A Language for Specifying Type Contracts in Erlang and its Interaction with Success Typings

Miguel Jiménez Tobias Lindahl Konstantinos Sagonas

Department of Information Technology
Uppsala University, Sweden
migueljimg@gmail.com, {tobiasl,kostis}@it.uu.se

1. Introduction

For quite some time now, programs in ERLANG have been developed without any mention of types which describe their intended use. With the advent of automatic documentation tools such as Edoc many ERLANG programmers have discovered the usefulness of types as documentation. However, while type annotations given as comments are better than no annotations at all, they tend to rot as they are not verified. In addition, the usefulness of the type annotations is restricted to the programmer's eyes, and without a standardized type language, tools for static analysis such as Dialyzer cannot take advantage of the information.

In this work, we propose a contract language that can serve both as documentation in the style of Edoc, and as a guidance to tools such as Dialyzer and TypEr. The contracts are in the form of success typings, a framework developed for expressing type information in dynamically typed programming languages. Our contracts are designed for ease of use and clarity, but also to provide some key functionality, such as contract overloading and bounded parametric polymorphism, which can provide analyses with more refined information.

The contract language is yet another step in the authors' attempt to exploit type information in ERLANG programs and raise the type awareness of the ERLANG community. Earlier experiences with Dialyzer and TypEr have shown that there is a lot of type information already available in ERLANG code, but with the help of the programmer, more type information can be explicitly available both for the eyes of other programmers and for the benefit of type-based static analysis tools.

The remainder of the paper is structured as follows. In Section 2 we recapitulate the main ideas behind success typings and motivate why this is a useful framework for dynamically typed languages. The basic contract language is described in Section 3; its interaction with success typings is described in Section 4. In Section 5 some examples are given, followed by related work and some concluding remarks.

2. Success Typings

Using type information in dynamically typed languages is often called *soft typing*, a term coined by Cartwright and Fagan [1]. Soft typing encompasses various approaches, but commonly soft type systems use a static type domain extended with some way of expressing dynamic types, either to eliminate dynamic type tests or to find type clashes in the code. Soft type systems are by definition not allowed to reject programs, but they can bring the attention of the user to places in the code where there is a risk for a type clash.

If a soft type system reports all possible points in the code where there is a risk of a type error, we say that the reports (or warnings) are *complete*. If, on the other hand, the soft type system reports only definite type clashes we call the warnings *sound*. With these definitions, the warnings cannot be both sound and complete for a practical programming language, since this is the same problem as having a sound and complete type inference.

In dynamically typed languages type safety is guaranteed by dynamic type tests (i.e., by inspecting the type tag of values during runtime). Type reconstruction can be done with the help of available language constructs such as explicit type tests and primitive operations with known type behavior. However, the available information is often not enough to say whether there will be a type clash or not at a given program point. A soft type system that opts for complete warnings has no choice but to report the program point as a possible type error, thus giving a lot of spurious warnings.

Our experience in developing the static analysis tool Dialyzer (A Discrepancy Analyzer of ERLANG code [4, 8].) and interacting with its user community, has taught us that soundness of warnings is an important feature for such a tool from the usability point of view. By allowing programmers to see the benefits from using a type-based analysis with as little effort as possible, we can convince them to put more effort into incrementally adding more type information in the program. Without sound warnings, the benefit is typically hidden among the numerous false positives.

In prior work we have defined success typings [6], a framework for describing type information in dynamically typed programming languages. The notion of success typings accurately captures the dynamic type behavior of the ERLANG language and is the basis for type analyses which emit warnings that are sound rather than complete.

2.1 Basic idea

The key to giving sound warnings is determining when a program construct will surely fail. In some primitive operations of the language this is trivial and the corresponding type information can be hard-coded. For example consider addition in ERLANG: adding an integer to a list will definitely fail, but adding an integer to a float will probably succeed.[1] When dealing with user-defined functions the problem of automatically capturing the success and failure behavior of functions is more complex.

