fun(N, L) → let K = 1 in case L of
        [X|XS] when 'true'→
            case N of
                1 when 'true' → X
                Z when N > K → let M = N − K in Nth(M, XS)
            end
end in Nth
```

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

7. Correctness

In this section we shall prove that the types derived with the set of rules shown in Section 5 are success types for their corresponding expressions. The detailed proofs can be found in the Appendix.

Before getting into the correctness theorem, we need some auxiliary results. The first one states that if we rename the type variables occurring free in a type, the resulting type denotes the same sets of values, but with their instantiations modified accordingly. Given a renaming μ and an instantiation π, the notation $\pi\mu$ denotes the instantiation such that $(\pi\mu)(\alpha) = \pi(\mu^{-1}(\alpha))$ for every $\alpha \in rng\,\mu$ and $(\pi\mu)(\alpha) = \pi(\alpha)$ for every $\alpha \notin rng\,\mu$.

Lemma 1. *Let μ be a substitution from type variables to type variables. For every τ, C, v, and π such that $rng\ \mu \cap ftv(\tau) = \emptyset$ it holds that:*

1. $(v, \pi) \in \mathcal{T}[\![\tau]\!] \Longrightarrow (v, \pi\mu) \in \mathcal{T}[\![\tau\mu]\!]$
2. $\pi \vDash C \Longrightarrow \pi\mu \vDash C\mu$

Given a pair (v, π) belonging to the semantics of a type τ, only the type variables occurring free τ are relevant to the instantiation π. The rest of them could be remapped to different sets of values in π. This is what the following lemma formalises:

Lemma 2. *Let π and π' be two instantiations, τ a type and C a set of constraints.*

1. *For any value v, if $(v, \pi) \in \mathcal{T}[\![\tau]\!]$ and $\pi \equiv \pi'$ (modulo $ftv(\tau)$), then $(v, \pi') \in \mathcal{T}[\![\tau]\!]$.*
2. *$\pi \vDash C$ and $\pi \equiv \pi'$ (modulo $ftv(C)$) then $\pi' \vDash C$.*

In our type system there are some rules that allow us to derive types for patterns by obtaining judgements of the form $\Gamma \vdash p\colon \tau$. The intended meaning of these rules are given by the following result:

Lemma 3. *Assume an environment Γ, a pattern p and a type τ such that $\Gamma \vdash p\colon \tau$. For every instantiation π and value v such that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ and $(\theta, v) \in \mathcal{E}[\![p]\!]$ it holds that $(v, \pi) \in \mathcal{T}[\![\tau]\!]$.*

The main result states that, whenever we obtain a judgement of the form $\Gamma \vdash e\colon \rho$, the annotated ρ is a success type for e.

Theorem 1. *Assume an environment Γ, an expression e and an annotated type ρ. If $\Gamma \vdash e\colon \rho$ then*

$$\mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\rho]\!].$$

In the particular case in which our expression e is closed, our rules infer success types as defined in [11] which in turn generalises the definition given in [8].

Corollary 1. *For any closed expression e and annotated type ρ such that $[\,] \vdash e\colon \overline{\langle \tau_i; [\,] \rangle}^n$, then $\mathcal{E}[\![e]\!] \subseteq \mathcal{T}[\![\tau_1 \cup \ldots \cup \tau_n]\!]$.*

8. Comparison with other type systems

In this section we will compare success types with other type systems used for languages like JavaScript or Python. Some of those type systems follow a *gradual typing* [17,18] approach. For example: JavaScript has TypeScript [19] and Flow [20], and Python has MyPy [21]. The examples we will show in the section will be written in TypeScript and MyPy.

8.1. Getting types without annotations

Since TypeScript and MyPy are based on gradual type systems, they accept programs without type annotations. The type checker can find some potential problems in those programs without type annotations, but with some limitations. A basic example in JavaScript is:

```
let y = 3 - "2"; // y = 1
```

For TypeScript we get the following error:

```
The right-hand side of an arithmetic operation must be of type
'any', 'number', 'bigint' or an enum type.
```

This error assumes that the subtraction operator in JavaScript does not accept other types than numbers, which is not true according to the language standard, but TypeScript, assuming a conservative approach, gives a false positive for the sentence. Since this is more related to the tricky behaviour of operators in JavaScript, let us see a similar example in Python:

```
x = 3 - "2"
```

In MyPy we get the following error for this assignment:

```
error: Unsupported operand types for - ("int" and "str")
```

In this case if we try to subtract a string to a number in Python, the language will throw an exception and the program will crash. Both TypeScript and MyPy can give a type for literal values and check if they fit in the parameters expected by operators. However, if we use a function to encapsulate the code shown before:

```
function sub(a, b) {
 return a - b;
}
let x = sub(3, "foo");
```

TypeScript will infer for `sub` the type `(a:any,b:any):number`, for `x` the type `number`, and we will receive no error as in the previous example. We receive no error in the call to `sub` with a `string` as its second argument, because any value fits in the type `any`. Besides this, the variables passed as arguments to a function do not get additional type information that can be useful to catch mistakes when these variables are being applied to another function. For example, if `sub` had been defined as `sub(a:number,b:number):number`, the application `sub(x,y)` would not deduce the type `number` for the variables `x` and `y`. In Python we would have a similar situation:

```
def sub(a, b):
 return a - b
x = sub(3, "2")
```

The function `sum` in MyPy has the type `(a:Any,b:Any) -> Any`, so we face the same situation where a string fits in the type `Any`. With Flow the results are similar to TypeScript, but the error message blames the expression `a - b` inside `sub` when we obtain calls to `sub` with different types as in, for example, `sub(3, 2)` and `sub(3, "foo")`.

In our type system and in *Dialyzer*, program variables can have a type specified by the user. Those are contained within the assumption environment, but these specifications are not required, so we can have programs without type annotations and still derive types for then. Therefore the errors we have shown above would have been detected. Since our typing rules extract information from the applications for the program variables used as parameters in the call, we can infer additional type information for an abstraction, and therefore check for type errors when calling those abstractions.

8.2. Annotations and false positives

In the next example in TypeScript we will have a function that takes a list of values and returns only those that can be transformed into a number, packed into an object with their integer successor:

```
interface NextInt<T> {
 value: T;
 next: number;
}
function getNexts<T>(values: Array<T>): Array<NextInt<T>> {
 return values.map(x => {
 let temp = Number(x);
 if (isNaN(temp)) {
 return undefined;
 } else {
 return { value: x, next: Math.trunc(temp) + 1 };
 }
 })
 .filter(x => x !== undefined);
}
```

For TypeScript we get the following error:

```
Type '({ value: T; next: number; } | undefined)[]' is not assignable
to type 'NextInt<T>[]'.
```

The problem here is that —after executing the `map` function— we have an array of type `NextInt<T> | undefined` instead of `NextInt<T>`. For that reason we execute the filter function to remove the `undefined` values from the array, but the union type still remains after `filter`. A first solution would be to first execute `filter` using the conversion to check those elements in the array for which conversion succeeds, and then execute `map` using again the conversion to build the result:

```
function getNexts<T>(values: Array<T>): Array<NextInt<T>> {
 return values.filter(x => !isNaN(Number(x)))
 .map(x => ({ value: x, next: Math.trunc(Number(x)) + 1 }));
}
```

This leads us to execute the conversion function twice, which is inefficient and unnecessary. Another option to avoid the warning message is to execute another map to cast to `NextInt<T>` for each element in the array, but this is also unnecessary, because we can use a cast for the whole final result before we return it:

```
function getNexts<T>(values: Array<T>): Array<NextInt<T>> {
 return <Array<NextInt<T>>>values.map({...})
 .filter(x => x !== undefined);
}
```

In Flow the given error message is similar to TypeScript, but the problem cannot be solved with a simple casting, and therefore the code of the function must be changed using the first solution we have discussed for this example.

A similar example in Python using MyPy type annotations is as follows:

```
def toInteger(victim:Any) -> Optional[int]:
 try:
 return int(victim)
 except:
 return None
def makeTuple(value:T) -> Optional[Tuple[T, int]]:
 aux = toInteger(value)
 if type(aux) == int:
 return (value, aux + 1)
 else:
 return None
def getNexts(values:Iterable[T]) -> Iterable[Tuple[T, int]]:
 aux1 = map(makeTuple, values)
 aux2 = list(filter(lambda x: x != None, aux1))
 return aux2
```

Inside `makeTuple` we get the following error for expression `aux + 1`:

```
error: Unsupported operand types for + ("None" and "int")
note: Left operand is of type "Optional[int]"
```

And inside `getNexts` we get the following error in the return statement:

```
error: Argument 1 to "list" has incompatible type
"Iterator[Optional[Tuple[T, int]]]"; expected
"Iterable[Tuple[T, int]]"
```

The first error warns us that `aux` does not fit in the types for the plus operator, since the value `None` cannot be added to another number. Even if we use the condition `aux != None` in the `if` statement, the error message is thrown by the tool. In order to fix it we can set the type `Any` for `aux`:

```
aux:Any = toInteger(value)
```

Declaring `aux` as `Any`, MyPy uses the program variable as a dynamic one and does not check if it fits or not in the plus operator. After solving this error message, the second one is solved casting the final result into the final type:

```
return cast(Iterable[Tuple[T, int]], aux2)
```

We could also have executed a map and cast each element into `Tuple[T, int]`, but like in the TypeScript example it would have been unnecessarily inefficient.

A similar function to `getNexts` in Erlang would be:

```
getNexts(V) ->
 V2 = map(fun(X) ->
 case to_integer(X) of
 undefined -> undefined;
 N -> {X, N + 1}
 end, V),
 filter(fun(X) -> X =/= undefined end, V2).
```

In our type system the type $([\alpha]) \to [\{\alpha, \text{int}()\} \cup \text{'undefined'}]$ would be given for `getNexts`, like TypeScript and MyPy. But since the intersection between $[\{\alpha, \text{int}()\} \cup \text{'undefined'}]$ and $[\{\alpha, \text{int}()\}]$ is not empty, a success type system would not mark as an error if the user sets the type for `getNexts` as $([\alpha]) \to [\{\alpha, \text{int}()\}]$.

In contrast, both TypeScript and MyPy give false positives for safe programs, because the final goal of these tools is to adapt the static typing discipline to these languages. This can force the user to add extra type information and castings to fit the pieces into expected result types, or to create elaborated type hierarchies to keep things working. In the worst scenario, the adoption of this approach may remove flexibility to develop programs –unless the programmer chooses to disregard the warnings given by the type checking tool– and in other cases the programmer is forced to add additional hints to the type checker. With *success types* these examples would not be rejected, giving the user the option to set the input types for certain abstractions if the user wishes to do so, without having to be worried about false positives.

Overloaded functions, i.e. groups of functions having the same name and number of arguments, but different types for the input parameters, can be defined in languages like Java, but dynamically-typed languages do not support them. In languages like Erlang, JavaScript or Python, we can simulate this by dynamically checking at runtime the arguments' types to select one or another version. In our type system, and in *Dialyzer*, overloaded functional types can be defined as type specifications to represent the different execution branches. However, TypeScript and MyPy do not take overloaded functional types into account, and we can only have one functional type encompassing all the execution branches of the function.

9. Related work

A significant amount of research has been carried out in order to apply type-based static analysis to dynamically typed languages. As explained in the previous section, well-known examples of this are Typescript [19], and Python [22], but these techniques have also been applied to other languages, such as Ruby [23]. Although [22] is oriented towards the translation of Python into JVM and CLI primitive instructions (instead of emulating the Python model on top of the corresponding virtual machine), these systems allow the programmer to catch type errors at compile time. However, they follow the traditional approach of ensuring the absence of type errors at runtime, even if some false positives are reported. The type system introduced in this paper follows the opposite goal introduced by success types [8], that is, to avoiding false positives. Our goal is to assist the programmer in detecting as many definite errors as possible. Although some other subtler type errors may be left unreported, this approach can be combined with the variety of mechanisms that Erlang provides (such as supervision trees) for reporting and restarting the program state in the event of crashes.

Another approach to apply type-based static analysis is *soft typing* [24], which is a technique to find those places in a program where type consistency is not guaranteed, in order to insert run-time type checks. This approach shares Dialyzer's philosophy insofar it does not require type annotations from the programmer. A soft type checker does not reject programs with potential type errors, but unlike success typing, it is conservative in the sense that it inserts type checks whenever in doubt. Some implementations of soft type systems have been developed for Scheme [25] and Erlang [26], the latter introducing a specification language for specifying the interface of Erlang modules. As acknowledged by its author, the latter system might produce false positives such as in function `lists:nth/2` when there is no guarantee that the list is accessed within its bounds. Another difference of our system with respect to soft typing is the addition of type environments in annotated types, which capture the necessary conditions for the evaluation of the expression. Also the constraints related to a type in our system is another difference that improves the polymorphic functional types.

Another area of research related to the integration of static typing into dynamically typed languages is that of *gradual typing* [17,18]. Unlike the all-or-nothing approach provided by traditional languages, a gradual type system allows programmers to partially annotate their programs with types, while the unannotated parts of the program have implicitly a dynamic type, which roughly corresponds to the any() type in this work. Gradual type systems have also been studied in the context of imperative languages [27,28]. There exists an inference algorithm for gradual types [29] which consists in constraining the types of variables from their definitions and assignments (*inflows*) and from the context in which they appear (*outflows*). The latter bound the set of values which a variable may contain at runtime, in a similar way as our type environments represent an upper bound of the values of all the variables in scope. However, the goal of the gradual typing inference is not to detect type discrepancies, but to carry out performance optimizations, in the same way as soft type systems. In this sense, we can say that the type system presented here is closer to the notion of success types which we extend in order to obtain polymorphic types.

Although type systems that overapproximate runtime behaviours are rarely to be found in functional languages (with the exception of success types in Erlang), this idea has been widely studied in the context of logic programming with Prolog. Heintze and Jaffar [30] introduced a transformation of logic programs into sets of constraints that approximate their behaviour, and a method for simplifying such sets. Another approach to capture the meaning of a logic program is based on *regular types* [31,32], which can be expressed by unary logic programs. Type checking is decidable for regular types, and inference is feasible by using bottom-up abstract interpretation techniques with a suitable widening operator [33,34]. Besides this, in [35,36] the call-success

semantics for Constraint Logic Programming (CLP) —with a defined type system— are introduced and verification conditions for those semantics are provided in order to statically detect errors on programs.

The logic programming-related approaches mentioned so far involve only monomorphic types. Polymorphism is introduced by Barbuti and Giacobazzi [37], who use abstract interpretation to infer, given a predicate definition, an abstract success set that may contain type variables. However, this set does not overapproximate the set of successes of the predicate being defined; some refutable goals might be ill-typed. The authors also briefly describe how to introduce union types in their system. Their approach is subsequently refined by Lu [38,39], who devises an analysis for inferring dependencies among the parameters of a logic predicate. Such dependencies are expressed with rules involving polymorphic variables and union types. Unlike [37], the results soundly overapproximate the success set of the programs. In the same way as in our system, type information is propagated in function symbols from the arguments to the result and backwards. This propagation is explicitly stated by the dependencies being inferred by the analysis, whereas in our type system this information is implicit in the typing environments. A substantial difference of our type system with respect to these approaches is that constraints on type variables cannot be expressed in the latter.

With regard to success types, Jakob and Thiemann [6] formalise a success type system in which functional types include constraints that determine the failure of the function. If the constraints are satisfied, then the function will definitely fail. In our system we follow the opposite approach: those final environments whose constraints are *not* satisfied for a given assignment from variables to values are semantically vacuous. The types that can be derived with [6] and the sets of constraints determining failure are expressed recursively, since the authors focus on errors involving recursive data structures other than lists. The satisfiability of these sets of constraints is undecidable in general.

10. Conclusions

We have presented a set of typing rules for a significant subset of Core Erlang. These rules allow us to derive polymorphic type specifications, which can be overloaded in the sense of [9], and hence can capture the semantics of a function in more accurate way than our previous work. Formally, the type judgements derived by our rules obtain, under a given type environment, an annotated type for an expression e. Inside each pair of the obtained annotated type, we can find a new type environment expressing conditions for the free variables in e that are necessary for the successful evaluation of e, and a set of type constraints for the type variables of the given type in the same pair. When the rules are applied to closed expressions, they derive success types, i.e., overapproximations of the semantics.

The syntax of types presented in this paper involves the existence of universally quantified types nested inside other functional types, similar to the types in System F [40]. Although type inference in System F is undecidable, in our context this problem becomes trivial, as we can always derive a type for an expression. In fact, we can always derive the any() type, as stated in Proposition 2. The problem of inferring an *accurate* type for a given expression is more involved, and hence left as future work, for which the set of rules presented in this paper will hopefully provide a solid foundation.

One of the motivations of our work was to overcome some of the limitations of Dialyzer as a static type analysis tool for Erlang. But it is obvious that this work, by itself alone, cannot considered as a real alternative to it. Dialyzer is a quite well engineered tool and its authors have been able to integrate it in the life cycle of real Erlang programs, without leaving out any part of the language. In order for our work to complement or replace the role of Dialyzer, there are still some steps to be taken. We have already mentioned the most important and technically interesting one, which is developing a polymorphic success type inference system with a good compromise between accuracy (it should infer not too trivial types and take sensible advantage of polymorphism and overloading) and efficiency (it should be applicable to real programs). Moreover, the type system, at the levels both of derivation and inference, must be extended to the whole Erlang language, so that it can be used for any Erlang program. After completing those two steps and checking the results for a set of real programs, we will be in a position of trying to disseminate our tool within the Erlang community, so that its integration into Erlang as a system could be considered.

Another future direction of this research will be to adapt these ideas to other dynamically-typed imperative languages, such as JavaScript or Python.

Appendix A. Appendix: Correctness proofs

Lemma 4. *Assume an instantiation* π, *a type* τ *and a set of instantiations* Π *such that* $\pi \in Dcp(\Pi, \tau)$. *Assume a renaming* μ *such that* $dom\ \mu \setminus rng\ \mu = \{\alpha_1, ..., \alpha_n\}$. *Then:*

$\pi\mu \in Dcp(\{(\pi'\mu)[\overline{\alpha_i \mapsto \pi(\alpha_i)}] | \pi' \in \Pi\}, \tau\mu)$

Proof. Let us denote by Π° the set $\{(\pi'\mu)[\overline{\alpha_i \mapsto \pi(\alpha_i)}] | \pi' \in \Pi\}$. We have to prove that:

$$\pi\mu = \bigcup \Pi^\circ \tag{A.1}$$

$$\pi\mu \equiv \pi^\circ \text{ (modulo } \mathbf{TypeVar} \setminus itv(\tau\mu)) \text{ for each } \pi^\circ \in \Pi^\circ \tag{A.2}$$

In order to prove (A.1) let us assume a variable α. If $\alpha \in rng\,\mu$ then:

$$\begin{aligned}
&(\pi\mu)(\alpha)\\
&\quad = \pi(\mu^{-1}(\alpha))\\
&\quad = \bigcup\{\pi'(\mu^{-1}(\alpha)) | \pi' \in \Pi\}\\
&\quad = \bigcup\{(\pi'\mu)(\alpha) | \pi' \in \Pi\}\\
&\quad = \bigcup\{(\pi'\mu)[\overline{\alpha_i \mapsto \pi(\alpha_i)}](\alpha) | \pi' \in \Pi\}\\
&\quad = \bigcup\{\pi^\circ(\alpha) | \pi^\circ \in \Pi^\circ\}
\end{aligned}$$

If $\alpha \notin rng\,\mu$ but $\alpha \in dom\,\mu$ then:

$$\begin{aligned}
&(\pi\mu)(\alpha)\\
&\quad = \pi(\alpha)\\
&\quad = \bigcup\{\pi^\circ(\alpha) | \pi^\circ \in \Pi^\circ\}
\end{aligned}$$

the last step justified by the fact that $\alpha \in dom\,\mu \backslash rng\,\mu$ and hence $\pi^\circ(\alpha) = \pi(\alpha)$ for each $\pi^\circ \in \Pi^\circ$. Finally, if $\alpha \notin rng\,\mu$ and $\alpha \notin dom\,\mu$ we get:

$$
\begin{aligned}
&(\pi\mu)(\alpha)\\
&\quad = \pi(\alpha)\\
&\quad = \bigcup\{\pi'(\alpha) | \pi' \in \Pi\}\\
&\quad = \bigcup\{(\pi'\mu)(\alpha) | \pi' \in \Pi\}\\
&\quad = \bigcup\{(\pi'\mu)[\overline{\alpha_i \mapsto \pi(\alpha_i)}](\alpha) | \pi' \in \Pi\}\\
&\quad = \bigcup\{\pi^\circ(\alpha) | \pi^\circ \in \Pi^\circ\}
\end{aligned}
$$

Now let us prove (A.2). Assume some $\pi^\circ \in \Pi^\circ$. We know that $\pi^\circ = (\pi'\mu)[\overline{\alpha_i \mapsto \pi(\alpha_i)}]$ for some $\pi' \in \Pi$. Assume a variable $\alpha \notin itv(\tau\mu)$ we have to prove that $(\pi\mu)(\alpha) = \pi^\circ(\alpha)$. If $\alpha \in rng\,\mu$ we get that $\mu^{-1}(\alpha) \notin itv(\tau)$ and hence $\pi(\mu^{-1}(\alpha)) = \pi'(\mu^{-1}(\alpha))$. Therefore:

$$(\pi\mu)(\alpha) = \pi(\mu^{-1}(\alpha)) = \pi'(\mu^{-1}(\alpha)) = (\pi'\mu)(\alpha) = \pi^\circ(\alpha)$$

On the contrary, if $\alpha \notin rng\,\mu$ but $\alpha \in dom\,\mu$ we get

$$(\pi\mu)(\alpha) = \pi(\alpha) = \pi^\circ(\alpha)$$

since $\alpha \in dom\,\mu \backslash rng\,\mu$. Finally, in the case in which α belongs to neither $dom\,\mu$ nor $rng\,\mu$ we get that $\alpha \notin itv(\tau)$ and hence:

$$(\pi\mu)(\alpha) = \pi(\alpha) = \pi'(\alpha) = (\pi'\mu)(\alpha) = \pi^\circ(\alpha)$$

□

In the following, given a set X of type variables and a renaming μ, we denote by $X\mu$ the result of applying the renaming to all the variables occurring inside X. More concretely:

$$X\mu = \{\mu(\alpha) | \alpha \in X,\ \alpha \in dom\ \mu\} \cup \{\alpha | \alpha \in X,\ \alpha \notin dom\ \mu\}$$

The following lemma states some properties of this definition:

Lemma 5. *Assume a renaming μ.*

1. *For every type τ, $itv(\tau)\mu = itv(\tau\mu)$ and $ftv(\tau)\mu = ftv(\tau\mu)$.*
2. *For every X, $Y \subseteq$ **TypeVar** such that $rng\ \mu \cap (X \cup Y) = \emptyset$ it holds that $(X \backslash Y)\mu = X\mu \backslash Y\mu$.*
3. *For every X, $Y \subseteq$ **TypeVar** it holds that $(X \cup Y)\mu = X\mu \cup Y\mu$.*

Proof. Let us prove the first property. Assume a type variable $\alpha \in itv(\tau)\mu$. From the definition of this set we distinguish two cases:

- If $\alpha = \mu(\beta)$ such that $\beta \in itv(\tau)$, then $\alpha \in itv(\tau\mu)$.
- If $\alpha \in itv(\tau)$ with $\alpha \notin dom\,\mu$, then $\alpha \in itv(\tau\mu)$.

Now assume that $\alpha \in itv(\tau\mu)$. There are two possibilities:

- α has appeared in $itv(\tau\mu)$ as a consequence of some β occurring in $itv(\tau)$ that has been renamed into α by μ. In this case we have that $\alpha = \mu(\beta) \in itv(\tau)\mu$.
- α has appeared in $itv(\tau\mu)$ because it was already in τ and it has been left untouched by μ. This means that $\alpha \notin dom\,\mu$ and hence $\alpha \in itv(\tau)\mu$.

We could repeat the same reasoning above with $ftv(\tau)$ instead of $itv(\tau)$. Now let us prove the second property. Assume that $\alpha \in (X \backslash Y)\mu$. According to the definition of the latter, there are two possibilities:

- $\alpha = \mu(\beta)$ for some $\beta \in X \backslash Y$. Since $\beta \in X$ we get that $\alpha \in X\mu$. Now let us prove that $\alpha \notin Y\mu$ by contradiction. Assume that $\alpha \in Y\mu$.
  - If $\alpha = \mu(\beta')$ for some $\beta' \in Y$, then we would have that $\beta' = \beta$ because μ is injective, and hence $\beta \in Y$ which contradicts $\beta \in X \backslash Y$.
  - If $\alpha \in Y$ and $\alpha \notin dom\,\mu$, then we contradict the fact that $Y \cap rng\ \mu = \emptyset$.
  Therefore, $\alpha \notin Y\mu$, so $\alpha \in X\mu \backslash Y\mu$.
- $\alpha \in X \backslash Y$ and $\alpha \notin dom\,\mu$. Since $\alpha \in X$ we get in this case that $\alpha \in X\mu$. Now let us prove, again by contradiction, that $\alpha \notin Y\mu$. Assume that $\alpha \in Y\mu$.
  - If $\alpha = \mu(\beta)$ for some $\beta \in Y$ then we would contradict the fact that $X \cap rng\ \mu = \emptyset$.
  - If $\alpha \in Y$ and $\alpha \notin dom\,\mu$ the former contradicts the fact that $\alpha \in X \backslash Y$.
  Therefore $\alpha \notin Y\mu$ and hence $\alpha \in X\mu \backslash Y\mu$.

We have then proved that $(X \backslash Y)\mu \subseteq X\mu \backslash Y\mu$. Now we prove the opposite inclusion. Assume $\alpha \in X\mu \backslash Y\mu$. The fact that $\alpha \notin Y\mu$ implies the following:

$$\alpha \notin \{\mu(\beta) | \beta \in Y,\ \beta \in dom\ \mu\} \tag{A.3}$$

$$\alpha \notin \{\alpha | \alpha \in Y,\ \alpha \notin dom\ \mu\} \tag{A.4}$$

With the fact that $\alpha \in X\mu$ we distinguish two cases:

- If $\alpha = \mu(\beta)$ for some $\beta \in X$ and $\beta \in dom\,\mu$, this implies that $\beta \notin Y$, since otherwise we would contradict (A.3). Therefore $\beta \in X \backslash Y$ and $\alpha \in (X \backslash Y)\mu$.
- If $\alpha \in X$ and $\alpha \notin dom\,\mu$ then we need $\alpha \notin Y$ in order not to contradict (A.4). Therefore $\alpha \in X \backslash Y$ and $\alpha \in (X \backslash Y)\mu$.

Therefore, we have proved $(X \backslash Y)\mu \supseteq X\mu \backslash Y\mu$ and hence the equality between both sets.

Finally, let us prove the third property. We first prove that $(X \cup Y)\mu \subseteq X\mu \cup Y\mu$. Assume that $\alpha \in (X \cup Y)\mu$. We distinguish cases:

- Assume that $\alpha = \mu(\beta)$ for some $\beta \in X \cup Y$. If $\beta \in X$ we get that $\alpha \in X\mu \subseteq X\mu \cup Y\mu$. Similarly with the case $\beta \in Y$.
- Assume that $\alpha \in X \cup Y$ and $\alpha \notin dom\,\mu$. If $\alpha \in X$ then $\alpha \in X\mu$. If $\alpha \in Y$ then $\alpha \in Y\mu$. Therefore, $\alpha \in X\mu \cup Y\mu$.

Now we prove that $X\mu \cup Y\mu \subseteq (X \cup Y)\mu$. Assume that $\alpha \in X\mu \cup Y\mu$. We only prove the case in which $\alpha \in X\mu$, since the case $\alpha \in Y\mu$ is similar. We have two possibilites:

- If $\alpha = \mu(\beta)$ for some $\beta \in X$, then $\beta \in X \cup Y$ and hence $\alpha \in (X \cup Y)\mu$.
- If $\alpha \in X$ and $\alpha \notin dom\,\mu$, then $\alpha \in X \cup Y$ and hence $\alpha \in (X \cup Y)\mu$.

Therefore, $X\mu \cup Y\mu \subseteq (X \cup Y)\mu$ and we have proved the equality. □

Lemma 6. *Assume two instantiations π, π', a renaming μ, and a set X of type variables such that $\pi \equiv \pi'$ (modulo* **TypeVar**$\setminus X$*). Let us denote by $\{\overline{\alpha_i}\}$ the set of type variables in $dom\,\mu \setminus rng\,\mu$. Then:*

$\pi\mu[\overline{\alpha_i \mapsto \varnothing}] \equiv \pi'\mu[\overline{\alpha_i \mapsto \varnothing}]$(modulo**TypeVar**$\setminus X\mu$)

Proof. For the sake of brevity, let us denote by π_0 and π_1 the instantiations $\pi\mu[\overline{\alpha_i \mapsto \pi(\alpha_i)}]$ and $\pi'\mu[\overline{\alpha_i \mapsto \pi(\alpha_i)}]$ respectively. Assume a variable $\alpha \notin X\mu$. We prove that $\pi_0(\alpha) = \pi_1(\alpha)$. If $\alpha \in rng\,\mu$ we get that $\mu^{-1}(\alpha) \notin X$ and hence:

$\pi_0(\alpha) = \pi(\mu^{-1}(\alpha)) = \pi'(\mu^{-1}(\alpha)) = \pi_1(\alpha)$

If $\alpha \notin rng\,\mu$ but $\alpha \in dom\,\mu$ we get that $\alpha \in dom\,\mu \setminus rng\,\mu$. Therefore:

$\pi_0(\alpha) = \varnothing = \pi_1(\alpha)$

Finally, if $\alpha \notin rng\,\mu$ and $\alpha \notin dom\,\mu$ we get that $\alpha \notin X$ and hence:

$\pi_0(\alpha) = \pi(\alpha) = \pi'(\alpha) = \pi_1(\alpha)$

□

Lemma 7. *Assume a renaming μ and a set of variables X such that $X \cap rng\ \mu = \varnothing$. Then, for every instantiation π, if $\pi \equiv [\,]$ (modulo X), then $\pi\mu \equiv [\,]$ (modulo $X\mu$).*

Proof. Assume a variable $\alpha \in X\mu$. We have to prove that $(\pi\mu)(\alpha) = \varnothing$. According to the definition of $X\mu$ we have two possibilities:

- $\alpha = \mu(\beta)$ for some $\beta \in X$. Then, by assumption, $\pi(\beta) = \varnothing$. We get:

  $(\pi\mu)(\alpha) = \pi(\mu^{-1}(\alpha)) = \pi(\beta) = \varnothing$

- $\alpha \in X$ with $\alpha \notin dom\,\mu$. Since X and $rng\,\mu$ are disjoint we get that $\alpha \notin rng\,\mu$, which entails $(\pi\mu)(\alpha) = \pi(\alpha)$. From the assumption it follows that $\pi(\alpha) = \varnothing$.

□

Lemma 1. *Let μ be a substitution from type variables to type variables. For every τ, C, v, and π such that $rng\ \mu \cap ftv(\tau) = \varnothing$ it holds that:*

1. $(v, \pi) \in \mathcal{T}[\![\tau]\!] \Longrightarrow (v, \pi\mu) \in \mathcal{T}[\![\tau\mu]\!]$
2. $\pi \vDash C \Longrightarrow \pi\mu \vDash C\mu$

Proof. By induction on the structure of τ and C. The cases in which τ is a simple ground type (singleton, none(), any(), integer(), etc.) are trivial. Without loss of generality, let us assume that $dom\,\mu \subseteq ftv(\tau)$. Otherwise we would restrict the domain of μ to $ftv(\tau)$ in order to obtain μ' and prove the property on μ' instead of μ. In this way, we would get $(v, \pi\mu) \in \mathcal{T}[\![\tau\mu]\!] \Leftrightarrow (v, \pi\mu') \in \mathcal{T}[\![\tau\mu']\!]$ because of Lemma 2 and the fact that $\mathcal{T}[\![\tau\mu']\!] = \mathcal{T}[\![\tau\mu]\!]$. We consider now the remaining cases.

- **Case** $\tau = \alpha$. If $(v, \pi) \in \mathcal{T}[\![\alpha]\!]$ then $\pi(\alpha) = \{v\}$. Since $\alpha \notin rng\,\mu$ we know that $(\pi\mu)(\alpha) = \{v\}$, so if $\alpha \notin dom\,\mu$ we get $(v, \pi\mu) \in \mathcal{T}[\![\alpha]\!] = \mathcal{T}[\![\alpha\mu]\!]$. If $\alpha \in dom\,\mu$ let us assume that $\mu(\alpha) = \beta$. Hence $(\pi\mu)(\beta) = \pi(\alpha) = \{v\}$ and therefore $(v, \pi\mu) \in \mathcal{T}[\![\beta]\!] = \mathcal{T}[\![\alpha\mu]\!]$.
- **Case** $\tau = \{\overline{\tau_i}^n\}$. Assume that $(v, \pi) \in \mathcal{T}[\![\{\overline{\tau_i}^n\}]\!]$, then $v = \{\overline{v_i}^n\}$ for some $\overline{v_i}^n$ such that $(v_i, \pi) \in \mathcal{T}[\![\tau_i]\!]$ for each $i \in \{1..n\}$. By induction hyphotesis we get that $(v_i, \pi\mu) \in \mathcal{T}[\![\tau_i\mu]\!]$ for every $i \in \{1..n\}$ and therefore $(v, \pi\mu) \in \mathcal{T}[\![\tau\mu]\!]$.
- **Case** $\tau = \text{nelist}(\tau_1, \tau_2)$. In this case we get that $v = [v_1, ..., v_n|v']$ such that $(v_i, \pi_i) \in \mathcal{T}[\![\tau_1]\!]$ for every $i \in \{1..n\}$ such that $\pi \in Dcp(\{\overline{\pi_i}^n\}, \tau_1)$ and $(v', \pi) \in \mathcal{T}[\![\tau_2]\!]$. By i.h. we get that $(v_i, \pi_i\mu) \in \mathcal{T}[\![\tau_1\mu]\!]$ for every $i \in \{1..n\}$ and $(v', \pi\mu) \in \mathcal{T}[\![\tau_2\mu]\!]$. Now assume that $dom\ \mu \setminus rng\ \mu = \{\overline{\alpha_i}\}$. Then none of the α_i occurs free in $\tau_1\mu'$. Hence we can use Lemma 2 to ensure that $(v_i, (\pi_i\mu)[\overline{\alpha_i \mapsto \pi_i(\alpha_i)}]) \in \mathcal{T}[\![\tau_1\mu]\!]$ for each $i \in \{1..n\}$. Since we know that $\pi \in Dcp(\{\overline{\pi_i}^n\}, \tau_1)$ we can apply Lemma 4 to obtain

  $\pi\mu \in Dcp(\{(\pi_i\mu)[\overline{\alpha_i \mapsto \pi_i(\alpha_i)}] | i \in \{1..\ n\}\}, \tau_1\mu)$

  And therefore $(v, \pi\mu) \in \mathcal{T}[\![\text{nelist}(\tau_1\mu, \tau_2\mu)]\!] = \mathcal{T}[\![\text{nelist}(\tau_1, \tau_2)\mu]\!]$.
- **Case** $\tau = \tau_1 \cup \tau_2$. Assume that $(v, \pi) \in \mathcal{T}[\![\tau_1]\!] \setminus (itv(\tau_2) \setminus itv(\tau_1))$. The case $(v, \pi) \in \mathcal{T}[\![\tau_2]\!] \setminus (itv(\tau_1) \setminus itv(\tau_2))$ is proved similarly. Then there exists some π' such that $(v, \pi') \in \mathcal{T}[\![\tau_1]\!]$ and

  $\pi \equiv \pi'$(modulo **TypeVar**$\setminus (itv(\tau_2) \setminus itv(\tau_1))$)
  $\pi \equiv [\,]$(modulo $(itv(\tau_2) \setminus itv(\tau_1))$)

Assume that $dom\ \mu \setminus rng\ \mu = \{\overline{\alpha_i}\}$ for some $\overline{\alpha_i}$. We can apply Lemma 6 to the first congruence and, since $rng\,\mu$ is disjoint from $itv(\tau_2)$, we can apply Lemma 7 to the second congruence. We get:

$$\pi\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv \pi'\mu[\overline{\alpha_i \mapsto \emptyset}]\,(\text{modulo } \mathbf{TypeVar} \setminus (itv(\tau_2) \setminus itv(\tau_1))\mu)$$
$$\pi\mu \equiv [\,]\,(\text{modulo } (itv(\tau_2) \setminus itv(\tau_1))\mu)$$

We can transform these congruences by using Lemma 5 and the fact that if $\pi \equiv [\,]$ modulo some set we get that $\pi[\overline{\alpha_i \mapsto \emptyset}] \equiv [\,]$ modulo the same set:

$$\pi\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv \pi'\mu[\overline{\alpha_i \mapsto \emptyset}]\,(\text{modulo } \mathbf{TypeVar} \setminus (itv(\tau_2\mu) \setminus itv(\tau_1\mu)))$$
$$\pi\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv [\,]\,(\text{modulo } (itv(\tau_2\mu) \setminus itv(\tau_1\mu)))$$

Now we knew that $(v, \pi') \in \mathcal{T}[\![\tau_1]\!]$. By i.h. we get $(v, \pi'\mu) \in \mathcal{T}[\![\tau_1\mu]\!]$. Since the variables in $dom\,\mu \setminus rng\,\mu$ do not occur in $\tau_1\mu$, we can use Lemma 2 to obtain $(v, \pi'\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{T}[\![\tau_1\mu]\!]$. The latter, with the equations shown above, leads to $(v, \pi\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{T}[\![\tau_1\mu]\!] \setminus (itv(\tau_2\mu) \setminus itv(\tau_1\mu))$, so finally $(v, \pi\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{T}[\![\tau_1\mu \cup \tau_2\mu]\!] = \mathcal{T}[\![(\tau_1 \cup \tau_2)\mu]\!]$. Since the variables $\overline{\alpha_i}$ do not appear free in $(\tau_1 \cup \tau_2)\mu$, we can use Lemma 2 to obtain $(v, \pi\mu) \in \mathcal{T}[\![(\tau_1 \cup \tau_2)\mu]\!]$.

- **Case** $\tau = \tau_1$ **when** C. This follows directly from applying induction hypothesis on its constituents.
- **Case** $\tau = \bigsqcup_{i=1}^{l} \sigma_i$. Firstly let us prove the following fact for every n-ary graph v, every instantiation π, every renaming μ, and all $\overline{\tau_i}^n$, τ' such that $rng\ \mu \cap ftv((\overline{\tau_i}^n) \to \tau) = \emptyset$:

$$(v, \pi) \in \mathcal{F}[\![(\overline{\tau_i}^n) \to \tau']\!] \Longrightarrow (v, \pi\mu) \in \mathcal{F}[\![((\overline{\tau_i}^n) \to \tau')\mu]\!] \tag{A.5}$$

Assume some $(v, \pi) \in \mathcal{F}[\![(\overline{\tau_i}^n) \to \tau']\!]$. According to the definition of $\mathcal{F}[\![\_]\!]$ there are two cases. In the first one, we have $v = \emptyset$ and $\pi \equiv [\,]$ (modulo $itv((\overline{\tau_i}^n) \to \tau')$). We can apply Lemma 7 in order to obtain $\pi\mu \equiv [\,]$ (modulo $itv((\overline{\tau_i}^n) \to \tau')\mu$) and then Lemma 5 in order to obtain $\pi\mu \equiv [\,]$ (modulo $itv(((\overline{\tau_i}^n) \to \tau')\mu)$). Therefore, $(v, \pi\mu) = (\emptyset, \pi\mu) \in \mathcal{F}[\![((\overline{\tau_i}^n) \to \tau')\mu]\!]$. In the second case, we get that $v = f|_1$ and $\pi \in Dcp(f|_2, (\overline{\tau_i}^n) \to \tau')$ for some f such that,

$$\emptyset \subset f \subseteq \{((\overline{v_i}^n, v), \pi') \mid \forall\, i \in \{1.\ n\}.\ (v_i, \pi') \in \mathcal{T}[\![\tau_i]\!],\ (v, \pi') \in \mathcal{T}[\![\tau']\!]\}$$

Let us define $f\mu$ as follows:

$$f\mu = \{((\overline{v_i}^n, v), \pi'\mu[\overline{\alpha_i \mapsto \pi(\alpha_i)}]) \mid ((\overline{v_i}^n, v), \pi') \in f\}$$

where $\{\overline{\alpha_i}\}$ is the set of type variables in $dom\,\mu \setminus rng\,\mu$. For every pair $((\overline{v_i}^n, v), \pi') \in f$ it holds that $(v_i, \pi') \in \mathcal{T}[\![\tau_i]\!]$ for each $i \in \{1.n\}$ and $(v, \pi') \in \mathcal{T}[\![\tau']\!]$. By induction hypothesis we get that $(v_i, \pi'\mu) \in \mathcal{T}[\![\tau_i\mu]\!]$ for each $i \in \{1.n\}$ and $(v, \pi'\mu) \in \mathcal{T}[\![\tau'\mu]\!]$ and, since the variables $\{\overline{\alpha_i}\}$ do not occur free in $((\overline{\tau_i}^n) \to \tau')\mu$ we can apply Lemma 2 so as to get that $(v_i, \pi'\mu[\alpha_i \mapsto \pi(\alpha_i)]) \in \mathcal{T}[\![\tau_i\mu]\!]$ for each $i \in \{1.n\}$ and $(v, \pi'\mu[\overline{\alpha_i \mapsto \pi(\alpha_i)}]) \in \mathcal{T}[\![\tau'\mu]\!]$. Therefore, we have proved that:

$$\emptyset \subset f\mu \subseteq \{((\overline{v_i}^n, v), \pi'') \mid \forall\, i \in \{1.\ n\}.\ (v_i, \pi'') \in \mathcal{T}[\![\tau_i\mu]\!],\ (v, \pi'') \in \mathcal{T}[\![\tau'\mu]\!]\}$$

Moreover, we can apply Lemma 4 to the fact $\pi \in Dcp(f|_2, (\overline{\tau_i}^n) \to \tau')$ in order to obtain . Therefore, $(v, \pi\mu) = (f|_1, \pi\mu) = (f\mu|_1, \pi\mu) \in \mathcal{F}[\![((\overline{\tau_i}^n) \to \tau')\mu]\!]$ and we have proved (A.5).Now we prove the following fact for every v, π, μ, α, $\overline{\tau_i}^n$, τ' such that $rng\ \mu \cap ftv((\overline{\tau_i}^n) \to \tau') = \emptyset$:

$$(v, \pi) \in \mathcal{S}[\![\forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau']\!] \Longrightarrow (v, \pi\mu) \in \mathcal{S}[\![(\ \forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau')\mu]\!] \tag{A.6}$$

Firstly let us assume that α does not belong to either $dom\,\mu$ or $rng\,\mu$. In this case we would have that $(\ \forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau')\mu = \forall\ \alpha.\ ((\overline{\tau_i}^n) \to \tau')\mu$. Assume that $(v, \pi) \in \mathcal{S}[\![\forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau']\!]$, then we know that $(v, \pi') \in \mathcal{F}[\![(\overline{\tau_i}^n) \to \tau']\!]$ for some π' such that $\pi' \equiv \pi$ (modulo $\mathbf{TypeVar}\setminus\{\alpha\}$). Since $ftv((\overline{\tau_i}^n) \to \tau') \subseteq ftv(\forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau') \cup \{\alpha\}$ and α does not belong to $rng\,\mu$ we can ensure that $ftv((\overline{\tau_i}^n) \to \tau')$ is disjoint from $rng\,\mu$ and hence apply (A.5) in order to obtain $(v, \pi'\mu) \in \mathcal{F}[\![((\overline{\tau_i}^n) \to \tau')\mu]\!]$. Now let us denote by $\{\overline{\alpha_i}\}$ the variables in the set $dom\,\mu rng\,\mu$. Those variables do not appear free in $((\overline{\tau_i}^n) \to \tau')\mu$, so we can apply Lemma 2 so as to get $(v, \pi'\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{F}[\![((\overline{\tau_i}^n) \to \tau')\mu]\!]$. We can apply Lemma 6 to the fact that $\pi' \equiv \pi$ (modulo $\mathbf{TypeVar}\setminus\{\alpha\}$) in order to obtain $\pi'\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv \pi\mu[\overline{\alpha_i \mapsto \emptyset}]$ (modulo $\mathbf{TypeVar}\setminus\{\alpha\}\mu$) or, since α is not in the domain of μ, $\pi'\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv \pi\mu[\overline{\alpha_i \mapsto \emptyset}]$ (modulo $\mathbf{TypeVar}\setminus\{\alpha\}$). By the definition of $\mathcal{S}[\![\_]\!]$ we know that $(v, \pi\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{S}[\![\forall\ \alpha.\ ((\overline{\tau_i}^n) \to \tau')\mu]\!] = \mathcal{S}[\![(\ \forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau')\mu]\!]$. Again, since the $\overline{\alpha_i}$ are not free in the type scheme (since α do not occur in them) we can apply Lemma 2 to obtain $(v, \pi\mu) \in \mathcal{S}[\![(\ \forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau')\mu]\!]$, as we wanted to prove.Now let us assume that α does belong to $dom\,\mu \cup rng\,\mu$. In this case we have $(\ \forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau')\mu = \forall\ \beta.\ ((\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta])\mu$ where β is a fresh variable not appearing in $dom\ \mu \cup rng\ \mu \cup ftv((\overline{\tau_i}^n) \to \tau')$. Assume that $(v, \pi) \in \mathcal{S}[\![\forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau']\!]$, then we know that $(v, \pi') \in \mathcal{F}[\![(\overline{\tau_i}^n) \to \tau']\!]$ for some π' such that $\pi' \equiv \pi$ (modulo $\mathbf{TypeVar}\setminus\{\alpha\}$). Since β is fresh, we can apply property (A.5) to obtain $(v, \pi'[\alpha/\beta]) \in \mathcal{F}[\![(\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta]]\!]$. Now that we know that α does not appear free in this functional type after applying the renaming $[\alpha/\beta]$ we can, on the one hand, apply Lemma 2 to obtain $(v, \pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]) \in \mathcal{F}[\![(\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta]]\!]$ and, on the other hand, ensure that $rng\,\mu$ is disjoint from the free variables in that type. Hence let us apply (A.5) again to obtain $(v, \pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu) \in \mathcal{F}[\![((\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta])\mu]\!]$. By Lemma 2 we get that $(v, \pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{F}[\![((\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta])\mu]\!]$ where $\overline{\alpha_i}$ are those variables occurring in $dom\,\mu \setminus rng\,\mu$. Now we apply Lemma 6 to $\pi' \equiv \pi$ (modulo $\mathbf{TypeVar}\setminus\{\alpha\}$) and get $\pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)] \equiv \pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]$ (modulo $\mathbf{TypeVar}\setminus\{\beta\}$). Let us apply it again to the obtained equivalence, but now with μ, so as to obtain $\pi'[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu[\overline{\alpha_i \mapsto \emptyset}] \equiv \pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu[\overline{\alpha_i \mapsto \emptyset}]$ (mod. $\mathbf{TypeVar}\setminus\{\beta\}$). Given the above, we can finally state that

$$(v, \pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu[\overline{\alpha_i \mapsto \emptyset}]) \in \mathcal{S}[\![\forall\ \beta.\ ((\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta])\mu]\!].$$

Since the $\overline{\alpha_i}$ variables do not occur free in the type scheme in the right hand side, let us apply Lemma 2 to obtain

$$(v, \pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu) \in \mathcal{S}[\![\forall\ \beta.\ ((\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta])\mu]\!] \tag{A.7}$$

Now let us prove that, for every type variable $\gamma \neq \beta$,

$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi\mu)(\gamma)$.

Assume that $\gamma \in rng\,\mu$. Then we get:

$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)])(\mu^{-1}(\gamma))$

If $\mu^{-1}(\gamma) = \alpha$, the latter would be equivalent to $\pi(\alpha) = \pi(\mu^{-1}(\gamma)) = (\pi\mu)(\gamma)$. Otherwise we would get

$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = \pi[\alpha/\beta](\mu^{-1}(\gamma))$

since β is not in the domain of μ, this is equal to $\pi(\mu^{-1}(\gamma)) = (\pi\mu)(\gamma)$. Now let us assume that $\gamma \in rng\,\mu$. We would get

$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)])(\gamma)$

If $\gamma = \alpha$ the right-hand side is equivalent to $\pi(\alpha) = \pi(\gamma) = (\pi\mu)(\gamma)$. Otherwise we get

$(\pi[\alpha/\beta][\alpha \mapsto \pi(\alpha)]\mu)(\gamma) = (\pi[\alpha/\beta])(\gamma)$

Since we are assuming that $\gamma \neq \beta$, the latter is equivalent to $\pi(\gamma) = (\pi\mu)(\gamma)$. Therefore, we have proved that $\pi[\alpha/\beta][\alpha \mapsto (\pi\mu)(\alpha)]\mu \equiv \pi\mu$ (modulo **TypeVar**$\backslash\{\beta\}$). We can thus rewrite A.7 by using Lemma 2 in order to obtain

$(v, \pi\mu) \in \mathcal{S}[\![\forall\ \beta.\ ((\overline{\tau_i[\alpha/\beta]}^n) \to \tau'[\alpha/\beta])\mu]\!]$,

or, equivalently,

$(v, \pi\mu) \in \mathcal{S}[\![(\ \forall\ \alpha.\ (\overline{\tau_i}^n) \to \tau')\mu]\!]$,

which we wanted in order to prove (A.6).Finally, let us assume that $(v, \pi) \in \mathcal{T}[\![\bigsqcup_{i=1}^n \sigma_i]\!]$. This means that v is the union of some f_i ($i \in \{1..n\}$), such that $(f_i, \pi) \in \mathcal{S}[\![\sigma_i]\!]$. For each $i \in \{1..n\}$ we can apply property (A.6) repeatedly (as many times as bound variables in σ_i) in order to obtain $(f_i, \pi\mu) \in \mathcal{S}[\![\sigma_i\mu]\!]$, and hence $(v, \pi\mu) \in \mathcal{T}[\![(\bigsqcup_{i=1}^n \sigma_i)\mu]\!]$.

- **Case** $\pi \vDash \alpha \subseteq \tau'$. We know that $\alpha\mu$ is a type variable. In order to prove that $\pi\mu \vDash \alpha\mu \subseteq \tau'\mu$, let us assume some $v \in (\pi\mu)(\alpha\mu)$. Firstly we prove that $v \in \pi(\alpha)$ by case distinction:
  - If $\alpha \in dom\,\mu$ we get that $(\pi\mu)(\alpha\mu) = (\pi\mu)(\mu(\alpha)) = \pi(\mu^{-1}(\mu(\alpha))) = \pi(\alpha)$. Therefore, $v \in \pi(\alpha)$.
  - If $\alpha \notin dom\,\mu$ we get that $(\pi\mu)(\alpha\mu) = (\pi\mu)(\alpha)$. Since α occurs free in the constraint, it does not belong to $rng\,\alpha$, so $(\pi\mu)(\alpha) = \pi(\alpha)$. Therefore, $v \in \pi(\alpha)$.
    Since $v \in \pi(\alpha)$, by definition of $\pi \vDash \alpha \subseteq \tau'$ we get that $(v, \pi') \in \mathcal{T}[\![\tau']\!]$ for some $\pi' \subseteq \pi$. By induction hypothesis we get that $(v, \pi'\mu) \subseteq \mathcal{T}[\![\tau'\mu]\!]$. Since $\pi' \subseteq \pi$ implies that $\pi'\mu \subseteq \pi\mu$. Hence v belongs to the set $\{v | (v, \pi'') \in \mathcal{T}[\![\tau'\mu]\!],\ \pi'' \subseteq \pi\mu\}$. So $\pi\mu \vDash \alpha\mu \subseteq \tau'\mu$ holds.
- **Case** $\pi \vDash c \subseteq \tau$. It is similar to the previous case.
