Polymorphic Types in Erlang Function Specifications *

Francisco J. López-Fraguas, Manuel Montenegro, and Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, montenegro@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. Erlang is a concurrent functional programming language developed by Ericsson, well suited for implementing distributed systems. Although Erlang is dynamically typed, the Dialyzer static analysis tool can be used to extract implicit type information from the programs, both for documentation purposes and for finding errors that will definitively arise at program execution. Dialyzer is based on the notion of success types, that correspond to safe over-approximations for the semantics of expressions. Erlang also supports user given function specifications (or just *specs*), that are contracts providing more information about the semantics of functions. Function specs are useful not only as documentation, but also can be employed by Dialyzer to improve the precision of the analysis. Even though specs can have a polymorphic shape, in practice Dialyzer is not able to exploit all their potential. One reason for that is that extending the notion of success types to a polymorphic setting is not trivial, and several interpretations are possible. In this work we propose a precise formulation for a novel interpretation of function specs as polymorphic success types, and a program transformation that allows us to apply this new interpretation on the call sites of functions with a declared spec. This results in a significant improvement in the number of definite errors that Dialyzer is able to detect.

1 Introduction

Erlang [2] is an eager concurrent functional programming language developed by Ericsson. It is a dynamically typed language, in contrast with languages like Haskell or ML where programs must be recognized as well typed by a static analysis, according typically with some variant of Hindley-Milner system [4], ensuring statically type safety, i.e., that the evaluation of a well-typed expression within a well-typed program will not incur a type clash at any step. This kind of analysis is conservative in the sense that it rejects programs that could be free of runtime errors in purely operational terms, but that are not detected as such by the analysis.

* Work partially supported by the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731) and UCM-Santander grants GR3/14-910502, GR3/14-910398.

Dynamic typing provides usually more liberality; however, runtime error detection means probably late error detection, a serious inconvenience in practice. To address this in Erlang, the Dialyzer static analysis tool was proposed in [7] with two essential design principles: it should be applicable to already existing Erlang programs and should not produce *false positives*: signalling a type error must imply that a runtime error will certainly happen. As it is said in [14], the lemma *'well-typed programs never go wrong'* of Hindley-Milner types is replaced in the Dialyzer approach by *'ill-typed programs always fail'*.

Dialyzer, success types and type specifications Dialyzer considers primitive types *integer, atom,...*, tuple types $\{\tau_1, \ldots, \tau_n\}$, list types $[\tau]$, functional types $(\tau_1, \ldots, \tau_n) \to \tau, \ldots$. Each individual Erlang value v is itself a type and the union $\tau_1|\tau_2$ of two types is also a type. Types represent sets of values that can be ordered by set inclusion. The empty and the total set of values are represented by the types *none* and *any*[1].

Dialyzer tries to infer *success types* [9] that are over-approximations for the semantics of expressions: τ is a success type for e if τ contains all the possible values to which e can be reduced. A type $(\tau_1, \ldots, \tau_n) \to \tau$ is a success type for a function f if whenever $f(e_1, \ldots, e_n)$ reduces to a value v then $v \in \tau$ and each e_i reduces to a value $v_i \in \tau_i$. Notice that *any* is a success type for any expression, that $(any, \ldots, any) \to any$ is a success type for any f, and that if *none* is a success type for e then e cannot be reduced to any value, thus indicating a definite error, not just a possible one. Singleton and union types allow Dialyzer to infer frequently quite precise success types, as this simple example shows:

```
f(0) -> 1 ; f(1) -> 0.   g(2) -> 0.          h(0) -> 0.
e1() -> f(0).            e2() -> g(e1()).    e3() -> h(e1()).
```

Dialyzer (more exactly, its associated tool Typer [8]) infers the following success types, reported in the form of `specs` (type specifications or signatures):

```
-spec f(0 | 1) -> 0 | 1.      -spec e1() -> 0 | 1.
-spec g(2) -> 0.              -spec e2() -> none.
-spec h(0) -> 0.              -spec e3() -> 0.
```

Dialyzer has detected that e_2 is not reducible to any value. The rest of specs are strict overapproximations of the corresponding semantics, but are nevertheless more precise than types like $Int \to Int$ (for f) or Int (for e_i) that would have been inferred in ML or Haskell. But, as we will see soon, Dialyzer has also important limitations which are the focus of this paper.

User given type specifications were considered in [6] and later on incorporated to Erlang, as contracts specifying the intended behavior of functions. They are useful as documentation and also used by Dialyzer to refine its analysis. For instance, the user could give the specification `-spec f(0) -> 1 ; (1) -> 0` that allows Dialyzer to refine its analysis, obtaining

[1] Written in actual Erlang as `none()`, `any()`, but we omit those `()` for types.

`-spec` e_1`() -> 1.` `-spec` e_2`() -> none.` `-spec` e_3`() -> none.`

which corresponds better (perfectly, in this case) to reality. In this paper we assume that user specs correspond indeed to success types, i.e., that the contract corresponding to each spec is fulfilled by its function definition. In practice, Dialyzer only checks that user specs are compatible with the inferred success types. In presence of user specs, errors reported by Dialyzer anticipate definite runtime errors or violations of the contract given by the specs. It is in this sense that the absence of false positives must be understood.

Dialyzer, polymorphism and this work The limitations of Dialyzer become quickly apparent with polymorphic functions. The simplest example is given by the function `id(X) -> X`, whose Hindley-Milner type would be $\forall\alpha.\alpha \to \alpha$. What Dialyzer infers is `-spec id(any) -> any`, with no connection between the two *any*'s. This causes a great loss of precision: for the function f above, Dialyzer infers *none* for $f(2)$, but $0|1$ for $f(id(2))$, since $id(2)$ is analyzed as *any*.

Could user specs come to our rescue? Yes ... in principle. User specs permit polymorphic specifications like `-spec id(X) -> X when X::any` where a condition `X::`τ expresses that `X` is a subtype of τ (so, in this case, any `X` fulfils `X::any`). Erlang's documentation [1] says that 'it is up to the tools that process the specifications to choose whether to take this extra information into account or not'. However, Dialyzer does not use it in a sensible way, but replaces each occurrence of X by its bound *any*, thus falling exactly into the same imprecisions as before. We do not know of any other Erlang tool that improves the situation. Moreover, it is not obvious how polymorphism of function specs must be interpreted, due to the union nature of success types.

Those are precisely the problems addressed in this paper: how to interpret polymorphic specifications in a setting of success types and how to improve Dialyzer's treatment of them. We do not investigate here inference of polymorphic success types, a subject of obvious interest but left to future work. We postpone until Sect. 3 the discussion of suitable interpretations of polymorphism, but we elaborate a bit more via examples our ideas for improving Dialyzer's behavior.

The kind of imprecisions pointed out with *id* occur with any other polymorphic function. Consider for instance *map* and two applications of it.

```
map(F,[]) when is_function(F) -> [];
map(F,[X|Xs]) -> [F(X)|map(F,Xs)].

e1() -> map(fun(X)->not(X) end,[1,2]). %this expression will fail
e2() -> map(fun(X)->not(X) end,[true,false]).
```

Dialyzer infers the rather imprecises `-spec map(fun(),[any]) -> [any]` and `-spec` e_i`() -> [any]`. Adding the polymorphic `-spec map(fun((A) -> B),[A1]) -> [B] when A1::A` does not help too much: we still obtain `-spec` e_i`() -> [any]`.