Consider the ERLANG implementation[2] of the Boolean `and` function shown below.

```
and(true, true) -> true;
and(false, _) -> false;
and(_, false) -> false.
```

The first clause matches if both the arguments are `true`, and the remaining clauses match if either of the arguments is `false`. Assuming we have defined the Boolean type, $bool()$, as `true | false`, we would expect a Hindley-Milner type inferencer to derive the type

$$(bool(), bool()) \rightarrow bool()$$

for this function. In the first function clause, this description is obvious, and nothing in the following clauses contradicts it. We can say for sure that if this function is applied with Booleans as arguments, we will have no type clash and we will get a Boolean as the return value. A static type checker can enforce this type signature by rejecting programs that contain calls with non-Booleans. A soft type system can give a warning based on this type signature whenever the arguments are not Booleans, but in some cases these warnings will be spurious. For example, the call

```
and(false, 3.14)
```

does not conform to the type signature, but will indeed evaluate to `false` without any type error.

In the work of Marlow and Wadler [7] a subtype domain is used. They report having problems with the `and` function and their inference finds the type

$$(any(), \texttt{false}) \rightarrow bool()$$

where $any()$ is the type that includes all ERLANG terms, and `false` is the singleton type containing only the atom `false`. In this particular case, the odd type signature is a side effect of pattern matching compilation, but it indicates a more general problem. Inferred domains for a function might be too restrictive and might not describe a function's actual behavior. In particular, they do not state when a function call will fail, but how to restrict the arguments to avoid type clashes. If the second argument in our example is restricted to `false` there will never be a type clash, but arguably this restriction does not reflect how the function can, and should, be used.

The inference of success typings takes another approach. Instead of restricting the domain to avoid type clashes, the inferred domain must include all values for which a function application can succeed, even if this means including values for which there might be a type clash.

DEFINITION 1 (Success Typing). *A success typing of a function f is a type signature, $(\bar{\alpha}) \rightarrow \beta$, such that whenever an application $f(\bar{p})$ reduces to a value v, then $v \in \beta$ and $\bar{p} \in \bar{\alpha}$.*

The key property is that the domain of a success typing expresses for which arguments an application has a chance of succeeding, with a guarantee of failure whenever the arguments are outside this domain. In other words, success typings are *sound for failure* rather than sound for type safety, a property already guaranteed by the ERLANG language through dynamic type tests.

[1] We write "probably succeed" because in ERLANG the addition will result in a `badarith` exception if the result is bigger than the maximum value that can be represented as a float.

[2] The example is taken from Marlow and Wadler's work on a subtype system for ERLANG [7, Section 9.3].

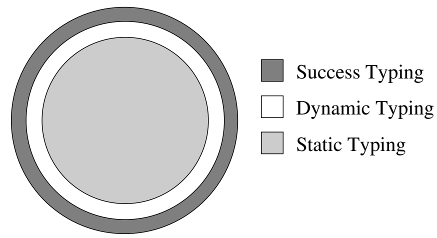

Figure 1. An illustration of function domains

In Figure 1 there is an illustration of inclusion of function domains in different frameworks. The dynamic typing domain is the domain for which a function will evaluate without type clashes in a dynamically typed language. This is in some sense the ideal description of the function, since it is not restricted by the static type system nor over-approximated due to analysis imprecision. The static typing domain for the function will always be a subset of the dynamic typing domain. If the static types have the principal type property, the static typing domain will be as large as possible and sometimes will coincide with the dynamic typing domain. In general this is not the case, and the area between the two domains consists of the arguments that will be disallowed by a static type checker, although the function call would evaluate without a type clash. The success typing domain will always be a superset of the dynamic typing domain. The ultimate aim of any inference algorithm should be to make these domains coincide.

A formal description of an automatic inference algorithm for success typings is given in [6], but basically the algorithm relies on the fact that there is a trivial success typing for all functions, namely the type signature that accepts any input and returns any value. For example, $(any()) \rightarrow any()$ is a success typing for all functions of arity one. The analysis then tries to limit the domain and range of this signature until it can no longer do so without excluding values for which the function could possibly succeed.