- **Case** $\pi \vDash \bowtie\{\alpha_1, \ldots, \alpha_n\}$. In the same way as in the case $\pi \vDash \alpha \subseteq C$ we can prove that $(\pi\mu)(\alpha_i\mu) = \pi(\alpha_i)$ for every $i \in \{1..n\}$. Therefore, if $\bigcup_{i=1}^n \pi(\alpha_i) = \emptyset$ we know that $\bigcup_{i=1}^n (\pi\mu)(\alpha_i\mu) = \emptyset$, so $\pi\mu \vDash \bowtie\{\alpha_1\mu, \ldots, \alpha_n\mu\}$. The case in which $\bigcap_{i=1}^n \pi(\alpha_i) \neq \emptyset$ is analogous.
- **Case** $\pi \vDash \alpha \Leftarrow \tau'$. We know that there exists a family of instantiations $\Pi = \{\pi_v | v \in \pi(\alpha)\}$. such that $\pi = \bigcup\Pi$ and $(v, \pi_v) \in \mathcal{T}[\![\tau']\!]$ for every $v \in \pi(\alpha)$. Now let us define the set $\Pi' = \{\pi'_v | v \in (\pi\mu)(\alpha\mu)\}$, where each π'_v is defined as $\pi_v\mu$. It is well defined, since $(\pi\mu)(\alpha\mu) = \pi(\alpha)$ and there is a π_v for every element $v \in \pi(\alpha)$. From $(v, \pi_v) \in \mathcal{T}[\![\tau']\!]$ and the induction hypothesis we get $(v, \pi'_v) \in \mathcal{T}[\![\tau'\mu]\!]$ for each $v \in (\pi\mu)(\alpha\mu)$. Now let us prove that $\pi\mu = \bigcup\Pi'$. Assume some type variable β. If $\beta \in rng\,\mu$ we get:

$$\begin{aligned}(\pi\mu)(\beta) &= \pi(\mu^{-1}(\beta)) \\ &= \bigcup_{v \in \pi(\alpha)} \pi_v(\mu^{-1}(\beta)) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} (\pi_v\mu)(\beta) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} \pi'_v(\beta)\end{aligned}$$

whereas if $\beta \notin rng\,\mu$ we get:

$$\begin{aligned}(\pi\mu)(\beta) &= \pi(\beta) \\ &= \bigcup_{v \in \pi(\alpha)} \pi_v(\beta) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} (\pi_v\mu)(\beta) \\ &= \bigcup_{v \in (\pi\mu)(\alpha\mu)} \pi'_v(\beta)\end{aligned}$$

Therefore we have proved that $\pi\mu \vDash \alpha\mu \Leftarrow \tau'\mu$.

□

Lemma 2. *Let π and π' be two instantiations, τ a type and C a set of constraints.*

1. *For any value v, if $(v, \pi) \in \mathcal{T}[\![\tau]\!]$ and $\pi \equiv \pi'$ (modulo $ftv(\tau)$), then $(v, \pi') \in \mathcal{T}[\![\tau]\!]$.*
2. *If $\pi \vDash C$ and $\pi \equiv \pi'$ (modulo $ftv(C)$) then $\pi' \vDash C$.*

Proof. By induction on the structure of τ and C. When τ is of the form α, if $(v, \pi) \in \mathcal{T}[\![\alpha]\!]$ then $\pi(\alpha) = \{v\}$. Since $\pi \equiv \pi'$ (modulo $ftv(\tau)$), we know that $\pi(\alpha) = \pi'(\alpha)$ for every $\alpha \in ftv(\tau)$, therefore $\pi'(\alpha) = \{v\}$ and $(v, \pi') \in \mathcal{T}[\![\alpha]\!]$. All other cases are straightforward applying the induction hypothesis. □

Lemma 3. *Assume an environment Γ, a pattern p and a type τ such that $\Gamma \vdash p: \tau$. Then for every instantiation π and value v such that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ and $(\theta, v) \in \mathcal{E}[\![p]\!]$ it holds that $(v, \pi) \in \mathcal{T}[\![\tau]\!]$.*

Proof. By induction on the derivation of $\Gamma \vdash p: \tau$. We distinguish cases depending on the last rule applied.

- **Case** [LIT_P]
  In this case we get $p = c$ for some constant c, so it holds that $(c, \pi) \in \mathcal{T}[\![c]\!]$.
- **Case** [VAR_P]
  In this case we get $p = x$ for some variable x such that $\Gamma(x) = \alpha_x$ for some type variable α_x, and $v = \theta(x)$. From $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ it follows that $(\theta(x), \pi) \in \mathcal{T}[\![\Gamma(x)]\!]$ or, equivalently, $(v, \pi) \in \mathcal{T}[\![\alpha_x]\!]$.
- **Case** [LST_P]
  Now p is of the form $[p_1|p_2]$ for some patterns p_1 and p_2 and v is of the form $([\_|\_], v_1, v_2)$ for some v_1 and v_2 such that $(\theta, v_1) \in \mathcal{E}[\![p_1]\!]$ and $(\theta, v_2) \in \mathcal{E}[\![p_2]\!]$. By applying induction hypothesis to the derivations of both patterns we get $(v_1, \pi) \in \mathcal{T}[\![\tau_1]\!]$, $(v_2, \pi) \in \mathcal{T}[\![\tau_2]\!]$ which leads to $(v, \pi) \in \mathcal{T}[\![\text{nelist}(\tau_1, \tau_2)]\!]$.
- **Case** [TPL_P]
  In this case p is of the form $\{\overline{p_i}^n\}$ for some patterns $p_1, ..., p_n$ and v is of the form $(\{\cdots\}^n, v_1, ..., v_n)$ for some $v_1, ..., v_n$. We apply the induction hypothesis to each subderivation $\Gamma \vdash p_i: \tau_i$ so as to get $(v_i, \pi) \in \mathcal{T}[\![\tau_i]\!]$ for every $i \in \{1..n\}$, hence $(v, \pi) \in \mathcal{T}[\![\{\overline{\tau_i}^n\}]\!]$.

□

Lemma 8. *Assume that* $(\theta, v) \in \mathcal{T}[\![\rho]\!]$. *For every variable* x *and value* v' *it holds that* $(\theta[x/v'], v) \in \mathcal{T}[\![\rho \setminus \{x\}]\!]$.

Proof. When $(\theta, v) \in \mathcal{T}[\![\rho_1; \rho_2]\!]$, we know that $(\theta, v) \in \mathcal{T}[\![\rho_1]\!]$ or $(\theta, v) \in \mathcal{T}[\![\rho_2]\!]$. Then by induction hypothesis we get $(\theta[x/v'], v) \in \mathcal{T}[\![\rho_1 \setminus \{x\}]\!]$ or $(\theta[x/v'], v) \in \mathcal{T}[\![\rho_2 \setminus \{x\}]\!]$, which becomes $(\theta[x/v'], v) \in \mathcal{T}[\![(\rho_1; \rho_2) \setminus \{x\}]\!]$.

When $(\theta, v) \in \mathcal{T}[\![\langle \tau; \Gamma \rangle]\!]$, we know that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$ and $(v, \pi) \in \mathcal{T}[\![\tau]\!]$. From $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$, we know that $\forall\, z \in \mathbf{Var}.\ (\theta(z), \pi) \in \mathcal{T}[\![\Gamma(z)]\!]$ and $\pi \vDash \Gamma|_C$, then we get:

$$
\begin{array}{ll}
 & \forall\, z \in \mathbf{Var}.\ (\theta(z), \pi) \in \mathcal{T}[\![\Gamma(z)]\!] \\
\Leftrightarrow & \forall\, z \in \mathbf{Var} \setminus \{x\}.\ (\theta(z), \pi) \in \mathcal{T}[\![\Gamma(z)]\!] \vee \forall\, z \in \{x\}.\ (\theta(z), \pi) \in \mathcal{T}[\![\Gamma(z)]\!] \\
 & \qquad \text{since } \Gamma(x) = \text{any}() \\
\Rightarrow & \forall\, z \in \mathbf{Var} \setminus \{x\}.\ (\theta(z), \pi) \in \mathcal{T}[\![\Gamma(z)]\!] \vee \forall\, z \in \{x\}.\ (v', \pi) \in \mathcal{T}[\![\text{any}()]\!] \\
\Leftrightarrow & \forall\, z \in \mathbf{Var}.\ (\theta[x/v'](z), \pi) \in \mathcal{T}[\![(\Gamma \setminus \{x\})(z)]\!]
\end{array}
$$

Since $\Gamma|_C = \Gamma \setminus \{x\}|_C$, with $\forall\, z \in \mathbf{Var}.\ (\theta[x/v'](z), \pi) \in \mathcal{T}[\![(\Gamma \setminus \{x\})(z)]\!]$ and $\pi \vDash \Gamma \setminus \{x\}|_C$, we get $\theta[x/v'] \in \mathcal{T}^{\pi}_{Env}[\![\Gamma \setminus \{x\}]\!]$; and with $(v, \pi) \in \mathcal{T}[\![\tau]\!]$, we get $(\theta[x/v'], v) \in \mathcal{T}[\![\langle \tau; \Gamma \rangle \setminus \{x\}]\!]$. □

Lemma 9. *For every* π, α_1 *and* α_2*:*

1. $\pi \vDash \alpha_1 \subseteq \alpha_2$ *if and only if* $\pi(\alpha_1) \subseteq \pi(\alpha_2)$.
2. $\pi \vDash \alpha_1 = \alpha_2$ *if and only if* $\pi(\alpha_1) = \pi(\alpha_2)$.

Proof. Let us prove (1): ($\Rightarrow$) Assume that $\pi \vDash \alpha_1 \subseteq \alpha_2$. Given some $v \in \pi(\alpha_1)$ it holds that $(v, \pi') \in \mathcal{T}[\![\alpha_2]\!]$ for some $\pi' \subseteq \pi$. This means that $\pi'(\alpha_2) = \{v\}$ and hence $v \in \pi(\alpha_2)$. Therefore $\pi(\alpha_1) \subseteq \pi(\alpha_2)$. ($\Leftarrow$) Let us prove that $\pi \vDash \alpha_1 \subseteq \alpha_2$. For any $v \in \pi(\alpha_1)$ it holds that $v \in \pi(\alpha_2)$. Therefore $\pi[\alpha_2 \mapsto \{v\}] \subseteq \pi$ and $(v, \pi[\alpha_2 \mapsto \{v\}]) \in \mathcal{T}[\![\alpha_2]\!]$.

Property (2) can be proved by applying (1) twice. □

Proposition 1. *For every substitution* θ, *values* $v_1, ..., v_n$, *and annotated types* $\rho_1, ..., \rho_n$ *such that* $(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]$ *for each* $i \in \{1..n\}$ *we get* $(\theta, (\{\overset{n}{...}\}, v_1, ..., v_n)) \in \mathcal{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!]$.

Proof. When ρ_i has the form $\langle \tau_i; \Gamma_i \rangle$ for each $i \in \{1..n\}$, by definition we know:

$$
\begin{array}{l}
(\theta, (\{\overset{n}{...}\}, \overline{v_i}^n)) \in \mathcal{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!] \ \wedge\ \forall\, i \in \{1..n\}.\ (\theta, v_i) \in \mathcal{T}[\![\rho_i]\!] \\
\qquad\qquad \Updownarrow \\
(\theta, (\{\overset{n}{...}\}, \overline{v_i}^n)) \in \mathcal{T}[\![\langle \tau_1; \Gamma_1 \rangle \otimes \cdots \otimes \langle \tau_n; \Gamma_n \rangle]\!] \ \wedge\ \forall\, i \in \{1..n\}.\ (\theta, v_i) \in \mathcal{T}[\![\langle \tau_i; \Gamma_i \rangle]\!]
\end{array}
$$

From the semantics of $\langle \tau_i; \Gamma_i \rangle$ we have:

$$
\begin{array}{l}
\theta \in \mathcal{T}^{\pi_i}_{Env}[\![\Gamma_i]\!] \ \wedge\ (v_i, \pi_i) \in \mathcal{T}[\![\tau_i]\!] \\
\qquad \Updownarrow \\
\forall\, x \in \mathbf{Var}.\ (\theta(x), \pi_i) \in \mathcal{T}[\![\Gamma_i(x)]\!] \ \wedge\ \pi_i \vDash \Gamma_i|_C \ \wedge\ (v_i, \pi_i) \in \mathcal{T}[\![\tau_i]\!]
\end{array}
$$

First we want to normalize each Γ_i into a $\Gamma_i' = [\overline{x_j: \alpha_j}|\Gamma_i|_C \cup \{\overline{\Gamma_i(x_j) \Leftarrow \alpha_j}\}]$, such that $ftv(\langle \tau_i; \Gamma_i \rangle) \cap \{\overline{\alpha_j}\} = \emptyset$ for every $i \in \{1..n\}$, this means we need a new $\pi_i' = \pi_i[\overline{\alpha_j \mapsto \{\theta(x_j)\}}]$, and since $\pi_i' \equiv \pi_i$ (modulo $ftv(.\Gamma_i|_C)$) with Lemma 2 we obtain $\pi_i' \vDash \Gamma_i|_C$. Because $\pi_i'(\alpha_j) = \{\theta(x_j)\}$, we know that:

$$
\begin{array}{lll}
(\theta(x_j), \pi_i') \in \mathcal{T}[\![\alpha_j]\!] & \Longleftrightarrow & (\theta(x_j), \pi_i') \in \mathcal{T}[\![\Gamma_i'(x_j)]\!] \\
 & \Longrightarrow & \forall\, x \in \mathbf{Var}.\ (\theta(x), \pi_i') \in \mathcal{T}[\![\Gamma_i'(x)]\!]
\end{array}
$$

We also have that $\pi_i' \vDash \{\overline{\Gamma_i(x_j) \Leftarrow \alpha_j}\}$, since $v \in \pi_i'(\alpha_j)$ means that $v = \theta(x_j)$ then $\pi_v = \pi_i'$, and $(v, \pi_i) \in \mathcal{T}[\![\Gamma_i(x_j)]\!]$ can be changed into $(v, \pi_i') \in \mathcal{T}[\![\Gamma_i(x_j)]\!]$, using Lemma 2, since $\pi_i' \equiv \pi_i$ (modulo $ftv(\Gamma_i(x_j))$). Then with $\pi_i' \vDash \Gamma_i|_C$ and $\pi_i' \vDash \{\overline{\Gamma_i(x_j) \Leftarrow \alpha_j}\}$, we have that $\pi_i' \vDash \Gamma_i'|_C$, and we can use again Lemma 2 to obtain $(v_i, \pi_i') \in \mathcal{T}[\![\tau_i]\!]$, hence:

$$\forall\, x \in \mathbf{Var}.\ (\theta(x), \pi_i') \in \mathcal{T}[\![\Gamma_i'(x)]\!] \ \wedge\ \pi_i' \vDash \Gamma_i'|_C \ \wedge\ (v_i, \pi_i') \in \mathcal{T}[\![\tau_i]\!]$$

for all $i \in \{1..n\}$. Now we need to use a single π' defined as:

$$\pi'(\alpha) = \begin{cases} \{\theta(x_j)\} & \alpha \in \{\overline{\alpha_j}\} \\ \pi'_i(\alpha) & \alpha \in ftv(\langle \tau_i; \Gamma_i \rangle) \text{ for all } i \in \{1..n\} \\ \varnothing & \text{otherwise} \end{cases}$$

Since $ftv(\langle \tau_i; \Gamma_i \rangle) \cap \{\overline{\alpha_j}\} = \varnothing$ and $ftv(\langle \tau_i; \Gamma_i \rangle) \cap ftv(\langle \tau_k; \Gamma_k \rangle) = \varnothing$ for all $i, k \in \{1.n\}$ and $i \neq k$, and, we know that π' is well formed. Then, using Lemma 2:

$$\pi' \equiv \pi'_i(\text{modulo } ftv(\langle \tau_i; \Gamma'_i \rangle)) \implies \begin{cases} \forall\, x \in \mathbf{Var}.\ (\theta(x), \pi') \in \mathcal{T}[\![\Gamma'_i(x)]\!] \\ \pi' \vDash \Gamma'_i|_C \\ (v_i, \pi') \in \mathcal{T}[\![\tau_i]\!] \end{cases}$$

With these results, we obtain by definition:

$$\forall\, i \in \{1..n\}.\ (v_i, \pi') \in \mathcal{T}[\![\tau_i]\!] \iff ((\{\overset{n}{\cdot\cdot}\}, \overline{v_i}^n), \pi') \in \mathcal{T}[\![\{\overline{\tau_i}^n\}]\!] \tag{A.8}$$

After obtaining the semantics of the tuple, we need to unify the environments Γ' such that $\Gamma' = [\overline{x_j{:}\ \alpha_j}|\Gamma'_1|_C \cup \cdots \cup \Gamma'_n|_C]$. Because $\Gamma'(x_j) = \Gamma'_i(x_j)$ for every $i \in \{1.n\}$:

$$\forall\, x \in \mathbf{Var}.\ (\theta(x), \pi') \in \mathcal{T}[\![\Gamma'_i(x)]\!] \iff \forall\, x \in \mathbf{Var}.\ (\theta(x), \pi') \in \mathcal{T}[\![\Gamma'(x)]\!]$$

which leads us to:

$$\begin{array}{l} \forall\, x \in \mathbf{Var}.\ (\theta(x), \pi') \in \mathcal{T}[\![\Gamma'(x)]\!] \ \wedge\ \pi' \vDash \Gamma'_1|_C \ \wedge\ \cdots\ \wedge\ \pi' \vDash \Gamma'_n|_C \\ \qquad\qquad \Updownarrow \\ \qquad\qquad \theta \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma']\!] \end{array} \tag{A.9}$$

Finally with A.8 and A.9 we obtain:

$$(\theta, (\{\overset{n}{\cdot\cdot}\}, \overline{v_i}^n)) \in \mathcal{T}[\![\langle \{\overline{\tau_i}^n\}; \Gamma' \rangle]\!] = \mathcal{T}[\![\langle \tau_1; \Gamma_1 \rangle \otimes \ \cdots \otimes \langle \tau_n; \Gamma_n \rangle]\!]$$

proving the current case.

When ρ_j has the form $\rho'_j; \rho''_j$ for some $j \in \{1.n\}$, by definition we know:

$$\rho_1 \otimes \ \cdots \otimes (\rho'_j; \rho''_j) \otimes \ \cdots \otimes \rho_n = \rho_1 \otimes \ \cdots \otimes \rho'_j \otimes \ \cdots \otimes \rho_n; \rho_1 \otimes \ \cdots \otimes \rho''_j \otimes \ \cdots \otimes \rho_n$$