Forcing Dialyzer to be polymorphic To overcome the diagnosed problem we could have tried to identify which parts of Dialyzer should be changed or even to build a completely new inference system and tool. Instead, we have done

something much more lightweight: we run Dialyzer (as it is, no change in the tool) not over the original program but over a program transformation so that the effect is as if Dialyzer had used properly the polymorphic specifications provided in the program. The key idea is replacing *inline* each application of a function with a polymorphic spec by an expression having the same type as the original application and where the dependencies between types –lost in the direct use of Dialyzer– are kept and properly managed by Dialyzer because of the inlining. A convenient way of doing such a transformation is by means of parameterized Erlang macros, that are expanded at compile time. Abstracting out the macros to auxiliary functions would not be useful, because those functions would suffer of the same problems of the original ones. To get an idea of how this works, consider the *map* example: we distil a macro `MAP(F,L)` from the polymorphic spec of *map* and replace each application of *map* by one of `?MAP`.

```
-define(MAP(F,L),begin
                   F1 = F , L1 = L,
                   receive {A,A1,B} ->
                     F1 = ?FUN(A,B),
                     L1 = ?LIST(A1),
                     A = A1,
                     ?LIST(B)
                   end
                 end).
%  Other auxiliary macros FUN/2, LIST/2, ALT/2, ...
e1() -> ?MAP(fun(X)->not(X) end,[1,2]).
e2() -> ?MAP(fun(X)->not(X) end,[true,false]).
```

We leave detailed explanations for Sect. 4; but we remark that the transformation does not pretend to preserve evaluation, since `MAP` is not based on the code of *map* but only on its spec. The noticeable fact is that now Dialyzer infers for e_i the expected 'good' types: `-spec e1() -> none` and `-spec e2() -> [boolean]`. That is precisely the purpose of the transformation. Sect. 4 contains a complete transformation scheme that, being automatic and general, produces a slightly more complex code.

Organization of the paper The two main sections come after formalizing simple success types in Sect. 2. In Sect. 3 we discuss and propose a precise interpretation of function specifications as polymorphic success types. Sect. 4 contains the program transformation that forces Dialyzer to simulate polymorphic specs, as well as some results about its correctness. Some auxiliary technical contents, including proof sketches, have been left to a technical report [10].

2 Simple success types

In this section we formalize an interpretation of Dialyzer success types, which is hopefully equivalent to the original notion from [9, 8]. The main idea is that a

$\mathcal{CS}^0 = Atom \uplus Integer \uplus Float \uplus \{[]\} \uplus Pid$ $\mathcal{CS}^2 = \{[\_\|\_], \{\_, \_\}\}$ $\mathcal{CS}^n = \{\{\_, \ldots^n, \_\}\}\ \forall n \in \mathbb{N}\backslash\{2\}$	$DVal \simeq Atom \oplus Integer \oplus Float \oplus Pid \oplus \{[]\} \oplus \bigoplus_{c \in \mathcal{CS}^n} \{c\} \otimes DVal \otimes \ldots^n \otimes DVal \ \oplus \bigoplus_{n \in \mathbb{N}} (DVal \otimes \ldots^n \otimes DVal) \hookrightarrow \mathcal{P}(DVal)$

Fig. 1. Definition of the set of Erlang values

success type τ for an expression e represents a safe over-approximation for the semantics of e, formalized through a denotational semantics for expressions and types that gives e a smaller denotation than that for τ in a preorder over the semantic domain.

For this task we use a variation of Core Erlang [3], that is expressive enough to represent most Erlang programs, but that allows for a simpler presentation. A detailed description of the syntax and denotational semantics of the considered language is available in [10]. We use a reflexive semantic domain $DVal$ (see Fig. 1), whose definition is based on standard primitive domains and standard domain constructors, which ensure it is correctly defined [5]. The denotation of expressions is defined by the semantic function $\mathcal{E}[\![\_]\!] : Exp \to (\mathcal{UFS} \to Exp) \to \mathcal{P}(DVal)$ where Exp is the set of expressions, and definitional environments $\Lambda \in \mathcal{UFS} \to Exp$ are mappings from user function symbols to Exp that serve to model programs. We write $\mathcal{E}[\![e]\!]^\Lambda$ for the semantics of e within Λ and frequently omit Λ when implied by the context. In general $\mathcal{E}[\![e]\!]$ is a set of values, due to the non-determinism caused by concurrency primitives like `receive`. Note that values can be functions, hence $\mathcal{E}[\![e]\!]$ can be a set of functions. We will need some notations regarding (sets of) functions: we write $f|_C$ for the restriction of a function f to a subset C of its domain; the range restriction of f is denoted by $f|_C^{-1}$, and is defined by $f|_C^{-1}(x) = f(x)$ iff $f(x) \in C$; $f|_C^{-1}(x)$ is undefined otherwise. This is extended to set of functions as $fs|_C(x) = \{f|_C(x) \mid f \in fs\}$, and $fs|_C^{-1}(x) = \{f|_C^{-1}(x) \mid f \in fs\}$. We define the application of a set of functions fs with common domain to a set of values vs in that domain as $fs(vs) = \{f(v) \mid f \in fs, v \in vs\}$, and to a value as $fs(v) = fs(\{v\})$.

We consider a preorder $e \sqsubseteq e'$ on $DVal$ (see [10]) and extend it to $\mathcal{P}(DVal)$ to capture the notion that $\mathcal{E}[\![e']\!]$ is more powerful than $\mathcal{E}[\![e]\!]$, in the sense that for each value in $\mathcal{E}[\![e]\!]$ there is a greater one in $\mathcal{E}[\![e']\!]$ (i.e. a function with a greater graph, a tuple with greater elements, ...).

We refer to the original success types from [9, 8] as *simple* success types, to stress their difference to the success *type schemes* we will consider for function specs in Sect. 3, reminding to what is usually done also in Hindley-Milner-like type systems. We assume a set of type variables $\mathcal{TV}$ and use $\alpha, \beta \in \mathcal{TV}$. The set of simple types $\mathcal{T}$ is defined in Fig. 2. To better reflect its meaning, we write here $\tau \cup \tau'$ instead of the concrete Erlang syntax $\tau \mid \tau'$. Notice that individual values $v \in Val$ are types, where Val is a subset of Exp that only contains (intensional) values. We assume the existence of a denotation $\mathcal{V}[\![\_]\!]$ of these intensional values—see [10] for details—. The type $nelist(\tau, \tau')$ stands for (possibly improper) not empty lists with elements of type τ and ending of type τ'; note all the variant

$$\mathcal{TC}^0 \ni C^0 ::= none \mid any \mid atom \mid integer \mid float \mid pid \mid v \quad (v \in Val)$$
$$\mathcal{TC}^2 \ni C^2 ::= \_ \cup \_ \mid nelist(\_,\_) \qquad \mathcal{TC}^n \ni C^n ::= \{\_,\overset{n}{\ldots},\_\}$$
$$\mathcal{TC}^{n+1} \ni C^{n+1} ::= (\_,\overset{n}{\ldots},\_) \to \_$$
$$\mathcal{T} \ni \tau ::= \alpha \mid C^n(\tau_1,\ldots,\tau_n)$$

$$\mathcal{T}[\![none]\!] = \emptyset \quad \mathcal{T}[\![any]\!] = DVal \quad \mathcal{T}[\![atom]\!] = Atom \quad \mathcal{T}[\![integer]\!] = Integer$$
$$\mathcal{T}[\![float]\!] = Float \quad \mathcal{T}[\![pid]\!] = Pid \quad \mathcal{T}[\![v]\!] = \{\mathcal{V}[\![v]\!]\ [\,]\} \quad \mathcal{T}[\![\tau_1 \cup \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] \cup \mathcal{T}[\![\tau_2]\!]$$
$$\mathcal{T}[\![nelist(\tau_v,\tau_c)]\!] =$$
$$\quad lfp\ (\lambda Z.\{([\,|\,],v,c) \mid v \in \mathcal{T}[\![\tau_v]\!], c \in \mathcal{T}[\![\tau_c]\!]\} \cup \{([\,|\,],v,z) \mid v \in \mathcal{T}[\![\tau_v]\!], z \in Z\})$$
$$\mathcal{T}[\![\{\tau_1,\ldots,\tau_n\}]\!] = \{(\{,\overset{n}{\cdots},\},v_1,\ldots,v_n) \mid \forall i \in \{1..n\}.v_i \in \mathcal{T}[\![\tau_i]\!]\}$$
$$\mathcal{T}[\![(\tau_1,\ldots,\tau_n) \to \tau]\!] = \{ \bigoplus_{z \in Dom} \hat{\lambda} z.\mathcal{T}[\![\tau]\!]\} \text{ where } Dom \overset{\text{def}}{=} \prod_{i=1}^{n} \mathcal{T}[\![\tau_i]\!]$$

Fig. 2. Syntax and semantics of simple types

types for lists from [1] can be expressed with *nelist* and $\cup$: for example `list(0 | 1)` can be expressed as $nelist(0 \cup 1, [\,])$, as $[\,] \in \mathcal{CS}^0$ implies $[\,] \in Val$, hence $[\,] \in \mathcal{TC}^0$ and so $[\,] \in \mathcal{T}$. Type substitutions $\pi \in TSubst$ are finite mappings $\pi : \mathcal{TV} \to \mathcal{T}$. We say π is ground when $\pi(\alpha)$ is ground for any α.