The `and` function has the success typing:

$$(any(), any()) \rightarrow bool()$$

This type might seem unnecessarily general. However, first note that it clearly is a success typing for the function. Secondly, note that in the absence of information about the uses of the function we cannot restrict any of function arguments in any way. Using a type domain that is more expressive (e.g., with intersection or with dependent types) we could possibly get a more precise description of the function's type, but can we find a better description without altering the type domain? In many cases, we can answer this question positively using the notion of *refined success typings*.

2.2 Refined success typings

In every program, there is a finite number of call sites for each function and assume that the analysis has knowledge about all these call sites. Also suppose that the analysis can find that the function is only called with inputs of some type(s). For example, suppose that the analysis determines that our `and` function is only called with Booleans. This situation does not change the correctness of the success typing, but it shows that the user intended to use the function in a certain way. We would like the inferred type signature to reflect not only the most general use of the function, but also how the function is actually used in a program.

DEFINITION 2 (Refined Success Typing). *Let f be a function with success typing $(\bar{\alpha}) \rightarrow \beta$. A refined success typing for f is a typing of the form $(\bar{\alpha}') \rightarrow \beta'$ such that*

1. $\bar{\alpha}' \subseteq \bar{\alpha}$ *and* $\beta' \subseteq \beta$*, and*

2. for all $\bar{p} \in \bar{\alpha}'$ *for which the application* $f(\bar{p})$ *reduces to a value,* $f(\bar{p}) \in \beta'$.

A refined success typing is a success typing with some additional constraints on the function's domain. Note that there is nothing in the definition that states where these constraints come from. In [6] the success typings are refined by a dataflow analysis that finds what domains a function is applied to. The result is function descriptions that not only describe how a function *could* be used, but also capture the actual uses of functions. The next logical step is to let the programmer state how the function is supposed to be used, something that fits nicely into the framework of refined success typings.

3. A Contract Language

A contract is a way for the programmer to explicitly state the intended uses of functions. In the general case, the success typing of a function over-approximates the types of its intended uses and it can be refined by taking the contracts into account. The basic idea is to infer the types of a function by using some inference algorithm for success typings, and then check if the success typing is compatible with the contract. If the success typing and the contract do not contradict each other, a refined success typing can be constructed based on both the information in the contract and the inferred success typing. As we will see, the resulting refined success typing can be more expressive than the one we can infer with the algorithm for automatic inference of (refined) success typings.

When encountering a function call, the types of the arguments are checked against the contract. If a violation is found this is reported, otherwise the contract is used to refine the type information at the call site. By using this approach, we gain precision in the inference, while preserving soundness of failure under the side condition that the user respects the contracts which have been specified. The contract checking follows the same approach as the rest of the inference with respect to soundness for failure, i.e., soundness for contract violations. Only when a contract cannot possibly hold, a contract violation is reported.

3.1 Basic syntax of contracts with types

Contracts in a module are given as compiler attributes. The basic contract specification follows the syntax:

$$\texttt{-spec(}F\texttt{/}A\texttt{::((}a_1, \ldots, a_n\texttt{) -> }r\texttt{)).}$$

where F is a function name, A is its arity, $a_1, \ldots, a_n$ is a possibly empty sequence of type expressions for the function's arguments and r is the type expression for the function's range.

The language for type expressions is an extension of the type language defined and used by the TypEr tool [5] and is similar to the language also used by Edoc. We briefly describe its syntax.

Type expressions are built from basic components which can be partitioned into four main groups:

- The first group consists of type expressions denoting *singleton types*. Examples of singleton types are: the atom `true`, the integer 42, the empty list `[]`, etc.
- The second group consists of a predefined set of type names (e.g., $atom()$, $integer()$, $float()$, $binary()$, $list()$, $tuple()$, $pid()$, $port()$, $ref()$, ...) for all different kinds of ERLANG terms. Integers get a special treatment and there exists a long list of predefined subtypes of integers (e.g. $byte()$, $char()$, $pos\_integer()$, $non\_neg\_integer()$, ...) and a notation for integer ranges of the form $(L..U)$ where L and U are integers representing the lower and upper bound of the range. Also, the complex types often include type expressions as arguments, in which case they contain these types in parentheses. For example, $list(integer())$ denotes the type expression for lists containing integers and for convenience this type expression can also be written as $[integer()]$. A special notation for tuples is also available; for example, $\{atom(), integer()\}$ denotes pairs (i.e., 2-tuples) whose first element is an atom and whose second element is an integer.
- The third group consists of types that are defined and given names by the user. We will soon show an example of how types are declared in our language, but we mention that their names always start with an atom followed by parentheses. The parentheses are needed in order to distinguish a type from a plain ERLANG atom, since as mentioned the type language also accepts atom names as singleton types.
- Finally, a type variable is also a type expression. Type variables are used for parametric polymorphism as described in Section 3.4 below. We have closely followed the ERLANG convention for variables and thus type variables always begin with a capital letter.