and we also know by definition:

$$\begin{array}{lll} (\theta, v_j) \in \mathcal{T}[\![\rho_j]\!] & \iff & (\theta, v_j) \in \mathcal{T}[\![\rho'_j; \rho''_j]\!] \\ & \iff & (\theta, v_j) \in \mathcal{T}[\![\rho'_j]\!] \cup \mathcal{T}[\![\rho''_j]\!] \end{array}$$

hence we have two subcases:

- When $(\theta, v_j) \in \mathcal{T}[\![\rho'_j]\!]$ by induction hypothesis with $(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]$ for each $i \in \{1.n\}$ when $i \neq j$, we obtain:
$$\begin{array}{lll} (\theta, (\{\overset{n}{\cdot\cdot}\}, \overline{v_i}^n)) \in & \mathcal{T}[\![\rho_1 \otimes \ \cdots \otimes \rho'_j \otimes \ \cdots \otimes \rho_n]\!] \\ \subseteq & \mathcal{T}[\![\rho_1 \otimes \ \cdots \otimes \rho'_j \otimes \ \cdots \otimes \rho_n; \rho_1 \otimes \ \cdots \otimes \rho''_j \otimes \ \cdots \otimes \rho_n]\!] \end{array}$$

- When $(\theta, v_j) \in \mathcal{T}[\![\rho''_j]\!]$ by induction hypothesis with $(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]$ for each $i \in \{1.n\}$ when $i \neq j$, we obtain:
$$\begin{array}{lll} (\theta, (\{\overset{n}{\cdot\cdot}\}, \overline{v_i}^n)) \in & \mathcal{T}[\![\rho_1 \otimes \ \cdots \otimes \rho''_j \otimes \ \cdots \otimes \rho_n]\!] \\ \subseteq & \mathcal{T}[\![\rho_1 \otimes \ \cdots \otimes \rho'_j \otimes \ \cdots \otimes \rho_n; \rho_1 \otimes \ \cdots \otimes \rho''_j \otimes \ \cdots \otimes \rho_n]\!] \end{array}$$

proving the current case and the proposition. □

Theorem 1. *Assume an environment* Γ, *an expression e and a sequence* ρ *of pairs. If* $\Gamma \vdash e{:}\ \rho$ *then*

$$\mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\rho]\!]$$

Proof. By induction on size of the typing derivation. We distinguish cases depending on the last rule applied.

- **Case [SUB1]**
We assume that we have obtained a derivation of $\Gamma' \vdash e{:}\ \rho$ for some $\Gamma \subseteq \Gamma'$. Then we get:
$$\begin{array}{lll} & \mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} & \\ \subseteq & \mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma']\!]} & \text{since} \mathcal{T}_{Env}[\![\Gamma]\!] \subseteq \mathcal{T}_{itEnv}[\![\Gamma']\!] \\ \subseteq & \mathcal{T}[\![\rho]\!] & \text{by i.h.} \end{array}$$

- **Case [SUB2]**
In this case we assume a derivation of $\Gamma \vdash e{:}\ \rho'$ for some $\rho \subseteq \rho'$. Then we get:

$\mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$
$\subseteq$ $\mathcal{T}[\![\rho]\!]$ by i.h.
$\subseteq$ $\mathcal{T}[\![\rho']\!]$ since$\mathcal{T}[\![\rho]\!] \subseteq \mathcal{T}[\![\rho']\!]$

- **Case [CNS]**
In this case the expression e being typed is a constant literal c. Assume that $(\theta, v) \in \mathcal{E}[\![c]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. This means that $v = c$ and that $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. From the latter it follows the existence of an instantiation π such that $\pi \vDash \Gamma|_C$ and $(\theta(x), \pi) \in \mathcal{T}[\![\Gamma(x)]\!]$ for all $x \in \mathbf{Var}$. Besides this, we get that $(c, \pi) \in \mathcal{T}[\![c]\!]$, so we finally obtain $(\theta, c) \in \mathcal{T}[\![\langle c; \Gamma\rangle]\!]$.
- **Case [VAR]**
We know that the expression e is a variable $x \in \mathbf{Var}$. Assume a pair $(\theta, v) \in \mathcal{E}[\![x]\!]$ such that $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. In this case $v = \theta(x)$. We know that there exists an instantiation π such that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$. In particular, $(\theta(x), \pi) \in \mathcal{T}_{Env}[\![\Gamma(x)]\!]$. Therefore, $(\theta, v) = (\theta, \theta(x)) \in \mathcal{T}[\![\langle \Gamma(x); \Gamma\rangle]\!]$, which proves the lemma.
- **Case [TPL]**
Assume that $(\theta, v) \in \mathcal{E}[\![\{\overline{e_i}^n\}]\!]$ with $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. In this case v is a value of the form $(\{\overset{n}{\cdots}\}, v_1, ..., v_n)$, where $v_i \in \mathcal{E}[\![e_i]\!]$ for each $i \in \{1.n\}$. By induction hypothesis, since $(\theta, v_i) \in \mathcal{E}[\![e_i]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$, it holds that $(\theta, v_i) \in \mathcal{T}[\![\rho_i]\!]$ for each i. By Lemma 1 we get that $(\theta, v) \in \mathcal{T}[\![\rho_1 \otimes \cdots \otimes \rho_n]\!]$.
- **Case [LST]**
Assume that $(\theta, v) \in \mathcal{E}[\![[e_1|e_2]]\!]$, so $v = ([\_|\_], v_1, v_2)$ for some v_1, v_2. Similarly as in the previous case, we can apply induction hypothesis and Lemma 1 to get that $(\theta, (\{\overset{2}{\cdots}\}, v_1, v_2)) \in \mathcal{T}[\![\rho_1 \otimes \rho_2]\!] = \mathcal{T}[\![\overline{\langle\{\tau_i, \tau_i'\}; \Gamma_i\rangle}^n]\!]$, so $(\theta, (\{\overset{2}{\cdots}\}, v_1, v_2))$ belongs to $\mathcal{T}[\![\langle\{\tau_k, \tau_k'\}; \Gamma_k\rangle]\!]$ for some $k \in \{1.n\}$. There exists an instantiation π such that $(\theta(x), \pi) \in \mathcal{T}[\![\Gamma_k(x)]\!]$, $\pi \vDash \Gamma_k|_C$, and $((\{\overset{2}{\cdots}\}, v_1, v_2), \pi) \in \mathcal{T}[\![\langle\{\tau_k, \tau_k'\}; \Gamma_k\rangle]\!]$, the latter of which implies $(v_1, \pi) \in \mathcal{T}[\![\tau_k]\!]$ and $(v_2, \pi) \in \mathcal{T}[\![\tau_k']\!]$. Now we can apply the semantic definition of $\text{nelist}(\tau_k, \tau_k')$ in order to get $(([\_|\_], v_1, v_2), \pi) \in \mathcal{T}[\![\text{nelist}(\tau_k, \tau_k')]\!]$. Therefore, we get:

$(\theta, v) \in \mathcal{T}[\![\langle \text{nelist}(\tau_k, \tau_k'); \Gamma_k\rangle]\!]$

and hence

$(\theta, v) \in \mathcal{T}[\![\overline{\langle \text{nelist}(\tau_k, \tau_k'); \Gamma_k\rangle}^n]\!]$.

- **Case [ABS]**
In this case $e = \mathbf{fun}(\overline{x_i}^n) \to e'$ for some variables $\overline{x_i}^n$ and expression e'. Assume a pair $(\theta, v) \in \mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. From the semantic definition of $\mathcal{E}[\![\mathbf{fun}(\overline{x_i}^n) \to e']\!]$ it follows that v is the graph of an n-ary function. Moreover, the fact that $\theta \in \mathcal{T}[\![\Gamma]\!]$ implies the existence of an instantiation π such that:

$$\forall\, z \in \mathbf{Var}.\ (\theta(z), \pi) \in \mathcal{T}[\![\Gamma(z)]\!] \qquad \pi \vDash \Gamma|_C \tag{A.10}$$

Assume a tuple $w \in v$. Then v is of the form $((\overline{v_i}^n), v')$ for some values $\overline{v_i}^n$, v' such that $(\theta[\overline{x_i/v_i}^n], v') \in \mathcal{E}[\![e']\!]$. Given that $\Gamma(x_i) = \text{any}()$ for every $i \in \{1.n\}$, we get, in fact, that $(\theta[\overline{x_i/v_i}^n], v') \in \mathcal{T}[\![e']\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Therefore, the tuple $(\theta[\overline{x_i/v_i}^n], v')$ also belongs to $\mathcal{T}[\![\overline{\langle \tau_j; \Gamma[\overline{x_i: \tau_{j,i}}^n]\rangle}^m]\!]$. In particular, it belongs to $\mathcal{T}[\![\langle \tau_k; \Gamma[\overline{x_i: \tau_{k,i}}^n]\rangle]\!]$ for some $k \in \{1.m\}$. As a result, we have proven the following

$$\begin{aligned} &\text{For every } w = ((\overline{v_i}^n), v') \in v \text{ there exists some } k_w \in \{1.\ m\} \\ &\qquad\qquad \text{s.t. } (\theta[\overline{x_i/v_i}^n], v') \in \mathcal{T}[\![\langle \tau_k; \Gamma[\overline{x_i: \tau_{k,i}}^n]\rangle]\!] \end{aligned} \tag{A.11}$$

We use k_w to highlight its dependence of the tuple w. For every $j \in \{1.m\}$, let us define the set W_j as follows:

$W_j = \{w \in v | k_w = j\}$

From (A.11) it follows that $v = W_1 \cup ... \cup W_m$. Now let us assume a given $j \in \{1.m\}$. If W_j is empty, let us define an instantiation π' as follows:

$$\forall\, \alpha \in \mathbf{TypeVar}:\ \pi'(\alpha) = \begin{cases} \varnothing & \text{if } \alpha \in itv((\overline{\tau_{j,i}}^n) \to \tau_j) \\ \pi(\alpha) & \text{otherwise} \end{cases}$$

Then, according to the semantic definition of a functional type, $(\varnothing, \pi') \in \mathcal{F}[\![(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$. By the way in which π' is defined, it holds that $\pi \equiv \pi'$ (modulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{j,i}}^n) \to \tau_j)$), so $(\varnothing, \pi) \in \mathcal{S}[\![\overline{\forall}\ (\overline{\tau_{j,i}}^n) \to \tau_j]\!]$. Equivalently, $(W_j, \pi) \in \mathcal{S}[\![\overline{\forall}\ (\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ since we are assuming that W_j is empty.
Now let us assume that W_j is nonempty. Assume some tuple $w \in W_j$ of the form $((\overline{v_i}^n), v')$. By (A.11) it holds that $(\theta[\overline{x_i/v_i}^n], v') \in \mathcal{T}[\![\langle \tau_j; \Gamma[\overline{x_i: \tau_{j,i}}^n]\rangle]\!]$. This implies the existence of some instantiation π_w such that:

$$\left.\begin{array}{ll} (v_i, \pi_w) \in \mathcal{T}[\![\tau_{j,i}]\!] & \text{for every } i \in \{1.\ n\} \\ (\theta(z), \pi_w) \in \mathcal{T}[\![\Gamma(z)]\!] & \text{for any other } z \notin \{\overline{x_i}^n\} \\ (v', \pi_w) \in \mathcal{T}[\![\tau_j]\!] & \\ \pi_w \vDash \Gamma|_C & \end{array}\right\} \tag{A.12}$$

Again, we write π_w to highlight the fact that each tuple $w \in W_j$ has its own π_w that witnesses the fact that w belongs to $\mathcal{T}[\![\langle \tau_j; \Gamma[\overline{x_i: \tau_{j,i}}^n]\rangle]\!]$. For each one of such π_w, let us define π_w' as follows:

$$\forall\ \alpha \in \mathbf{TypeVar}\colon \pi'_w(\alpha) = \begin{cases} \pi_w(\alpha) & \text{if } \alpha \in itv((\overline{\tau_{j,i}}^n) \to \tau_j) \\ \pi(\alpha) & \text{otherwise} \end{cases}$$

where π is the instantiation that satisfies (A.10). Now let us prove that $\pi_w \equiv \pi'_w$ (modulo $ftv((\overline{\tau_{j,i}}^n) \to \tau_j)$). Assume a type variable α.

- If $\alpha \in itv((\overline{\tau_{j,i}}^n) \to \tau_j)$, then $\pi_w(\alpha) = \pi'_w(\alpha)$ by definition.
- If $\alpha \in ftv((\overline{\tau_{j,i}}^n) \to \tau_j)$ but $\alpha \notin itv((\overline{\tau_{j,i}}^n) \to \tau_j)$, then $\alpha \in ftv(\overline{\forall}\ (\overline{\tau_{j,i}}^n) \to \tau_j)$. The [ABS] rule specifies that there exists some y such that $\Gamma(y) = \alpha$ and $y \notin \{\overline{x_i}^n\}$, that is, α is one of the $\overline{\alpha_i}$ type variables that have been used when applying [ABS]. Therefore, we get $(\theta(y), \pi_w) \in \mathcal{T}[\![\Gamma(y)]\!] = \mathcal{T}[\![\alpha]\!]$, which implies $\pi_w(\alpha) = \{\theta(y)\}$. On the other hand, from (A.10) we get $\pi(\alpha) = \{\theta(y)\}$. Therefore, $\pi'_w(\alpha) = \pi(\alpha) = \{\theta(y)\} = \pi_w(\alpha)$.
  Since $\pi_w \equiv \pi'_w$ (modulo $ftv((\overline{\tau_{j,i}}^n) \to \tau_j)$) we can apply Lemma 2 to (A.12) in order to obtain:

$$(v_i, \pi'_w) \in \mathcal{T}[\![\tau_{j,i}]\!] \quad \text{for every } i \in \{1.\ n\}$$
$$(v', \pi'_w) \in \mathcal{T}[\![\tau_j]\!]$$

Now let us define $\Pi_j = \{\pi'_w | w \in W_j\}$, and $\pi_j = \bigcup \Pi_j$. Obviously, by definition of π'_w, it holds that $\pi_j \equiv \pi$ (modulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{j,i}}^n) \to \tau_j)$). Therefore, $\pi_j \in Dcp(\Pi_j, (\overline{\tau_{j,i}}^n) \to \tau_j)$, which entails $(W_j, \pi_j) \in \mathcal{F}[\![(\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ for each $j \in \{1.m\}$. Moreover, we get that $(W_j, \pi) \in \mathcal{S}[\![\overline{\forall}\ (\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ for each $j \in \{1.m\}$ (again as a consequence of being π_j equal to π modulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{j,i}}^n) \to \tau_j)$). Therefore $(v, \pi) = (W_1 \cup \ldots \cup W_m, \pi) \in \mathcal{T}[\![\bigsqcup_{j=1}^m \overline{\forall}\ (\overline{\tau_{j,i}}^n) \to \tau_j]\!]$. Together with (A.10): $(\theta, v) \in \mathcal{T}[\![\langle \bigsqcup_{j=1}^m \overline{\forall}\ (\overline{\tau_{j,i}}^n) \to \tau_j;\ \Gamma \rangle]\!]$.

- **Case [APP1]**
  In this case $e = f(\overline{x_i}^n)$ for some function symbol f and variables $\overline{x_i}^n$. Given that $(\theta, v) \in \mathcal{E}[\![f(\overline{x_i}^n)]\!]$, we get that $\theta(f)$ is the graph of an n-ary function containing the tuple $((\overline{\theta(x_i)}^n), v)$. Since $\theta(f)$ is an n-ary function it holds that $\theta \in \mathcal{T}_{Env}[\![f\colon (\overline{\mathrm{any}()}^n) \to \mathrm{any}()]\!]$, and since $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$, we get that $\theta \in \mathcal{T}_{Env}[\![\Gamma_0]\!]$, where Γ_0 is the environment occurring in the [APP1] rule. This rule allows us to assume that the f function has the following overloaded scheme in Γ_0:

$$\Gamma_0(f) = \bigsqcup_{j=1}^{m} \forall\ \overline{\alpha_{j,i}}.\ (\overline{\tau_{j,i}}^n) \to \tau_j$$