The denotation $\mathcal{T}[\![\tau]\!]$ of a simple type τ is a set of semantic values in $DVal$ given by the mapping $\mathcal{T}[\![\_]\!] : \mathcal{T} \to \mathcal{P}(DVal)$ defined in Fig. 2. The notation $\hat{\lambda} v_1.v_2$ denotes a function with a single binding from v_1 to v_2 —i.e., with a single point as graph— whereas the $\oplus$ operator merges two functions provided their domains are disjoint. Note $\mathcal{T}[\![]\!]$ is for example able to distinguish the type *any* from the type $(any) \to any$, as $\mathcal{T}[\![any]\!] = \mathcal{P}(DVal)$ and $\mathcal{T}[\![(any) \to any]\!] = DVal \hookrightarrow DVal$, and therefore $\mathcal{T}[\![any]\!] \ni 0 \notin \mathcal{T}[\![(any) \to any]\!]$, showing that *any* contains more values than $(any) \to any$.

We formalize that hierarchy among types by overloading the preorder $\sqsubseteq$ on $DVal$ to $\mathcal{T}$ as $\tau_1 \sqsubseteq \tau_2$ iff $\mathcal{T}[\![\tau_1]\!] \sqsubseteq \mathcal{T}[\![\tau_2]\!]$. Note none of the overloadings of $\sqsubseteq$ is a partial order, as they are not antisymmetric: for example, with $\tau_1 = 0 \cup integer$, $\tau_2 = integer$ we have $\tau_1 \sqsubseteq \tau_2$ and $\tau_2 \sqsubseteq \tau_1$, but τ_1 and τ_2 are different types. Nevertheless, as is standard, this preorder defines a partial order on the quotient set for the equivalence relation $\tau' \sqsubseteq \tau \ \wedge \tau \sqsubseteq \tau'$, which is a lattice [8, 9] with *any* as $\top$ and *none* as $\bot$. For the remainder of the paper, when using $\sqcup$ or $\sqcap$ on elements of $\mathcal{T}$, we implicitly work modulo that equivalence relation. We can now use the denotational semantics to formulate with precision the notion of success types.

Definition 1 (Success type, for simple types). *We say that $\tau \in \mathcal{T}$ is a success type for $e \in Exp$, written $e : \tau$, iff $\mathcal{E}[\![e]\!] \sqsubseteq \mathcal{T}[\![\tau]\!]$.*

Example 1. $\tau_1 = (0 \cup 1) \to 0 \cup 1$ is a success type for the expression $e = fun(X) \to case\ X\ of\ 0 \to 0$, as $\mathcal{E}[\![e]\!] = \hat{\lambda}(0).\{0\} \sqsubseteq \hat{\lambda}(0).\{0,1\} \ \oplus \ \hat{\lambda}(1).\{0,1\} = \mathcal{T}[\![(0 \cup 1) \to 0 \cup 1]\!]$. The type τ_1 is not the only valid success type for e. For example, $(0) \to 0$ is a more precise one.

This formulation tries to generalize Def. 1 from [9], that is only defined for functions, to arbitrary expressions. In general, expressions may have more than

one success type, because $e : \tau_1$ and $\tau_1 \sqsubseteq \tau_2$ imply $e : \tau_2$. In particular $e : \mathit{any}$ for all expressions e. On other other hand, $e : \mathit{none}$ is equivalent to $\mathcal{E}[\![e]\!] \sqsubseteq \mathcal{T}[\![\mathit{none}]\!] = \emptyset$, which implies that no value can be computed for e, i.e. that evaluating e will surely lead to a runtime error.

3 Success type schemes

Following the official Erlang documentation [1], we define the set of success type schemes $\mathcal{TS}$ as: $\mathcal{TS} \ni \sigma ::= \forall \alpha_1, \ldots, \alpha_m.\ \tau \mid \tau_1^1 \subseteq \tau_2^1, \ldots, \tau_1^l \subseteq \tau_2^l$ for $\alpha_i \in \mathcal{TV}$, $\tau, \tau_i^j \in \mathcal{T}$. This notion of type schemes expresses a form of bounded polymorphism, as type variables can be instantiated only with types that respect the corresponding type inclusion constraint. We use the semantics above to characterize these constraints, so $\tau_1 \subseteq \tau_2$ is satisfied iff $\emptyset \neq \mathcal{T}[\![\tau_1]\!]$ and $\tau_1 \sqsubseteq \tau_2$. Note success type schemes are just another presentation of Erlang function specs, and that $\mathcal{TS}$ contains type schemes corresponding to overloaded specs, which have the general form $\forall \alpha_1, \ldots, \alpha_m.\ (\tau_{p_1}^1, \ldots, \tau_{p_n}^1) \to \tau_r^1 \cup \ldots \cup (\tau_{p_1}^o, \ldots, \tau_{p_n}^o) \to \tau_r^o \mid \tau_1^1 \subseteq \tau_2^1, \ldots, \tau_1^l \subseteq \tau_2^l$ for a given $f \in \mathcal{FS}^n$. For this reason we will use the terms 'function spec' and 'type scheme' interchangeably for the rest of the paper. Notice that, in this system, overloaded schemes can be understood as union types (represented with the $\cup$ operator), since success types overapproximate the behaviour of programs. This contrasts with the traditional approach in which overloading is achieved via intersection types [12].

Just like we have characterized whether a simple type is a success type for an expression or not, we are interested in defining when a type scheme is a success type scheme for a function. We discuss here the issue.

A first obvious approach would be trying to mimic Def. 1, for which we would need a suitable definition for the denotation of a success type scheme $\mathcal{TS}[\![\_]\!] : \mathcal{TS} \to \mathcal{P}(\mathit{DVal})$, and then require $\mathcal{E}[\![f]\!] \sqsubseteq \mathcal{TS}[\![\sigma]\!]$ for σ to be a success type scheme of f. Let us consider for now a simplified setting where specs σ have the shape $\forall \overline{\alpha_j \subseteq \tau_j}.\ \tau$. A first possible definition of $\mathcal{TS}[\![\sigma]\!]$ could be $\mathcal{TS}[\![\sigma]\!] = \mathcal{T}[\![C(\sigma)]\!]$ where $C(\sigma)$ is the *compaction* of σ, the simple type resulting of replacing type variables by their bounds, i.e., $C(\forall \overline{\alpha_j \subseteq \tau_j}.\ \tau) = \tau[\overline{\alpha_j / \tau_j}]$. However, this corresponds to the observed behaviour of Dialyzer, as described through the examples of *id* and *map* in Sect. 1, for which we know that the polymorphism nature of type schemes is lost.