The union of any two type expressions t_1 and t_2 (written as $t_1 \mid t_2$) is also a type expression. An example of such an expression is `0 | 42` which denotes the type consisting of only the integers 0 and 42. Naturally, unions can appear anywhere where a type expression can be used. For example, an heterogeneous list consisting of integers and atoms can be defined with the type expression $[atom() \mid integer()]$.

Using type expressions containing unions, the user can define new types such as the ones below:

```
-type(fruit() :: apple | orange | banana).
-type(my_list() :: [atom() | integer()]).
```

Both examples define names, namely `fruit` and `my_list`, which exist only as aliases for more complex type expressions.

The union of all terms, whether built-in or user-defined, is the universal type which is denoted by $any()$. Also, the type language allows for the empty set of terms, denoted by the type $none()$. This type is typically not used by the user but is needed for the type lattice and in order to denote the presence of a type error.

The type system also includes *funs*, i.e., functions with either a known or an unknown number of arguments. If the number of arguments is known then these arguments are denoted as $(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are their respective type expressions. If the number of arguments is unknown but it is known that the fun's return type is described by the type expression t, then the fun is denoted by $(\ldots) \rightarrow t$. Note that t can also be the type expression $any()$.

For user convenience and for documentation purposes, the notation for records has also been extended to allow for record fields which contain type information. In other words, the user can define a record such as:

$$\texttt{-record(employee, \{name::}atom()\texttt{, age::}integer()\texttt{\}).}$$

and she can subsequently refer to this record in another record definition or in a contract specification using the notation `#employee{}`, which in turn is syntactic sugar for the type expression $\{\texttt{employee}, atom(), intege$

Type aliases can also be used in record definitions and vice versa. The only restriction is that the alias or record to be used must have been previously declared.

Optionally, the user can also give names to type expressions:

$$Name :: T$$

Currently, these names are only used for documentation purposes, i.e., they are treated as comments and are essentially ignored. How-

[3] As a matter of fact, this particular change is orthogonal to the subject of this paper and is already present in Erlang/OTP R11B-4.

ever, they can serve as a link between the language we describe in this paper and the one used by the Edoc tool which automatically creates documentation based on information given in comments. Quite often, the information supplied to Edoc is similar and contains names for variables and function arguments.

Table 1 shows a list of commonly used predefined shorthands.

Shorthand	Type Alias for
_	$any()$
$bool()$	('true' | 'false')
$number()$	$(integer() \mid float())$
$byte()$	(0..255)
$non\_neg\_integer()$	(0..)
$pos\_integer()$	(1..)
$identifier()$	$(pid() \mid port() \mid ref())$
$[atom()]$	$list(atom())$
$function()$	$(\ldots) \rightarrow any()$
$string()$	$[char()]$
$nonempty\_string()$	$[char(), ...]$

Table 1. Common type aliases

3.2 Example uses

Consider the factorial function shown below:

```
fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

We can write the following contract specification:

```
-spec(fac/1 :: ((byte()) -> integer())).
```

If we want to add further comments for documentation purposes, we can use variable names. A variant of the later contract that will have the same result is:

```
-spec(fac/1 :: ((Factor :: byte()) -> integer())).
```

Taking up again the *a*nd function described in section 2.1, we can think of a more suitable type signature that accepts only booleans as types for the arguments and range. The contract will be:

```
-spec(and/2 :: ((bool(),bool()) -> bool())).
```

But the use of the function can also be extended to accept any atom by writing:

```
-spec(and/2 :: ((atom(),atom()) -> bool())).
```

Despite being quite strange, this contract is valid and reflects one of the possible uses of the function. As we will see in section 4, we have a wide range of freedom to adjust the behaviour of a certain function to fit our needs.

The `nth/2` function of the `lists` module returns the element which is contained in the nth position of a list. Suppose that we are using this function in a module but we want it to work only for lists of atoms. We also know that the length of the lists will never be over a certain threshold, for example, 10. We can add these new constraints to the analysis by writing the contract:

```
-spec(nth/2 :: ((1..10, [atom()]) -> atom())).
```

A warning will be emitted if a call is statically found to not fulfill all the constraints.

3.3 Contract overloading

In ERLANG, functions can be defined to operate on different types in an overloaded fashion. In order to capture this, contracts are allowed to be overloaded as well. For example, consider the function `inc/1` in Figure 2. Its two clauses are written to operate on integers

```
-spec(inc/1 :: ((integer()) -> integer());
               ((float()) -> float())).

inc(X) when is_integer(X) ->
  X + 1;
inc(X) when is_float(X) ->
  X + 1.0.
```

Figure 2. An overloaded increment function

and floats respectively, adding one to the input argument.[4] The success typing for this function is $(number()) \rightarrow number()$, losing the information about overloading, and abstracting to a supertype. By specifying an overloaded contract the underlying type information is kept. Overloaded contracts are specified as a list of simple contracts separated by semicolons. In Figure 2 an overloaded contract is specified to allow calls to `inc/1` with $float()$ or $integer()$ to return $float()$ or $integer()$ respectively.

The language accepts overloading contracts with overlapping domains, i.e., a domain for a certain contract can be subset of another contract's domain within the same specification. Domains can even be exactly the same for two (or more) overloading contracts. This is also allowed for contract ranges. The interaction between overloading contracts and the success typings is discussed on section 4.1

3.4 Polymorphism and bounded quantification

One feature of the contract language is support for parametric polymorphism. As an example where this can be useful, consider the higher order library function `lists:map/2`, which applies a function to each element of a given list and returns the resulting list. The success typing for this function is:

$$(((any()) \rightarrow any()), [any()]) \rightarrow [any()]$$

We can connect the types of the function with those of the lists by specifying the contract:

```
-spec(map/2 :: ((((A) -> B), [A]) -> [B])).
```

where A and B are universally quantified variables. The interpretation of the type variables will be further discussed in Section 4.1.

In addition, type variables can be bounded by subtype constraints by adding a guard-like constraint to the contract. For example, the increment function in Figure 2 could be assigned the contract

```
-spec(inc/1 :: ((X) -> X)
                 when is_subtype(X, number())).
```

where `is_subtype/1` is a new guard function which succeeds when the type variable in its first argument is a subtype of the type expression of its second argument.

These type variable constraints can also be combined with contract overloading. The scope of a type variable lies within a single overloading contract. For example, in this specification:

```
-spec(foo/1 ::((atom(), X) -> X)
                 when is_subtype(X, integer()));
               ((string(), X) -> X)
                 when is_subtype(X, float()))).
```

type variables in each contract are different. The first one is bounded to integers and the second one, to floats.

[4] Note that this could have been written in one clause since addition is overloaded in ERLANG.

4. Interaction with Success Typings

Contracts can be used to guide the refinement of success typings. By taking the user-defined contracts into account in the type inference, the type information can be significantly improved. However, care must be taken so that wrongly specified contracts do not make the information less precise or even false. In general, the contracts cannot be soundly verified, since this is the same problem as having a sound type checker for a dynamically typed language such as ERLANG. However, contracts allow for a more refined analysis and for reporting interface violations when these occur.

A contract can be interpreted as a set of constraints on the behavior of a function and more specifically on the set of terms which are allowed for arguments and returned as result. These type constraints can be both over-approximating and constraining depending on the purpose of the contract. Sometimes it may be convenient to abstract for readability, and other times the programmer may want to specify how a function should be used rather than how it could be used. The success typing for the function is an upper bound of the actual behavior, so a contract cannot be allowed to be in contradiction with the success typing. In order to get as precise information as possible in the analysis, both the success typing and the contract are used to find a refined success typing that is then used in the remainder of the analysis.

Assume that a function has the success typing Sig_t and the contract signature Sig_c. Success typings are covariant in the domain and range (e.g, the most general success typing of arity one is $(any()) \rightarrow any()$), which means that the subtype relation, $\subseteq$, on success typings is defined covariantly.[5] Furthermore, the infimum operator, $\cap$, is also covariant on function types.[6] When comparing the contract and the success typing we have the following four situations:

$$Sig_c \subseteq Sig_t \tag{1}$$

$$Sig_t \subseteq Sig_c \tag{2}$$

$$Sig_c \cap Sig_t \neq none() \tag{3}$$

$$Sig_c \cap Sig_t = none() \tag{4}$$

In case (1) the contract is constraining the function more than the success typing, but does not contradict it. In case (2) the contract is over-approximating the behavior of the function, which is not in conflict with the success typing. In both cases, the resulting refined success typing is simply the infimum of the contract and the success typing since we are interested in the most specific description. In case (3) the contract and the success typing are incomparable, but there is a common description of the type behavior, so this case can be viewed as a combination of the two former cases. In some aspects the contract is refining the success typing and in some aspects it is making it more general. The refined success typing is once again the infimum of the contract and the success typing. In case (4) there is no common description of the type behavior of the function. This is clearly a violation of the contract and the user should be warned. Following the principle of soundness for failure, this is also the only case where the user will be warned about the contract validation.

A contract must be respected not only by the function for which it is declared, but also by the users of the function. As explained in more detail in [6], the success typing domain is used as an upper bound of the argument types of a call site. Since the contract domain is also an upper bound, the constraints must be used in conjunction, effectively forming the infimum of the two domains. The ranges are treated analogously. If we find that the arguments cannot satisfy the constraints we consider this as a contract violation at the call site. Likewise, if the caller fails to handle the return type, the contract violation is at the call site, even though it might have been the contract that was malformed. In general, if a contract cannot be disproved at the declaration point, it is trusted and all violations are considered to be the fault of the callers.

4.1 Issues with overloading and type variables

Adding the expressibility of overloading and bounded quantification to the contract language does not cause any considerable overhead in the analysis. One might fear that expressibility adds complexity, and this is of course true in the general case, but since the contracts in this work are verified on a best-effort basis, where contracts are only rejected if they are proved to be false, the extra effort is reasonably small. However, there are some issues.

When faced with an overloaded contract, the type inference gains most information when the domains of the different parts of the contract are disjoint. However, if this is not the case, or if the information about the applied arguments is not specific enough to choose which overloaded part to consider, the union of the overloaded parts that can match the arguments can be used. Note that this corresponds to assigning a larger domain and range to a success typing, so this does not break any assumptions. For example, the overloaded contract in Figure 2 can be collapsed to $(number()) \rightarrow number()$ if the analysis cannot find which of the overloaded clauses is used at a certain call site.

Determining how to instantiate type variables in our type domain is problematic, and we do not claim to have found the best solution. However, while any analysis that take the type variables into account must take care not to surprise the user with unpredicted results, it is clearly useful to have the possibility to express parametric polymorphism in the contracts. For documentation purposes if not for anything else.

The main problem with instantiation is that our type domain includes constructor-free unions. Since types can be part of any union (that can also include any singleton type) we have an infinite number of types that any ERLANG term can belong to. For example, the integer 42 belongs the type $integer()$, but also to the union types $integer()|atom()$, $number()|tuple()$ and `42|77`.

As an example of a polymorphic contract, consider the following contract for the increment function in Figure 2.