This means that there exists some instantiation π such that $(\theta(f), \pi) \in \mathcal{T}[\![\bigsqcup_{j=1}^m \forall\ \overline{\alpha_{j,i}}.\ (\overline{\tau_{j,i}}^n) \to \tau_j]\!]$ and $\pi(\alpha_{x_i}) = \{\theta(x_i)\}$ for every parameter x_i where $i \in \{1.n\}$. According to the semantic definition of an overloaded type scheme, it holds that $\theta(f) = f_1 \cup \ldots \cup f_m$, where each f_i is the subgraph corresponding to each branch of the overloaded type. That is, for each $j \in \{1.m\}$ we get $(f_j, \pi) \in \mathcal{S}[\![\forall\ \overline{\alpha_{j,i}}.\ (\overline{\tau_{j,i}}^n) \to \tau_j]\!]$. Since we know that $((\overline{\theta(x_i)}^n), v)$ is within $\theta(f)$, it has to belong to some f_k where $k \in \{1.m\}$. From the definition of $\mathcal{S}[\![\_]\!]$ we get that $(f_k, \pi') \in \mathcal{F}[\![(\overline{\tau_{k,i}}^n) \to \tau_k]\!]$ for some $\pi' \equiv \pi$ (modulo $\mathbf{TypeVar} \setminus \{\overline{\alpha_{k,i}}\}$). Moreover, we get:

$$\forall\ i \in \{1.\ n\}.\ (\theta(x_i), \pi'') \in \mathcal{T}[\![\tau_{k,i}]\!] \quad \text{and} \quad (v, \pi'') \in \mathcal{T}[\![\tau_k]\!] \tag{A.13}$$

for some $\pi'' \subseteq \pi'$ such that $\pi'' \equiv \pi'$ (modulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{k,i}}^n) \to \tau_k)$). Let us denote by $\{\overline{\beta_i}\}$ the set of variables in $itv((\overline{\tau_{k,i}}^n) \to \tau_k) \cup \{\overline{\alpha_{k,i}}\}$ and let us assume another set $\{\overline{\beta'_i}\}$ of fresh variables such that $\mu_k = [\overline{\beta_i}/\overline{\beta'_i}]$ according to the [APP1] rule. Since the $\overline{\beta'_i}$ variables are fresh, we can apply Lemma 1 to (A.13) so as to get:

$$\forall\ i \in \{1.\ n\}.\ (\theta(x_i), \pi''\mu_k) \in \mathcal{T}[\![\tau_{k,i}\mu_k]\!] \quad \text{and} \quad (v, \pi''\mu_k) \in \mathcal{T}[\![\tau_k\mu_k]\!] \tag{A.14}$$

Notice that none of the $\overline{\beta_i}$ occur either in $\tau_{k,i}\mu_k$ or $\tau_k\mu_k$, since all these variables have been renamed by the application of μ_k. This allows us to define an instantiation $\pi_\circ = (\pi''\mu_k)[\overline{\beta_i \mapsto \pi(\beta_i)}]$ so that $\pi_\circ \equiv \pi''\mu_k$ (modulo $ftv(\tau_{k,i}\mu_k)$) for each $i \in \{1.n\}$ and $\pi_\circ \equiv \pi''\mu_k$ (modulo $ftv(\tau_k\mu_k)$). In this way we can use Lemma 2 to rewrite (A.14) as follows:

$$\forall\ i \in \{1.\ n\}.\ (\theta(x_i), \pi_\circ) \in \mathcal{T}[\![\tau_{k,i}\mu_k]\!] \quad \text{and} \quad (v, \pi_\circ) \in \mathcal{T}[\![\tau_k\mu_k]\!] \tag{A.15}$$

Now let us denote by $\pi_\bullet$ the instantiation $\pi[\overline{\beta'_i \mapsto \pi''(\beta_i)}]$ and prove that $\pi_\circ = \pi_\bullet$. Assume some $\alpha \in \mathbf{TypeVar}$:

- If $\alpha = \beta'_i$ for some i we get $\pi_\circ(\beta'_i) = (\pi''\mu_k)(\beta'_i) = \pi''(\beta_i) = \pi_\bullet(\beta'_i)$.
- If $\alpha = \beta_i$ for some i we get $\pi_\circ(\beta_i) = \pi(\beta_i) = \pi_\bullet(\beta_i)$.
- Otherwise α does not belong to $itv((\overline{\tau_{k,i}}^n) \to \tau_k) \cup \{\overline{\alpha_{k,i}}\}$. This means that:

  $$\pi_\circ(\alpha) = (\pi''\mu_k)(\alpha) = \pi''(\alpha) = \pi'(\alpha) = \pi(\alpha) = \pi_\bullet(\alpha)$$

  since $\pi'' \equiv \pi'$ (modulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{k,i}}^n) \to \tau_k)$) and $\pi'' \equiv \pi'$ (modulo $\mathbf{TypeVar} \setminus itv((\overline{\tau_{k,i}}^n) \to \tau_k)$).
  As a consequence, we can substitute $\pi_\bullet$ for $\pi_\circ$ in (A.15) and get:

$$\forall\ i \in \{1.\ n\}.\ (\theta(x_i), \pi_\bullet) \in \mathcal{T}[\![\tau_{k,i}\mu_k]\!] \quad \text{and} \quad (v, \pi_\bullet) \in \mathcal{T}[\![\tau_k\mu_k]\!] \tag{A.16}$$

Since the $\overline{\beta_i}'$ variables are different from the $\overline{\alpha_{x_i}}^n$, we know that $\pi_\bullet(\alpha_{x_i}) = \pi(\alpha_{x_i}) = \{\theta(x_i)\}$ and hence from (A.16) we know that $\pi_\bullet \vDash \tau_{k,i}\mu_k \Leftarrow \alpha_{x_i}$ for each $i \in \{1.n\}$. Moreover, since $\pi \equiv \pi_\bullet$ (modulo $\mathbf{TypeVar} \setminus \{\beta'_i\}$) and the β'_i variables do not occur in Γ_0, we get by Lemma 2:

$$\forall\ y.\ \ (\theta(y), \pi_\bullet) \in \mathcal{T}[\![\Gamma_0(y)]\!] \quad \text{and} \quad \pi_\bullet \vDash \Gamma_0|_C \tag{A.17}$$

Now let us prove that $\pi_\bullet \vDash \beta\mu_k \subseteq \beta$ for every $\beta \in itv(\forall\ \overline{\alpha_{k,i}}.\ (\overline{\tau_{k,i}}^n) \to \tau_k)$. This means that β is some of the β_i defined before, but none of the $\{\overline{\alpha_{k,i}}\}$. Hence we have to prove that $\pi_\bullet \vDash \beta'_i \subseteq \beta_i$. Assume that $v \in \pi_\bullet(\beta'_i)$. We get that:

$$v \in \pi_\bullet(\beta'_i) = \pi''(\beta_i) \subseteq \pi'(\beta_i) = \pi(\beta_i) = \pi_\bullet(\beta_i)$$

so $(v, [\beta_i \mapsto \{v\}]) \in \mathcal{T}[\![\beta_i]\!]$ and $[\beta_i \mapsto \{v\}] \subseteq \pi_.$. Therefore, $\pi_. \vDash \beta_i \mu_k \subseteq \beta_i$ which we wanted to prove. Given this result and equations (A.16) and (A.17) we get:

$$\begin{array}{ll}(\theta, v) \in & \mathcal{T}[\![\langle \tau_k \mu_k;\, \Gamma_k \rangle]\!] \subseteq \mathcal{T}[\![\overline{\langle \tau_j \mu_j;\, \Gamma_j \rangle}^{m}]\!] \\ \text{where} & \Gamma_k = \Gamma_0[\{\overline{\tau_{k,i} \mu_k \Leftarrow \alpha_{x_i}}^{n}\} \cup \{\beta \mu_k \subseteq \beta \mid \beta \in itv(\forall\, \overline{\alpha_{k,i}}.\, (\overline{\tau_{k,i}}^{n}) \to \tau_k)\}]\end{array}$$

which proves the Lemma.

- **Case [APP2]**
  In this case we have to prove that $\mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} \subseteq \mathcal{T}[\![\langle \text{none}();\, \bot \rangle]\!]$ or, equivalently, $\mathcal{E}[\![e]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]} = \varnothing$. Let us prove it by contradiction. Assume that $(\theta, v) \in \mathcal{E}[\![e]\!]$ and $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. As in the case [APP1], this implies that $\theta(f)$ is the graph of an n-ary function, so $\theta \in \mathcal{T}_{Env}[\![f\colon (\overline{\text{any}()}^{n}) \to \text{any}()]\!]$. Therefore, we get that $\theta \in \mathcal{T}_{Env}[\![\Gamma \sqcap [f\colon (\overline{\text{any}()}^{n}) \to \text{any}()]]\!]$. However, from the condition in rule [APP2] we get $\theta \in \mathcal{T}_{Env}[\![\, \bot \,]\!] = \varnothing$, leading to a contradiction.
- **Case [LET]**
  Assume a tuple $(\theta, v) \in \mathcal{E}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$. Then there exists some v_1 such that $(\theta, v_1) \in \mathcal{E}[\![e_1]\!]$, $(\theta[x/v_1], v) \in \mathcal{E}[\![e_2]\!]$. Since $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$ we get that $(\theta, v_1) \in \mathcal{E}[\![e_1]\!]\restriction_{\mathcal{T}_{Env}[\![\Gamma]\!]}$ and therefore it holds that $(\theta, v_1) \in \mathcal{T}[\![\overline{\langle \tau_i;\, \Gamma_i \rangle}^{n}]\!]$, where $\overline{\langle \tau_i;\, \Gamma_i \rangle}^{n}$ is the sequence of pairs used when applying the [LET] rule. This means that $(\theta, v_1) \in \mathcal{T}[\![\langle \tau_k;\, \Gamma_k \rangle]\!]$ for some $k \in \{1..n\}$. In particular there exists some π such that:

$$\begin{array}{l}\forall\, z \in \textbf{Var}.\, (\theta(z), \pi) \in \mathcal{T}[\![\Gamma_k(z)]\!] \\ \qquad (v_1, \pi) \in \mathcal{T}[\![\tau_k]\!] \\ \qquad\quad \pi \vDash \Gamma_k|_C\end{array}$$

  which we can rewrite as follows:

$$\begin{array}{l}\forall\, z \in \textbf{Var} \setminus \{x\}.(\theta[x/v_1](z), \pi) \in \mathcal{T}[\![\Gamma_k[x\colon \tau_k](z)]\!] \\ \qquad (\theta[x/v_1](x), \pi) \in \mathcal{T}[\![\Gamma_k[x\colon \tau_k](x)]\!] \\ \qquad\quad \pi \vDash \Gamma_k[x\colon \tau_k]|_C\end{array}$$

  The first two facts can be merged to get:

$$\begin{array}{l}\forall\, z \in \textbf{Var}.\, (\theta[x/v_1](z), \pi) \in \mathcal{T}[\![\Gamma_k[x\colon \tau_k](z)]\!] \\ \qquad\quad \pi \vDash \Gamma_k[x\colon \tau_k]|_C\end{array}$$

  and therefore $\theta[x/v_1] \in \mathcal{T}_{Env}[\![\Gamma_k[x\colon \tau_k]]\!]$. Since $(\theta[x/v_1], v) \in \mathcal{E}[\![e_2]\!]$ we can apply induction hypothesis on the derivation of $\Gamma_k[x\colon \tau_k] \vdash e_2\colon \rho_k$ so as to get . By Lemma 8 we get $(\theta, v) \in \mathcal{T}[\![\rho_k \setminus \{x\}]\!] \subseteq \mathcal{T}[\![\rho_i \setminus \overline{\{x\}}^{n}]\!]$, which proves the lemma.
- **Case [CAS]**
  Assume that $(\theta, v) \in \mathcal{E}[\![\textbf{case } x \textbf{ of } \overline{cls_i}^{n}]\!]$. There exists some $k \in \{1..n\}$ and $\overline{v_j}$ such that $(\theta[\overline{x_j/v_j}], v) \in \mathcal{E}[\![e']\!]$ and $\theta[\overline{x_j/v_j}] \in matches(\theta, \theta(x), cls_k)$ where cls_k is of the form $p \textbf{ when } e_g \mapsto e'$ and $\{\overline{x_j}\} = vars(p)$. Moreover, since we have applied [CAS] we know that $\Gamma(x) = \alpha$ and that $\Gamma \Vdash_\alpha cls_k\colon \rho_k$ for some ρ_k. We unfold the definitions of $matches(\theta, \theta(x), cls_k)$ and $\Gamma \Vdash_\alpha cls_k\colon \rho_k$ so as to get:

$$(\theta[\overline{x_j/v_j}], \theta(x)) \in \mathcal{E}[\![p]\!]\Gamma[\overline{x_j\colon \alpha_j}] \vdash p\colon \tau_p \tag{A.18}$$

$$(\theta[\overline{x_j/v_j}], \text{'true'}) \in \mathcal{E}[\![e_g]\!]\Gamma[\overline{x_j\colon \alpha_j} \mid \tau_p \Leftarrow \alpha] \vdash e_g\colon \overline{\langle \tau'_j;\, \Gamma'_j \rangle}^{m} \tag{A.19}$$

$$(\theta[\overline{x_j/v_j}], v) \in \mathcal{E}[\![e']\!] \quad \begin{array}{l}\Gamma'_j[\text{'true'} \subseteq \tau'_j] \vdash e\colon \rho'_j \\ \text{for every } j \in \{1..\, n\}\end{array} \tag{A.20}$$

  Since we know that $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$, there exists some π such that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma]\!]$. Let us denote by π' the instantiation $\pi[\overline{\alpha_j \mapsto \{v_j\}}]$. Since the $\overline{\alpha_j}$ variables are fresh, we know that $\theta \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma]\!]$ by Lemma 2 and hence $\theta[\overline{x_j/v_j}] \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma[\overline{x_j\colon \alpha_j}]]\!]$. Besides this, $(\theta(x), \pi') \in \mathcal{T}[\![\Gamma(x)]\!] = \mathcal{T}[\![\alpha]\!]$, so $\pi'(\alpha) = \{\theta(x)\}$. Therefore we can use these facts with (A.18) in order to apply Lemma 3 and obtain that $(\theta(x), \pi') \in \mathcal{T}[\![\tau_p]\!]$. Since we know that $\pi'(\alpha) = \pi(\alpha) = \{\theta(x)\}$ then it holds that $\pi' \vDash \tau_p \Leftarrow \alpha$ and therefore $\theta[\overline{x_j/v_j}] \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma[\overline{x_j\colon \alpha_j} \mid \tau_p \Leftarrow \alpha]]\!]$. Now we can apply induction hypothesis on the derivation (A.19) to obtain that $(\theta[\overline{x_j/v_j}], \text{'true'}) \in \mathcal{T}[\![\overline{\langle \tau'_j;\, \Gamma'_j \rangle}^{m}]\!]$. This implies the existence of some $l \in \{1..m\}$ such that $(\theta[\overline{x_j/v_j}], \text{'true'}) \in \mathcal{T}[\![\langle \tau'_l;\, \Gamma'_l \rangle]\!]$. There exists an instantiation π such that $(\text{'true'}, \pi) \in \mathcal{T}[\![\tau'_l]\!]$ and $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma'_l]\!]$. From the first fact it follows that $\pi \vDash \text{'true'} \subseteq \tau'_l$ which we can join with the second fact in order to get that $\theta[\overline{x_j/v_j}] \in \mathcal{T}[\![\Gamma'_l[\text{'true'} \subseteq \tau'_l]]\!]$. Now we can use (A.20) to apply the induction hypothesis and get $(\theta[\overline{x_j/v_j}], v) \in \mathcal{T}[\![\rho'_l]\!]$. From Lemma 8 we get $(\theta, v) \in \mathcal{T}[\![\rho'_l \setminus \{\overline{x_i}\}]\!] \subseteq \mathcal{T}[\![\overline{\rho'_j \setminus \{\overline{x_i}\}}^{m}]\!] = \mathcal{T}[\![\rho_k]\!] \subseteq \mathcal{T}[\![\overline{\rho_k}^{n}]\!]$, which proves the lemma.
- **Case [RCV]**
  Assume that $(\theta, v) \in \mathcal{E}[\![\textbf{receive } \overline{cls_i}^{n} \textbf{ after } x_t \to e']\!]$ with $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!]$. According to the semantic definition of **receive** expressions, there are two possibilities:

  [RCV-1] There is some $k \in \{1..n\}$, and some values $\overline{v_i}, v'$ such that $\theta(x_t) \in Integer \cup \{\text{'infinity'}\}$, $(\theta[\overline{x_i/v_i}], v) \in \mathcal{E}[\![e'']\!]$, and $\theta[\overline{x_i/v_i}] \in matches(\theta, v', cls_k)$, where $\overline{x_i}$ are the variables occurring in the pattern of cls_k and e'_k is the body of the k-th clause in the **receive** expression (i.e. cls_k).

  [RCV-2] $\theta(x_t) \in Integer$ and $(\theta, v) \in \mathcal{E}[\![e']\!]$.