We could also consider the other extreme, with the following "singleton" interpretation of success type schemes in which a polymorphic type variable is instantiated with individual values taken from the denotation of its bound:

$$\mathcal{TS}[\![\forall \overline{\alpha_j \subseteq \tau_j}.\ \tau]\!] = \bigcup_{\overline{v_j \in \mathcal{T}[\![\tau_j]\!] \cap \mathit{DVal}}} \mathcal{T}[\![\tau[\overline{\alpha_j / v_j}]]\!]$$

Just like the previous interpretation was too loose, this interpretation is too strict; for instance, when applied to $\mathit{take} : \forall \alpha_e \subseteq \mathit{any}, \alpha_t \subseteq \mathit{any}.\ (\mathit{integer}, [\,] \cup \mathit{nelist}(\alpha_e, \alpha_t)) \to [\,] \cup \mathit{nelist}(\alpha_e, \alpha_t)$ it does not allow α_e to be instantiated

with $0 \cup 1$, because that does not correspond to a value but to a set of values. Therefore $take(1, [0 \mid [1 \mid []]])$ would be considered a contract violation. So maybe we should try with something in the middle. The following interpretation allows to instantiate the type variables of a type scheme with any subtype of its bound.

$$\mathcal{TS}[\![\forall\overline{\alpha_j \subseteq \tau_j}.\ \tau]\!] = \bigcup_{\overline{\tau'_j \subseteq \tau_j}} \mathcal{T}[\![\tau[\overline{\alpha_j/\tau'_j}]]\!]$$

This seems to corresponds to the polymorphic treatment of the list constructor that can be observed in Dialyzer: for example the type $nelist(0 \cup 1, [])$ is inferred for the list $[1, 0]$. Sadly, the condition $\mathcal{E}[\![f]\!] \sqsubseteq \mathcal{TS}[\![\sigma]\!]$ for this $\mathcal{TS}$ is just as strong as $\mathcal{E}[\![f]\!] \sqsubseteq \mathcal{T}[\![C(\sigma)]\!]$, i.e., the first $\mathcal{TS}$ we considered. The problem is that the supremum of $\{\tau[\overline{\alpha_j/\tau'_j}] \mid \tau'_j \subseteq \tau_j\}$ is precisely $C(\sigma)$. So this interpretation is as loose as the first one.

Nevertheless, we are quite close to the final interpretation of type schemes we propose in this paper, which at the end does not define a semantics $\mathcal{TS}[\![\sigma]\!]$ for type schemes, but needs to be more complex. Let us define the decomposition of a type scheme, $D(\_) : \mathcal{TS} \to \mathcal{P}(\mathcal{T})$ as $D(\forall\overline{\alpha_j \subseteq \tau_j}.\ \tau) = \{\tau[\overline{\alpha_j/\tau'_j}] \mid \tau'_j \subseteq \tau_j\}$. It is easy to check that for any type scheme $\bigsqcup D(\sigma) = C(\sigma)$, and that $C(\sigma) \in D(\sigma)$. What it is interesting about $D(\sigma)$ is that it corresponds to a decomposition of the semantics $\mathcal{T}[\![C(\sigma)]\!]$ as $\{\mathcal{T}[\![\tau]\!] \mid \tau \in D(\sigma)\} \in \mathcal{P}(\mathcal{P}(DVal))$. The idea then is that σ is a success type scheme for f iff $f : C(\sigma)$ and $\mathcal{E}[\![f]\!]$ can be decomposed following $D(\sigma)$. We formalize this idea through several conditions that must be satisfied by the semantics of any function for which a type scheme is declared. These are understood as additional conditions that are part of the contract the programmer assumes when declaring a function spec. We continue focusing on simplified declarations $f : \sigma$ for $\sigma = \forall\overline{\alpha_j \subseteq \tau_j}.\ (\overline{\tau_p}) \to \tau_r$. Then $D(\sigma)$ defines the following decomposition of $\mathcal{T}[\![C(\sigma)]\!]$:

$$\{\mathcal{T}[\![((\overline{\tau_p}) \to \tau_r)[\overline{\alpha_j/\tau'_j}]]\!] \mid \overline{\tau'_j \subseteq \tau_j}\}$$

For $f : \sigma$ we require the following condition to hold for any $\overline{\tau'_j \subseteq \tau_j}$:

$$\mathcal{E}[\![f]\!]|_{\mathcal{T}[\![(\overline{\tau_p})[\overline{\alpha_j/\tau'_j}]]\!]} \sqsubseteq \mathcal{T}[\![((\overline{\tau_p}) \to \tau_r^l)[\overline{\alpha_j/\tau'_j}]]\!]$$

With this condition we are saying that f defines a relation between input arguments and function results that respects the shape of the semantics decomposition expressed by σ. Consider for example the identity function $id = fun(X) \to X$, and assume we declare $id : \forall\alpha \subseteq any.\ (\alpha) \to \alpha$. It is easy to see that for $\tau \subseteq any$ we have that $id|_{\mathcal{T}[\![\tau]\!]} \in \mathcal{T}[\![(\tau) \to \tau]\!]$, because id just returns its input argument. Now we can use the inequality above to conclude that $id(0) : 0$, reasoning only with the specification $id : \forall\alpha \subseteq any.\ (\alpha) \to \alpha$, regardless of the concrete definition of id, in the line of Wadler's 'free theorems' [15].

Proof. $\mathcal{E}[\![id(0)]\!] = \mathcal{E}[\![id]\!](\mathcal{E}[\![0]\!]) = \mathcal{E}[\![id]\!]|_{\mathcal{E}[\![0]\!]}(\mathcal{E}[\![0]\!]) = \mathcal{E}[\![id]\!]|_{\mathcal{T}[\![0]\!]}(\mathcal{E}[\![0]\!])$, as $\mathcal{E}[\![0]\!] \sqsubseteq \mathcal{T}[\![0]\!]$. But then by $id : \forall\alpha \subseteq any.\ (\alpha) \to \alpha$ and using the inequality above with $\overline{\tau'_j} = 0$ we have $\mathcal{E}[\![id]\!]|_{\mathcal{T}[\![0]\!]}(\mathcal{E}[\![0]\!]) \sqsubseteq \mathcal{T}[\![(0) \to 0]\!](\mathcal{E}[\![0]\!])$, and we can use $0 : 0$ to

get $\mathcal{T}[\![(0) \to 0]\!](\mathcal{E}[\![0]\!]) \sqsubseteq \mathcal{T}[\![(0) \to 0]\!](\mathcal{T}[\![0]\!]) = (\hat{\lambda}0.\{0\})(\{0\}) = \{0\}$. So we have $\mathcal{E}[\![id(0)]\!] \sqsubseteq \{0\} \sqsubseteq \mathcal{T}[\![0]\!]$, i.e. $id(0) : 0$.

On the other hand, for $g = fun(X) \to case\ X\ of\ \{Y_1\ when\ is\_integer(X) \to 0, Y_2\ when\ true \to X\}$ that inequality does not hold for the declaration $g : \forall\alpha \subseteq any.\ (\alpha) \to \alpha$, because for $\tau = 1$ we have that $\mathcal{E}[\![g]\!]|_{\mathcal{T}[\![1]\!]}(1) = \{0\} \notin \mathcal{T}[\![(1) \to 1]\!](1)$. One way to see this, that might be familiar to functional programmers, is that the inequality condition above tries to capture the notion of parametricity first proposed in Reynolds' abstraction theorem [13], and later exploited in [15]. The function g breaks parametricity, because its rules inspect the variable X, which has a polymorphic type α. Conversely, id respects parametricity, because it does not inspect its polymorphic argument, and just returns it untouched.

That was a form of bottom-up information flow, where the type of a function argument affects the type of the whole function application. For $f : \sigma$ with $\sigma = \forall\overline{\alpha_j \subseteq \tau_j}.\ (\overline{\tau_p}) \to \tau_r$ we also require the following inequality condition, that corresponds to top down information flow, that should hold for any $\overline{\tau'_j \subseteq \tau_j}$:

$$\mathcal{E}[\![f]\!]|^{-1}_{\mathcal{T}[\![\tau_r[\overline{\alpha_j/\tau'_j}]]\!]} \sqsubseteq \mathcal{T}[\![(\overline{\tau_p}) \to \tau^l_r)[\overline{\alpha_j/\tau'_j}]]\!]$$

We can use this inequality for equational reasoning with id, but now with top-down information flow, where the type of a function application affects the type of its arguments. In particular we will conclude (again by using *only* the type specification of id) that if $id(a)$ is evaluated to some value $v : 0$, then in that evaluation a must be reduced to a value $v_a : 0$, i.e. 0 is a value for a.