```
-spec(inc/1 :: ((X) -> X)
        when is_subtype(X, number())).
```

In this case $number()$ is an upper bound on X. Suppose there is a call to the increment function with the constant 42. The operations in the function is not closed on the singleton type 42, so this instantiation cannot be made. This is common for the singleton types, so one solution is to exclude the singleton types from the type domain when instantiating type variables. Singleton integers are widened to $integer()$, and singleton atoms are widened to $atom()$. Unions and structured types are also transformed by widening the subparts that are singleton. For example, $1|23$ is widened to $integer()$ and $\{foo\}$ is widened to $\{atom()\}$.

This somewhat ad hoc solution removes a lot of the surprising results, but it also makes the analysis lose precision. For example, the built-in head function, `hd/1`, that returns the head of a list can be described with the contract

```
-spec(hd/1 :: (([X,...]) -> X)).
```

When instantiated on a call site with the argument type $[1|2,...]$, the return becomes $integer()$. This is a loss of precision, but it is still better than the success typing of this function which is $([any(),...]) \rightarrow any()$.

[5] $(\alpha) \rightarrow \beta \subseteq (\alpha') \rightarrow \beta' \iff \alpha \subseteq \alpha' \wedge \beta \subseteq \beta'$

[6] $(\alpha) \rightarrow \beta \cap (\alpha') \rightarrow \beta'$
$$= \begin{cases} (\alpha \cap \alpha') \rightarrow \beta \cap \beta' & \text{when } \alpha \cap \alpha', \beta \cap \beta' \neq none() \\ none() & \text{otherwise} \end{cases}$$

When there is more than one type variable in the arguments of a contract, the variable is instantiated to the least upper bound of the argument types at the call site. For example, if the contract for some function `foo/2` is

```
-spec(foo/2 :: ((X, X) -> X)).
```

there is no bound on what types the variable X can represent. Essentially, the contract gives us little more information than the success typing $(any(), any()) \to any()$, but if we view the contract as pre- and postconditions, we can interpret the intention of the user as *"Whatever I give in the arguments should also be true for the return of the function."*. Under this interpretation, if there is a call site with the argument types $integer()$ and $atom()$, the type variable X is instantiated to $integer()|atom()$, which is also the return type of the call site.

We are exploring different ways of limiting the types that a type variable can be instantiated to, such as disallowing type unions completely, only allow unions if they are declared as a named type, or explicitly enumerating the types that a variable can be instantiated to. In general, such limitations can go into the contracts as side conditions in the same manner as the `is_subtype` constraint. We choose not to elaborate further at this point, and leave this as future work.

5. Two Examples

A commonly used function from the `lists` module in the standard library is `append/2`, whose intended use is for list concatenation. For efficiency reasons this function is actually implemented in C, but we can consider that its implementation is as follows:

```
append([], L) -> L;
append([H1|L1], L2) ->
  [H1|append(L1,L2)].
```

The problem is that, with an implementation such as the one above, the function's inferred success typing is: $([any()], any()) \to any()$. Indeed, in a language like ERLANG and with a type system like the one we are using (i.e., without intersection types), this success typing accurately captures the operational behavior of this function. Notice that the call `append([],3.14)`, however unintended, will match the first clause and succeed with `3.14` as result. We can make this function reflect its intended uses by defining a suitable contract for it:

```
-spec(append/2 :: (([T],[T]) -> [T])).
```

This would constraint the uses of this function and will flag calls like `append([],4.5)` or even `append([1,2],[3|4])` as violating the contract. Notice however, that the `append([1,2],[a,b])` call will *not* be flagged as violating the contract since it is actually possible for T to be the type expression $atom() \ | \ integer()$.

Another commonly used function from the `lists` module is the function `all/2`. It is defined as follows:

```
all(Pred, [Hd|Tail]) ->
  case Pred(Hd) of
    true -> all(Pred, Tail);
    false -> false
  end;
all(Pred, []) when is_function(Pred, 1) -> true.
```

The success typing which is inferred for this function is:

$$((any()) \to any(), possibly\_improper\_list(any())) \to bool()$$

At first sight this success typing might seem a bit counter-intuitive, and possibly even incorrect. We will argue that from the point of view of capturing all possible uses of this function, no matter how unintended they might be, it is actually the best we can do.

First of all, we infer that the function can accept a possibly improper list in its second argument because the function is short-circuiting. Indeed, the call `all(fun is_atom/1, [42|gazonk])` will evaluate without any type clash and will return `false`. The reason for the inferred type of the first argument is more subtle. Note that the `case` expression in the first clause can succeed not only when the `Pred` function returns the atoms `true` or `false`, but also for a function that returns these two atoms and even more, provided of course it happens to return `true` (and possibly `false`) for the elements of the list in `all`'s second argument. Since there is no upper limit in what the `Pred` function can return, the only reasonable type that we can infer for its range is $any()$.

Using a polymorphic contract like the one below we can restrict its uses to those which programmers used to statically typed languages would find most natural for this function.