  Firstly let us assume [RCV-1]. If $\theta(x_t) \in Integer \cup \{\text{'infinity'}\}$ then $\theta \in \mathcal{T}_{Env}[\![\Gamma_t]\!]$, where $\Gamma_t = [x_t\colon \text{integer}() \cup \text{'infinity'}]$; so $\theta \in \mathcal{T}_{Env}[\![\Gamma]\!] \cap \mathcal{T}_{Env}[\![\Gamma_t]\!] = \mathcal{T}_{Env}[\![\Gamma \sqcap \Gamma_t]\!]$. Equivalently, θ belongs to $\mathcal{T}^{\pi}_{Env}[\![\Gamma \sqcap \Gamma_t]\!]$ for some instantiation π. Now assume that the chosen clause cls_k is of the form $p \textbf{ when } e_g \to e''$, and $\overline{x_i}$ are the variables appearing in p. By definition of $matches(\theta, v', cls_k)$ it holds that $(\theta[\overline{x_i/v_i}], v') \in \mathcal{E}[\![p]\!]$. Let us denote by π' the instantiation $\pi[\alpha \mapsto \{v'\}, \overline{\alpha_i \mapsto \{v_i\}}]$, where α is the fresh variable that has been used in the

application of the [RCV] rule, and $\overline{\alpha_i}$ are the fresh variables used in the [CLS] rule that must have been applied to prove that $\Gamma \sqcap \Gamma_t \Vdash_\alpha cls_k: \rho_k$. Since π and π' only differ on those fresh variables, by Lemma 2 we get $\theta \in \mathcal{T}^{\pi'}_{Env}[\![\Gamma \sqcap \Gamma_t]\!]$. In fact, if we extend the environment we also get that $\theta[\overline{x_i/v_i}]$ is contained within $\mathcal{T}^{\pi'}_{Env}[\![(\Gamma \sqcap \Gamma_t)[\overline{x_i: \alpha_i}]]\!]$. We can then apply Lemma 3 and proceed as in the [CAS] case to prove the lemma.

Now we assume [RCV-2]. Since $\theta(x_t) \in$ *Integer* we get that $\theta \in \mathcal{T}_{Env}[\![\Gamma'_t]\!]$, where $\Gamma'_t = [x_t: \text{integer}()]$ and hence $\theta \in \mathcal{T}_{Env}[\![\Gamma \sqcap \Gamma'_t]\!]$. The lemma follows from applying the induction hypothesis to the derivation of $\Gamma \sqcap \Gamma'_t \vdash e': \rho$.

- **Case [LRC]**

In this case e is of the form **letrec** $\overline{x_i = f_i}^n$ **in** e'. Without loss of generality let us assume that $n = 1$, that is, that the **letrec** expression only defines one binding. The extension to two or more bindings is straightforward. In particular, assume that **letrec** $x = f$ **in** e', where f is an expression of the form $\mathbf{fun}(\overline{x_i}^m) \to e''$. By one of the assumptions of the [LRC] rule, we get that $\Gamma[x: \tau'] \vdash f: \langle \tau'; \Gamma[x: \tau'] \rangle$ for some τ'.

Let us define the function F_θ: **DVal** $\to$ **DVal** as follows: for each $v \in$ **DVal**,

$$F_\theta(v) = v' \quad \text{where } \{v\} = \{v' \mid (\theta[x/v], v') \in \mathcal{E}[\![f]\!]\}$$

Let us define the sequence $(v_k)_{k \in \mathbb{N}}$ as follows:

$$v_0 = \emptyset$$
$$v_k = F_\theta(v_{k-1}) \quad \text{for each } k > 0$$

It is easy to show that the semantics is monotone on the functions occurring in a fixed substitution θ. That is, if $(\theta[x/v_1], v'_1), (\theta[x/v_2], v'_2) \in \mathcal{E}[\![f]\!]$ and $v_1 \subseteq v_2$, then $v'_1 \subseteq v'_2$. Therefore, the sequence $(v_k)_k \in \mathbb{N}$ is, in fact, an ascending chain: $v_0 \subseteq v_1 \subseteq v_2 \subseteq \ldots$. Let us prove that, for each $k \in \mathbb{N}$, the substitution $\theta[x/v_k]$ belongs to $\mathcal{T}_{Env}[\![\Gamma[x: \tau']]\!]$. We proceed by induction on k:

- **Base case** ($k = 0$). Since f is a λ-abstraction, the [ABS] rule must have been used somewhere in the derivation tree of $\Gamma[x: \tau'] \vdash f: \langle \tau'; \Gamma[x: \tau'] \rangle$. Therefore, the semantics of τ' contains the empty graph $\emptyset$, hence $\theta[x/\emptyset] \in \mathcal{T}_{Env}[\![\Gamma[x: \tau']]\!]$.
- **Inductive step** ($k > 0$). Assume that $\theta[x/v_k] \in \mathcal{T}_{Env}[\![\Gamma_0]\!]$. Since $(\theta[x/v_k], v_{k+1}) \in \mathcal{E}[\![f]\!]$ and $\Gamma[x: \tau'] \vdash f: \langle \tau'; \Gamma[x: \tau'] \rangle$ we get, by induction hypothesis on this derivation that $(\theta[x/v_k], v_{k+1}) \in \mathcal{T}[\![\langle \tau'; \Gamma[x: \tau'] \rangle]\!]$. This implies the existence of some instantiation π such that $\theta \in \mathcal{T}^{\pi}_{Env}[\![\Gamma[x: \tau']]\!]$ and $(v_{k+1}, \pi) \in \mathcal{T}[\![\tau']\!]$, so $\theta[x/v_{k+1}] \in \mathcal{T}^{\pi}_{Env}[\![\Gamma[x: \tau']]\!] \subseteq \mathcal{T}_{Env}[\![\Gamma[x: \tau']]\!]$.

  As a consequence of this, it can be proved that $\theta[x/\bigcup_{k=1}^{\infty} v_k] \in \mathcal{T}_{Env}[\![\Gamma[x: \tau']]\!]$. By Tarksi's fixed point theorem, we get that $lfp\ F_\theta = \bigcup_{k=1}^{\infty} v_k$, so $\theta[x/lfp\ F_\theta] \in \mathcal{T}_{Env}[\![\Gamma[x: \tau']]\!]$. On the other hand, we get that $(\theta[x/lfp\ F_\theta], v) \in \mathcal{E}[\![e']\!]$, so we apply induction hypothesis on the derivation of $\Gamma[x: \tau'] \vdash e': \rho$ so as to get that $(\theta[x/lfp\ F_\theta], v) \in \mathcal{T}[\![\rho]\!]$, to which we apply Lemma 8 to obtain $(\theta, v) \in \mathcal{T}[\![\rho \setminus \{x\}]\!]$.

□

References

[1] N. Chechina, K. MacKenzie, S. Thompson, P. Trinder, O. Boudeville, V. Fördős, C. Hoch, A. Ghaffari, M.M. Hernandez, Evaluating scalable distributed Erlang for scalability and reliability, IEEE Trans. Parallel Distrib. Syst. 28 (8) (2017) 2244–2257.

[2] F. Cesarini, Which companies are using Erlang, and why?, 2019, https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html. Retrieved Oct. 18, 2019.

[3] L. Damas, R. Milner, Principal type-schemes for functional programs, Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of ProgrammingLanguages, ACM, 1982, pp. 207–212.

[4] S. Marlow, P. Wadler, A practical subtyping system for Erlang, Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97, ACM, New York, NY, USA, 1997, pp. 136–149, https://doi.org/10.1145/258948.258962.

[5] N. Valliappan, J. Hughes, Typing the Wild in Erlang, Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, Erlang 2018, ACM, New York, NY, USA, 2018, pp. 49–60, https://doi.org/10.1145/3239332.3242766.

[6] R. Jakob, P. Thiemann, A falsification view of success typing, in: K. Havelund, G.J. Holzmann, R. Joshi (Eds.), NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27–29, 2015, Proceedings, *Lecture Notes in Computer Science*, 9058 Springer, 2015, pp. 234–247, , https://doi.org/10.1007/978-3-319-17524-9_17.

[7] T. Lindahl, K. Sagonas, Detecting software defects in telecom applications through lightweight static analysis: a war story, Programming Languages and Systems, Springer, 2004, pp. 91–106.

[8] T. Lindahl, K. Sagonas, Practical type inference based on success typings, Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '06, ACM, New York, NY, USA, 2006, pp. 167–178.

[9] M. Jimenez, T. Lindahl, K.F. Sagonas, A language for specifying type contracts in Erlang and its interaction with success typings, in: S.J. Thompson, L. Fredlund (Eds.), Proceedings of the ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007, ACM, 2007, pp. 11–17.

[10] T. Lindahl, K. Sagonas, Typer: a type annotator of erlang code, Proceedings of the ACM SIGPLAN workshop on Erlang, ACM, 2005, pp. 17–25.

[11] F.J. López-Fraguas, M. Montenegro, J. Rodríguez-Hortalá, Polymorphic types in Erlang function specifications, Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4–6, 2016, Proceedings, (2016), pp. 181–197.

[12] F.J. López-Fraguas, M. Montenegro, G. Suárez-García, Polymorphic success types for Erlang, in: G. Barthe, G. Sutcliffe, M. Veanes (Eds.), LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16–21 November 2018, *EPiC Series in Computing*, 57 EasyChair, 2018, pp. 515–533.

[13] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, R. Virding, Core Erlang 1.0.3 language specification, 2004.

[14] J. Armstrong, Erlang, Commun. ACM 53 (9) (2010) 68–75, https://doi.org/10.1145/1810891.1810910.

[15] J.C. Reynolds, Types, abstraction and parametric polymorphism, Proceedings of the IFIP Congress, (1983), pp. 513–523.

[16] P. Wadler, Theorems for free!, in: J.E. Stoy (Ed.), Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989, London, UK, September 11–13, 1989, ACM, 1989, pp. 347–359, , https://doi.org/10.1145/99370.99404.

[17] J. Siek, W. Taha, Gradual typing for functional languages, Scheme and Functional Programming, (2006), pp. 81–92.

[18] J. Siek, W. Taha, Gradual typing for objects, ECOOP 2007 - Object-Oriented Programming(2007) 2–27. 10.1007/978-3-540-73589-2_2.

[19] G.M. Bierman, M. Abadi, M. Torgersen, Understanding Typescript, in: R.E. Jones (Ed.), ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28, - August 1, 2014. Proceedings, *Lecture Notes in Computer Science*, 8586 Springer, 2014, pp. 257–281, , https://doi.org/10.1007/978-3-662-44202-9_11.

[20] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, G. Levi, Fast and precise type checking for javascript, PACMPL 1 (OOPSLA) (2017) 48:1–48:30, https://doi.org/10.1145/3133872.

[21] N. Savage, Gradual evolution, Commun. ACM 57 (10) (2014) 16–18, https://doi.org/10.1145/2659764.

[22] D. Ancona, M. Ancona, A. Cuni, N.D. Matsakis, RPython: A step towards reconciling dynamically and statically typed OO languages, Proceedings of the 2007 Symposium on Dynamic Languages, DLS '07, ACM, New York, NY, USA, 2007, pp. 53–64, https://doi.org/10.1145/1297081.1297091.

[23] M. Furr, J.-h.D. An, J.S. Foster, M. Hicks, Static type inference for Ruby, Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, ACM, New York, NY, USA, 2009, pp. 1859–1866, https://doi.org/10.1145/1529282.1529700.

[24] R. Cartwright, M. Fagan, Soft typing, in: D.S. Wise (Ed.), Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26–28, 1991, ACM, 1991, pp. 278–292, ,

https://doi.org/10.1145/113445.113469.
[25] A.K. Wright, R. Cartwright, A practical soft type system for Scheme, LISP and Functional Programming, (1994), pp. 250–262, https://doi.org/10.1145/182409.182485.
[26] S. Nyström, A soft-typing system for Erlang, in: B. Däcker, T. Arts (Eds.), Proceedings of the ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003, ACM, 2003, pp. 56–71, , https://doi.org/10.1145/940880.940888.
[27] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, P. Vekris, Safe & efficient gradual typing for TypeScript, Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 167–180, https://doi.org/10.1145/2676726.2676971.
[28] M.M. Vitousek, A.M. Kent, J.G. Siek, J. Baker, Design and evaluation of gradual typing for Python, Proceedings of the 10th ACM Symposium on Dynamic Languages, DLS '14, ACM, New York, NY, USA, 2014, pp. 45–56, https://doi.org/10.1145/2661088.2661101.
[29] A. Rastogi, A. Chaudhuri, B. Hosmer, The ins and outs of gradual type inference, Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '12, (2012), p. 481, https://doi.org/10.1145/2103656.2103714.
[30] N. Heintze, J. Jaffar, A Finite Presentation Theorem for Approximating Logic Programs, Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90, ACM, New York, NY, USA, 1990, pp. 197–209, https://doi.org/10.1145/96709.96729.
[31] P.W. Dart, J. Zobel, A Regular Type Language for Logic Programs, Types in Logic Programming, (1992).
[32] E. Yardeni, E. Shapiro, A type system for logic programs, J. Logic Program. 10 (2) (1991) 125–153, https://doi.org/10.1016/0743-1066(91)80002-U.
[33] J.P. Gallagher, D.A. de Waal, Fast and precise regular approximations of logic programs, in: P.V. Hentenryck (Ed.), Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13–18, 1994, MIT Press, 1994, pp. 599–613.
[34] C. Vaucheret, F. Bueno, More precise yet efficient type inference for logic programs, in: M.V. Hermenegildo, G. Puebla (Eds.), Proceedings of the 9th International Symposium Static Analysis, SAS 2002, Madrid, Spain, September 17–20, 2002, Proceedings, *Lecture Notes in Computer Science*, 2477 Springer, 2002, pp. 102–116, , https://doi.org/10.1007/3-540-45789-5_10.
[35] P. Pietrzak, A type-based framework for locating errors in constraint logic programs, Linköping University Electronic Press, 2002 Ph.D. thesis.
[36] P. Pietrzak, J. Correas, G. Puebla, M.V. Hermenegildo, A practical type analysis for verification of modular prolog programs, in: R. Glück, O. de Moor (Eds.), Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM, San Francisco, California, USA, January 7–8, 2008, ACM, 2008, pp. 61–70, , https://doi.org/10.1145/1328408.1328418.
[37] R. Barbuti, R. Giacobazzi, A bottom-up polymorphic type inference in logic programming, Sci. Comput. Program. 19 (3) (1992) 281–313, https://doi.org/10.1016/0167-6423(92)90038-D.
[38] L. Lu, Improving precision of type analysis using non-discriminative union, Theory Pract. Logic Program. 8 (01) (2008) 33–79, https://doi.org/10.1017/S1471068407003055.
[39] L. Lu, A polymorphic type dependency analysis for logic programs, New Gener. Comput. 29 (4) (2011) 409–444, https://doi.org/10.1007/s00354-009-0117-5.
[40] J.-Y. Girard, The system f of variable types, fifteen years later, Theor. Comput. Sci. 45 (1986) 159–192, https://doi.org/10.1016/0304-3975(86)90044-7.