Proof. By hypothesis $\mathcal{E}[\![id(a)]\!] \sqsubseteq \mathcal{T}[\![0]\!]$. Also $\mathcal{E}[\![id(a)]\!] = \mathcal{E}[\![id]\!](\mathcal{E}[\![a]\!])$, hence $\mathcal{E}[\![id(a)]\!] = \mathcal{E}[\![id]\!]|^{-1}_{\mathcal{T}[\![0]\!]}(\mathcal{E}[\![a]\!]) \sqsubseteq \mathcal{T}[\![(0) \to 0)]\!](\mathcal{E}[\![a]\!])$ using the inequality above with $\overline{\tau'_j} = 0$. So, given $v \in \mathcal{E}[\![id(a)]\!]$ we have $v \in \mathcal{T}[\![(0) \to 0]\!](\mathcal{E}[\![a]\!])$. This implies that to compute any $v \in \mathcal{E}[\![id(a)]\!]$ we need compute some $v_a \in \mathcal{E}[\![a]\!]$ such that $v_a \in dom(\mathcal{T}[\![(0) \to 0]\!]) = \mathcal{T}[\![0]\!]$, i.e. $v_a : 0$.

Additionally, for $f : \sigma$ we require $f : C(\sigma)$, in order to avoid accepting trivially small type schemes. For example, that condition rejects $id : (0) \to 0$, since $(0) \to 0$ is not a success type for id.

After giving the intuitions, we generalize the previous conditions to arbitrary types schemes for function symbols of the shape $\forall\overline{\alpha_j}.\ \bigcup^o_{l=1}(\overline{\tau^l_{p_i}}) \to \tau^l_r \mid \overline{\tau^k_1 \subseteq \tau^k_2}$. We first need to generalize the notion of compaction. For any conjunction of constraints $\overline{\tau^k_1 \subseteq \tau^k_2}$ the set of its solutions is the set $Sol(\overline{\tau^k_1 \subseteq \tau^k_2})$ of ground $\pi \in TSubst$ such that $var(\overline{\tau^k_1 \subseteq \tau^k_2}) \subseteq dom(\pi)$ and $(\overline{\tau^k_1 \subseteq \tau^k_2})\pi$ is satisfied. For any $\sigma \in \mathcal{TS}$, given $\sigma = \forall\overline{\alpha_j}.\ \bigcup^o_{l=1}(\overline{\tau^l_{p_i}}) \to \tau^l_r \mid \overline{\tau^k_1 \subseteq \tau^k_2}$ we assume $var(\sigma) \subseteq var(\overline{\tau^k_1 \subseteq \tau^k_2})$ without loss of generality, by adding additional trivial constraints $\alpha \subseteq any$ for any $\alpha \in var(\sigma) \setminus var(\overline{\tau^k_1 \subseteq \tau^k_2})$. Then the compaction of σ is defined as $C(\sigma) = (\bigcup^o_{l=1}(\overline{\tau^l_{p_i}}) \to \tau^l_r)\pi_s$ for $\pi_s = \bigsqcup Sol(\overline{\tau^k_1 \subseteq \tau^k_2})$.

Definition 2 (Success type, for type schemes). *For any $f \in \mathcal{FS}^n$ and $\sigma \in \mathcal{TS}$ we say that σ is a success type scheme for f, denoted $f : \sigma$, iff given*

$\sigma = \forall\overline{\alpha_j}.\ \bigcup_{l=1}^{o}(\overline{\tau_{p_i}^l}) \to \tau_r^l \mid \overline{\tau_1^k \subseteq \tau_2^k}$ *we have* $f : C(\sigma)$ *and the following conditions are met for* $\pi_s = \bigsqcup Sol(\overline{\tau_1^k \subseteq \tau_2^k})$*:*

1. *For any* $\pi \in Sol(\overline{\tau_1^k \subseteq \tau_2^k})$*, given* $\pi_p^l = \pi|_{var(\overline{\tau_p^l})}$ *then*

$$\mathcal{E}[\![f]\!]|_{\bigcup_{l=1}^{o}\mathcal{T}[\![(\overline{\tau_{p_i}^l})\pi_p^l]\!]} \sqsubseteq \bigcup_{l=1}^{o}\mathcal{T}[\![(\overline{\tau_{p_i}^l})\pi_p^l \to \tau_r^l\pi_p^l\pi_s]\!]$$

2. *For any* $\pi \in Sol(\overline{\tau_1^k \subseteq \tau_2^k})$*, given* $\pi_r^l = \pi|_{var(\tau_r^l)}$ *then*

$$\mathcal{E}[\![f]\!]|^{-1}_{\bigcup_{l=1}^{o}\mathcal{T}[\![\tau_r^l\pi_r^l]\!]} \sqsubseteq \bigcup_{l=1}^{o}\mathcal{T}[\![(\overline{\tau_{p_i}^l})\pi_r^l\pi_s \to \tau_r^l\pi_r^l]\!]$$

3. *For any ground* π_p *such that* $dom(\tau_p) \subseteq \bigcup_{l=1}^{o} var(\overline{\tau_p^l})$ *and* $Sol((\overline{\tau_1^k \subseteq \tau_2^k})\pi_p) = \emptyset$ *then* $\mathcal{E}[\![f]\!]|_{\bigcup_{l=1}^{o}\mathcal{T}[\![(\overline{\tau_{p_i}^l})\pi_p]\!]} = \emptyset$
4. *For any ground* π_r *such that* $dom(\tau_p) \subseteq \bigcup_{l=1}^{o} var(\tau_r^l)$ *and* $Sol((\overline{\tau_1^k \subseteq \tau_2^k})\pi_r) = \emptyset$ *then* $\mathcal{E}[\![f]\!]|^{-1}_{\bigcup_{l=1}^{o}\mathcal{T}[\![\tau_r^l\pi_r]\!]} = \emptyset$

Items 1. and 2. of Def. 2 express the relation between function input and outputs. These are basically the same we discussed above, with minor modifications over the domain of solutions π, that for the sake of readability were omitted in the presentation above. To understand these changes, let's consider a function f with declared spec $f : \sigma$ with $\sigma = \forall\alpha.\ (\alpha) \to 0 \mid \alpha \subseteq any$. A function with this type can only return 0 regardless of its argument. So for any expression e used as an argument for f, we have $\mathcal{E}[\![f(e)]\!] = \mathcal{E}[\![f]\!]|^{-1}_{\mathcal{T}[\![0]\!]}(\mathcal{E}[\![e]\!])$, and we should not be able to say a thing about the type of e, because α does not appear in the right hand side of σ. If we could, then we would be able to perform wrong deductions, thus introducing false positives in Dialyzer. A less artificial example would be the case of *map*, assuming the same spec as in Sect. 1: $\forall\alpha, \alpha_1, \beta.\ ((\alpha) \to \beta, list(\alpha_1)) \to list(\beta) \mid \alpha_1 \subseteq \alpha, \alpha \subseteq any, \beta \subseteq any$. Assuming for instance a call $map(g, [e])$, in a context that forces $map(g, [e]) : [true \cup false]$, then β should be instantiated to $true \cup false$, but we should not be able to infer derive any constraint about the domain of g from that.

Regarding items 3. and 4. of Def. 2, they express relationships between polymorphic variables, which can be used to fail when an instantiation of the variables results in unsolvable constraints. Consider again the example of *map* together with the function *not* defined as $fun(X) \to case\ X\ of\ \{true \to false; false \to true\}$; then it is clear that the call $map(fun(X) \to not(X),\ [1, 2])$ will fail. We can use item 3. of Def. 2 to deduce $\mathcal{E}[\![map(fun(X) \to not(X), [1, 2])]\!] = \mathcal{E}[\![map]\!]|_{((\alpha)\to\beta, list(\alpha_1))\pi_p}(\mathcal{E}[\![(fun(X) \to not(X)]\!], \mathcal{E}[\![[1, 2])]\!]) = \emptyset(\mathcal{E}[\![(fun(X) \to not(X)]\!], \mathcal{E}[\![[1, 2])]\!]) = \emptyset = \mathcal{T}[\![none]\!]$ for $\pi_p = [\alpha/true \cup false, \alpha_1/1 \cup 2, \beta/true \cup false]$, as $1 \cup 2 \subseteq true \cup false$ has no solution.