```
-spec(all/2 :: (((T) -> bool(), [T]) -> bool())).
```

Of course, more liberal contracts are also possible. Two different ones are shown below.

```
-spec(all/2 :: (((_) -> bool(), list()) -> bool())).
-spec(all/2 :: (((T) -> bool(),
                possibly_improper_list(T)) -> bool())).
```

6. Related Work

WE ARE NOT (RE-)INVENTING THE WHEEL. WE ARE EXPLORING SOMETHING WHICH HAS ALSO BEEN EXPLORED IN OTHER "SIMILAR" CONTEXTS IN THE HOPE THAT IT WILL PROVE ITSELF USEFUL IN ERLANG. BECAUSE IN ERLANG THE PROCESS IS ALREADY SEMI-AUTOMATED, DUE TO THE EXISTENCE OF THE TYPER TOOL WHICH AUTOMATICALLY ANNOTATES PROGRAMS WITH TYPE INFORMATION, PERHAPS MORE SO?

THE FINAL VERSION OF THE PAPER WILL CONTAIN A RELATED WORK SECTION WHERE WE WILL REVIEW AND CONTRAST OUR WORK WITH THE FOLLOWING:

- FROM THE FIELD OF FUNCTIONAL PROGRAMMING: WITH SIMILAR PROPOSALS FOR LISP AND SCHEME AND IN PARTICULAR WITH THE ANNOTATION AND CONTRACT LANGUAGE OF THE DRSCHEME SYSTEM [2].
- FROM THE FIELD OF LOGIC PROGRAMMING: WITH THE TYPE LANGUAGE OF MERCURY [9] AND THE ANNOTATION LANGUAGE OF CIAO PROLOG [3]
- FROM ERLANG ITSELF: WITH DIALYZER [4], TYPER [5] AND THE TYPE LANGUAGE CURRENTLY USED BY EDOC

PLUS ANYTHING ELSE BROUGHT TO OUR ATTENTION BY THE REVIEWERS. SUGGESTIONS WELCOME!

7. Concluding Remarks and Future Work

We have described a language for specifying user-defined types in ERLANG and for annotating functions with contracts containing type information. These contracts document the intended uses of functions, but they can also be combined with success typings and help defect detection tools such as Dialyzer to detect type clashes in ERLANG programs. We have presented some simple examples of possible contracts for commonly used functions and described issues related to annotating libraries with such contract information.

The language we have described in this paper is already implemented in the development version of Erlang/OTP R12. For its actual use, the next step is to annotate standard libraries with contract information, a tedious and occasionally not totally straightforward job. Doing so, might possibly reveal cases for which the contract language is not expressive enough and needs to be extended, but we strongly believe that the basic machinery is the one we have described.

Eventually, it is up to the user community to decide whether contracts containing type information is a good idea in languages such as ERLANG or not. But we have good reasons to believe that our proposal will not remain unexplored or just a paper design.

References

[1] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–292. ACM Press, 1991.

[2] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, Mar. 2002.

[3] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Programming*, 58(1-2):115–140, 2005.

[4] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[5] T. Lindahl and K. Sagonas. Typer: a type annotator of erlang code. In *Proceedings of the 2005 ACM SIGPLAN Erlang Workshop*, pages 17–25, New York, NY, USA, 2005. ACM Press.

[6] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.

[7] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–149. ACM Press, June 1997.

[8] K. Sagonas. Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications. In *ACM SIGPLAN Workshop on the Evaluation of Defect Detection Tools (Bugs'05)*, June 2005.

[9] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1–3):17–64, Oct./Dec. 1996.