Although certainly complex, the notion of success type scheme given by us corresponds to the intuitive idea of parametricity honouring function. That can

be understood in simple terms considering that polymorphic functions should not inspect data variables with a polymorphic variable as type, i.e. data variables with a polymorphic variable as type are a kind of opaque data container. Note we can still perform matching against the constructed part of a polymorphic variable, as for example in *head* defined by $fun(Xs) \to case\ Xs\ of\ [X|\_] \to X$, declared as $head : \forall\alpha.\ nelist(\alpha) \to \alpha \mid \alpha \subseteq any$. In this case we inspect Xs, but only for the constructed fragment *nelist* of its type $nelist(\alpha, [\,])$. The same applies to the typical operations in polymorphic lists like $map, take, filter, \ldots$ These notions should be familiar to the seasoned functional programmer, that would then have an intuitive understanding of the additional contract she is accepting by assuming Def. 2.

At the time of writing, Erlang only allows the programmer to place specs in top-level function definitions. However, we can also apply Def. 2 to expressions that are always evaluated to a function.

4 A program transformation for simulating success types schemes

In this section we introduce an algorithm that transforms a given program by substituting macro expansions for polymorphic function calls. For each ground type τ there is a macro which is replaced with an Erlang term with the same semantics as $\mathcal{T}[\![\tau]\!]$. We can build these terms in a compositional way. For each function definition f/n with a monomorphic type $\overline{\tau_i}^n \to \tau$, the algorithm generates a macro with n arguments, and is expanded to a term that overapproximates $\mathcal{T}[\![\tau]\!]$ provided the set of possible arguments of the macro overapproximate their corresponding $\mathcal{T}[\![\tau_i]\!]$. If f/n has a polymorphic type $\forall\overline{\alpha_i}.\tau_f$, the macro generates fresh variables corresponding to the $\overline{\alpha_i}$, which are subsequently bound to ground types during type inference.

The generation of the macro for a type scheme requires the latter to be left linear, that is, that no variable occurs twice in the types of the parameters. In the presence of union types, nonlinearity can be a source of misconceptions. For instance, assume a function $f/2$ with type scheme $\forall\alpha.(\alpha, \alpha) \to true$. Although it seems at first sight that the definition `f(0,a) -> true` does not fit into this scheme, it actually does under the instance $[\alpha/0 \cup a]$. In fact, we can prove by using Def. 2 that this scheme is equivalent to $\forall\alpha_1, \alpha_2.(\alpha_1, \alpha_2) \to true$. In a similar way we can establish the equivalence between $\forall\alpha.(\alpha, \alpha) \to \alpha$ and $\forall\alpha.(\alpha_1, \alpha_2) \to \alpha_1 \cup \alpha_2$. If the programmer intends to convey the constraint of both parameters being equal, she would have to add the conditions $\alpha_1 \subseteq \alpha_2$ and $\alpha_2 \subseteq \alpha_1$ to the previous scheme. The resulting scheme would exclude any function f such that $\mathcal{E}[\![f(v_1, v_2)]\!] \neq \emptyset$ for some values $v_1, v_2 \in DVal$ such that there exist two types τ_1 and τ_2 that "separate" these values, that is, $v_1 \in \mathcal{T}[\![\tau_1]\!] \backslash \mathcal{T}[\![\tau_2]\!]$ and $v_2 \in \mathcal{T}[\![\tau_2]\!] \backslash \mathcal{T}[\![\tau_1]\!]$. In this case, the third condition of Def. 2 would not be satisfied, since we would get $Sol(\{\tau_1 \subseteq \tau_2, \tau_2 \subseteq \tau_1\}) = \emptyset$ but $\mathcal{E}[\![f]\!]|_{\mathcal{T}[\![(\tau_1,\tau_2)]\!]} \supseteq \mathcal{E}[\![f]\!]|_{\mathcal{E}[\![(v_1,v_2)]\!]} \supset \emptyset$. With the current type system there are values that cannot be separated by types. For instance, let us consider $fun(X) \to X + 1$ and $fun(X) \to$

$X - 1$. Each of these expressions has $(integer) \rightarrow integer$ as the smallest type containing its semantics, so the expressions cannot be separated.

In order to left-linearize a type scheme we rename each occurrence of the same variable with different type variables and substitute, in the right-hand side of the type scheme, the union of these variables for the original one.

Definition 3. *Given a type scheme* $\sigma = \forall\overline{\alpha_i}.(\overline{\tau_j}) \rightarrow \tau \mid C$*, we say that a type scheme* $\sigma' = \forall\overline{\alpha_i}'.(\overline{\tau_j}') \rightarrow \tau' \mid C'$ *is a left linearization of* σ *iff* σ' *does not contain free type variables, no type variable occurs twice in* $\overline{\tau_j}'$ *and there exists a substitution* $\pi : \{\overline{\alpha_i}'\} \rightarrow \{\overline{\alpha_i}\}$ *such that:*

1. $\tau_j'\pi = \tau_j$ *for every* j*.*
2. *If we define the substitution* $\pi_{img} = \overline{[\alpha_i / \cup_{\beta \in \pi^{-1}(\{\alpha_i\})} \beta]}$ *then* $\tau' = \tau\pi_{img}$*.*
3. $C' = \{\tau_1' \subseteq \tau_2' \mid var(\{\tau_1', \tau_2'\}) \subseteq \{\overline{\alpha_i}'\}, (\pi(\tau_1') \subseteq \pi(\tau_2')) \in C\}$*.*

The first condition of this definition requires the types of the parameters τ_j to be instantiations of their counterparts τ_j' in which type variables are replaced by type variables. The second condition states that whenever we replace a variable α by several variables $\alpha_1', \ldots, \alpha_n'$, the left linearization of σ replaces α by $\alpha_1' \cup \ldots \cup \alpha_n'$ in the result type of the function. The last condition specifies the set of constraints C' in the linearized type scheme. The constraints occurring in the original (non-linear) type scheme have to be replicated in the linear type scheme with their corresponding variables. For instance, the linearization of $\forall\alpha.(\alpha, \alpha) \rightarrow integer \mid \alpha \subseteq integer$ yields $\forall\alpha_1, \alpha_2, (\alpha_1, \alpha_2) \rightarrow integer \mid \alpha_1 \subseteq integer, \alpha_2 \subseteq integer$ as a result.

Now we show how to transform a type τ into an Erlang term or macro expansion with the same semantics. The function BT_{fun} (Fig. 3) does this transformation. It is given the simple type τ to be transformed. If τ is a type constructor applied to several arguments $\overline{\tau_i}^n$, the function receives a list of expressions $\overline{e_i}^n$ such that each e_i is an overapproximation of the semantics of τ_i. In the translation we assume function definitions such as `'ANY'`, `'NONE'`, etc. These functions are defined in [10], and their semantics are those of their corresponding type. For instance, $\mathcal{E}[\![\texttt{'ANY'}()]\!] = \mathcal{T}[\![any]\!]$. We assume an environment Λ^0 containing these auxiliary definitions. The `ALT` macro represents a nondeterministic choice between its arguments. The macro `NELIST` expands to a list of arbitrary length with elements of a given type. Finally, we have a family of macros $\{\texttt{FUN}_n\}_n i \in \mathbb{N}$ representing the functions of arity n. Each macro is parametric on the variables corresponding to the input arguments and the variable corresponding to the result. From these macro definitions we specify the translation of compound types in the right column of Fig. 3.

As we shall see later, the macro generated for a given function binds its parameters to the translation of their corresponding types. We could perform this translation directly via the BT_{fun} function, but the notion of parametricity implied by Def. 2 allows us to generate macros that reflect a given type scheme in a more accurate way. For instance, assume a function $f/1$ with type scheme $\forall\alpha.[\,] \cup nelist(\alpha) \rightarrow \alpha \cup false$. By using Def. 2 we can prove that $f([\,])$ can be

$BT_{fun}\ [\![none]\!]\ [\,] = \texttt{'NONE'}()$
$BT_{fun}\ [\![any]\!]\ [\,] = \texttt{'ANY'}()$
$BT_{fun}\ [\![atom]\!]\ [\,] = \texttt{'ATOM'}()$
$BT_{fun}\ [\![integer]\!]\ [\,] = \texttt{'INTEGER'}()$
$BT_{fun}\ [\![float]\!]\ [\,] = \texttt{'FLOAT'}()$
$BT_{fun}\ [\![pid]\!]\ [\,] = \texttt{'PID'}()$

$BT_{fun}\ [\![v]\!]\ [\,] = v$ where $v \in Val$
$BT_{fun}\ [\![\_ \cup \_]\!]\ [e_1, e_2] = \texttt{?ALT(}e_1\texttt{, }e_2\texttt{)}$
$BT_{fun}\ [\![\{\_, \overset{n}{\cdots}, \_\}]\!]\ [\overline{e_i}^n] = \{\overline{e_i}^n\}$
$BT_{fun}\ [\![nelist(\_, \_)]\!]\ [e_1, e_2] = \texttt{?NELIST(}e_1\texttt{, }e_2\texttt{)}$
$BT_{fun}\ [\![(\_, \overset{n}{\cdots}, \_) \to \_]\!]\ [\overline{e_i}^n, e] = \texttt{?FUN}_n\texttt{(}\overline{e_i}^n\texttt{, }e\texttt{)}$

Fig. 3. Translation of type constructors into expressions.

$TR_{par}\ [\![C()]\!]\ \eta\ \overline{\alpha_i}^n = \{\{\overline{fun() \to \texttt{'NONE'}()}^n\}, BT_{fun}\ [\![C]\!]\ [\,]\}$ (if $C \in \mathcal{TC}^0$)
$TR_{par}\ [\![\alpha]\!]\ \eta\ \overline{\alpha_i}^n = \{\{\overline{e_i}^n\}, \eta(\alpha)\}$ where $\forall i.e_i = \begin{cases} fun() \to \eta(\alpha) & \text{if } \alpha_i = \alpha \\ fun() \to \texttt{'NONE'}() & \text{otherwise} \end{cases}$
$TR_{par}\ [\![\tau_1 \cup \tau_2]\!]\ \eta\ \overline{\alpha_i}^n = \texttt{?ALT(}TR_{par}\ [\![\tau_1]\!]\ \eta\ \overline{\alpha_i}^n, TR_{par}\ [\![\tau_2]\!]\ \eta\ \overline{\alpha_i}^n\texttt{)}$
$TR_{par}\ [\![C(\overline{\tau_j}^m)]\!]\ \eta\ \overline{\alpha_i}^n = \{\{\bigsqcup \overline{e_{j,1}}^m, \ldots, \bigsqcup \overline{e_{j,n}}^m\}, BT_{fun}\ [\![C]\!]\ [\overline{e_j}^m]\}$
where $\forall j \in \{1..m\}.\{\{e_{j,1}, \ldots, e_{j,n}\}, e_j\} = TR_{par}\ [\![\tau_j]\!]\ \eta\ \overline{\alpha_i}^n$
if $C \in \{nelist(\_, \_), \{\_, \overset{m}{\cdots} \_\}, (\_, \overset{m-1}{\cdots} \_) \to \_\}$

$e_1 \sqcup e_2 = \begin{cases} e_2 & \text{if } e_1 = fun() \to \texttt{'NONE'}() \\ e_1 & \text{otherwise} \end{cases}$
$\bigsqcup \overline{e_i}^n = e_1 \sqcup (e_2 \sqcup ...(e_{n-1} \sqcup e_n)...)$

Fig. 4. Translation of the types of the parameters and type variable bindings

evaluated only to *false*. In fact, for every simple type τ:

$$\mathcal{E}[\![f([\,])]\!] \sqsubseteq \mathcal{E}[\![f]\!]|_{\mathcal{T}[\![[\,] \cup nelist(\tau)]\!]}\,(\mathcal{E}[\![[\,]]\!]) \sqsubseteq \mathcal{T}[\![[\,] \cup nelist(\tau) \to \tau \cup false]\!]\,(\mathcal{E}[\![[\,]]\!]) \sqsubseteq \mathcal{T}[\![\tau]\!] \cup \{false\}$$

In particular, for $\tau = 0$ and $\tau = 1$ we would obtain that $\mathcal{E}[\![f([\,])]\!]$ is a subset of both $\{0, false\}$ and $\{1, false\}$, so, if $f([\,])$ is evaluated to a value, then that value must be *false*. Thus, f has also the type scheme $\forall\alpha.([\,] \to none \cup false) \cup (nelist(\alpha) \to \alpha \cup false)$, which is equivalent to the scheme shown previously. In general, when a function expects a parameter of type τ, but the type of the actual argument does not bind some of the variables in τ, these variables are bound to *none* in f's result type. In Fig. 4 we define the TR_{par} function, which receives a simple type τ, a mapping η from type variables to program variables and a list of variables $\overline{\alpha_i}$. It returns a tuple whose second component is an Erlang term with the same semantics as τ, but replacing the type variables of the latter by program variables as specified by η. The first component of the result is another tuple with as many closures as type variables in the list $\overline{\alpha_i}$ given as third parameter. The i-th closure of the tuple will be evaluated to $\eta(\alpha_i)$ if α_i occurs free in τ, or to `'NONE'()` otherwise. We return closures instead of plain values, since a *none* value inside tuple component would make the whole tuple to have type *none*. As an example, let us assume $\eta = [\alpha/\texttt{A}]$. The result of $TR_{par}\ [\![0 \cup \alpha]\!]\ \eta\ [\alpha]$ is `?ALT({{`$fun() \to$ `'NONE'()}, 0}, {{`$fun() \to$ `A}, A})`.

In Fig. 5 we show the *GenMacro* function which, given a type scheme σ for a function f/n it returns the definition of a macro $M_{f/n}$ overapproximating its semantics. The macro receives as many parameters as the arity of f. Firstly, it

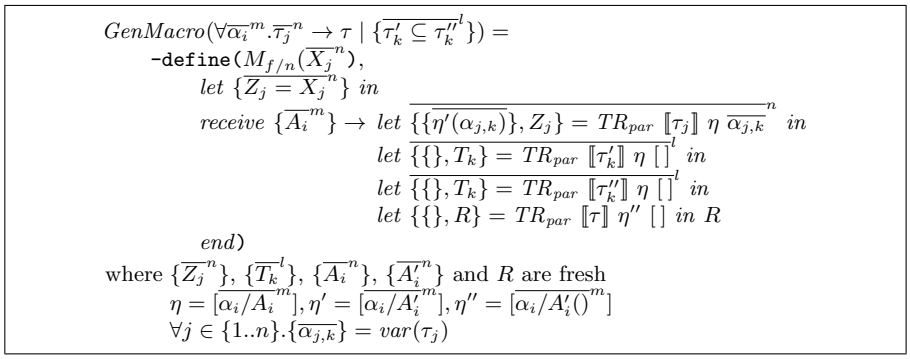

Fig. 5. Translation of a function f of a type scheme σ into a macro

assigns those parameters to fresh variables Z_j in order to avoid unnecesary code replication of the macro arguments at the macro expansion when the X_j occurs more than once in its definition. The *receive* statement brings the variables $\overline{A_i}^m$ into scope with type *any*. The types of these variables are subsequently bound to the types of the parameters Z_j by the assignments generated by TR_{par}, which also bind the $\overline{A'_i}$ variables to their corresponding closures containing either the A_i or `'NONE'()`, as explained in the previous paragraph. Then, *GenMacro* translates the constraints of the type schemes into assignments to the same fresh variable T. Notice that, assuming that $e_1 : \tau_1$ and $e_2 : \tau_2$, the sequence $T = e_1, T = e_2$ is typable if τ_1 and τ_2 are *joinable* (i.e. non disjoint). This is a less accurate, but safe, overapproximation of the $\subseteq$ relation between types. Finally, the result of the macro expansion is the type of the result of the function, in which the closures assigned to the A'_i are invoked. As an example, we consider the macro generated for the type scheme $\forall\alpha.(int, [\,] \cup nelist(\alpha, [\,])) \to [\,] \cup nelist(\alpha, [\,]))$:

```
-define(M(N, Xs), Z1 = N, Z2 = Xs,
    receive A ->
        {{}, Z1} = {{}, 'INT'()},
        {{AP}, Z2} = ?ALT({{fun 'NONE'/0}, []},
                          {{fun() -> A end}, ?NELIST(A,[])),
        ?ALT([], ?NELIST(AP(),[]))
    end).
```

The definition of Fig. 5 only covers the case in which the input scheme is not overloaded. In the case of overloaded schemes we would have to generate an auxiliary macro for each of the specifications and another one defined as the disjunction (via `?ALT`) of these auxiliary macros.

Once these macros have been generated, the transformation of the program is straightforward. Given an expression e, we denote by e^T the result of replacing each function call $f(e_1, \ldots, e_n)$ in e by the corresponding macro expansion $?M_{f/n}(e_1, \ldots, e_n)$. Aditionally, the transformed environment Λ^T of Λ is the environment resulting from the transformation of the expressions occurring in the right-hand side of the bindings in Λ plus the bindings contained within Λ^0.

The following results prove the adequacy of the transformation. The first one shows three related things: that M_σ reflects the largest semantics compatible with σ, that the transformation overapproximates the semantics of expressions and, as a consequence, that the transformation is sound for computing success types, hence for detecting failures.

Proposition 1. *Let Λ be an environment and Λ^T its transformation. Then:*

(i) *If $f/n : \sigma$, then $\mathcal{E}[\![f]\!]^\Lambda \sqsubseteq \mathcal{E}[\![fun(X_1, \ldots, X_n) \rightarrow ?M_\sigma(X_1, \ldots, X_n)]\!]^{\Lambda^0}$.*
(ii) *$\mathcal{E}[\![e]\!]^\Lambda \sqsubseteq \mathcal{E}[\![e^T]\!]^{\Lambda^T}$, for any e.*
(iii) *If $e^T : \tau$ for Λ^T then $e : \tau$ for Λ, for any e, τ.*

All this would be useless if the loss of precision of the transformation with respect to the real semantics implied also a loss of precision in Dialyzer's analysis. We shall study now under which conditions the transformed program produces less accurate results than the original one. As it was stated in Sect. 3, Dialyzer uses the compaction of the polymorphic spec given by the user. Let us denote by Λ^C the environment that results from replacing in Λ every type scheme σ by $C(\sigma)$. If Dialyzer used the environment Λ^C for inferring success types, then we would ensure that it yields the same or more accurate results when applied to the transformed program.

Proposition 2. *If Dialyzer infers $e : \tau$ in an environment Λ^C, then it infers $e^T : \tau'$ in Λ^T for some $\tau' \sqsubseteq \tau$.*

The improvement is in fact strict in many cases, as proved by the examples of *id*, *map* and all usual polymorphic functions. This proposition applies when the user has specified a spec σ whose compaction $C(\sigma)$ is equal or more accurate than the type τ inferred by Dialyzer, as it happens in the great majority of cases. If it does not, then Dialyzer uses $\tau \sqcap C(\sigma)$ in the environment that is used for analysing the calls to f in the rest of the program. This may lead to a loss of precision when applying our transformation, as the following example shows:

```
-spec f(any()) -> any().          g() -> f(1).
f(0) -> 0.
```

Dialyzer would infer the call `g()` to have type *none*, as it considers the type $((0) \rightarrow 0) \sqcap ((any) \rightarrow any) = 0 \rightarrow 0$ when analysing `f(1)`. However, our transformation replaces `f(1)` by a term `?F(1)` whose semantics is that of $(any) \rightarrow any$, so the expression `g()` is inferred with type *any*. Nevertheless, we can adapt our transformation such that it uses $GenMacro(\tau)$ instead of $GenMacro(\sigma)$ whenever $\tau \subseteq C(\sigma)$. We just would have to apply Dialyzer twice, firstly to the original program without user-given specs, and then to the transformed program.

5 Conclusions and future work

Dialyzer is a great tool for preventing statically different kinds of failures in Erlang programs; in particular, runtime reduction errors are detected when Dialyzer infers the empty type *none* as the return success type for a function.

The precision of the inference can be improved if the user provides more refined types by means of type *specs*, that can be even polymorphic and with subtyping constraints. However, polymorphism of specs is not fully exploited by Dialyzer, leading to a great loss of precision in many cases (for instance, most of the functions in the Erlang module `lists` have a polymorphic spec). Types inferred in previous proposals, like [11], were not better.

This weakness is probably not casual: as we have discussed in this paper, it is not obvious how polymorphic success types must be interpreted. Our first contribution has been a precise notion of what a polymorphic specification means, expressing the intuition, familiar to the seasoned functional programmer, that polymorphic functions are *parametric* and must not inspect argument positions corresponding to polymorphic data variables.

Our second contribution came from the observation that the content of polymorphic specs can expressed by pieces of Erlang code that, when inlined in a program replacing original function calls, force Dialyzer to really take into account the polymorphism of the function spec. This leads to a macro-based program transformation that, although losing precision from the point of view of the actual program semantics, permits Dialyzer to do a more refined analysis.

We think fair to say that our work improves significantly the present behavior of Dialyzer regarding polymorphism, and we see other positive aspects in the approach: it is lightweight, since no changes are needed in Dialyzer nor in user written programs, as far as the specifications are already in the program; it is scalable, because the macro expansions have a linear impact on the size of programs; and it is modular in the sense that only function specs are used for the transformation, so the actual definitions of function can be changed as far as specs are respected. We have implemented the transformation in an easy-to-use tool that can be found at `http://dalila.sip.ucm.es/poly_erlang`. Its source code is available at `https://github.com/manuelmontenegro/erlang-poly-transformer`. This tool runs under Dialyzer 2.7.3 with Erlang/OTP 17 (ERTS v6.13).

In this work we have assumed that the specs given by the user are correct, in the sense that they are success types schemes of the function to which they are attached, according to Def. 2. Checking that correctness of user-given specs is left to future work. We also aim to devise an inference algorithm for polymorphic specs, so that the programmer does not need to declare them.

Acknowledgements The authors would like to thank Kostis Sagonas and Stavros Aronis, for many fruitful discussions about Dialyzer and success types, that have been fundamental for developing the intuitions about the meaning of polymorphic success types that is proposed in this paper.

References

1. Erlang reference manual user's guide v 6.4: 7. types and function specifications. http://erlang.org/doc/reference_manual/typespec.html, 2015.
2. J. Armstrong. *Programming Erlang*. Pragmatic Programmers, 2013.
3. R. Carlsson. An introduction to core erlang. In *Proceedings of the PLI'01 Erlang Workshop*. Citeseer, 2001.

4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.
5. C. A. Gunter, P. D. Mosses, and D. S. Scott. Semantic domains and denotational semantics. Technical Report MS-CIS-89-16, Department of Computer and Information Science, University of Pennsylvania, February 1989.
6. M. Jimenez, T. Lindahl, and K. Sagonas. A language for specifying type contracts in erlang and its interaction with success typings. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 11–17. ACM, 2007.
7. T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Programming Languages and Systems*, pages 91–106. Springer, 2004.
8. T. Lindahl and K. Sagonas. Typer: a type annotator of erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25. ACM, 2005.
9. T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM.
10. F. J. López-Fraguas, M. Montenegro, and J. Sánchez-Hernández. Polymorphic types in Erlang function specifications (extended version). Technical Report TR-3-15, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2015.
11. S. Marlow and P. Wadler. A practical subtyping system for erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 136–149, New York, NY, USA, 1997. ACM.
12. B. C. Pierce. Programming with intersection types and bounded polymorphism. Technical report, 1991.
13. J. C. Reynolds. Types, abstraction and parametric polymorphism. 1983.
14. K. F. Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 13–18. Springer, 2010.
15. P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.