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

Miguel Jiménez[1], **Tobias Lindahl**[1,2], Konstantinos Sagonas[3,1]

[1] Department of Information Technology, Uppsala University, Sweden

[2] Ericsson AB, Sweden

[3] School of Electrical and Computer Engineering,

National Technical University of Athens, Greece

UPPSALA UNIVERSITET

ERICSSON TAKING YOU FORWARD

Erlang and Types

- Types are anonymous but important in Erlang
  - Erlang is strongly typed
  - The compiler uses types for optimizations
    - Pattern matching, removing type checks, ...
- Types have gained importance for users
  - Used as documentation
  - Dialyzer uses types for defect detection
  - Refactoring tools can use (are using?) type information

We believe there is more to gain by exposing types to the programmer!

Types for Documentation

- Type signatures are used in the Erlang documentation
- Projects use type signatures in comments (edoc, or home-brewed)
- Comments have a tendancy to rot if not checked
- Tools need to parse comments to get the information

Define a contract/type language that:

1. Has a defined syntax and meaning
2. Is parsed and stored in the beam file

UPPSALA UNIVERSITET

ERICSSON TAKING YOU FORWARD

An Informal Specification for append

Consider the following implementation of append

```
%% @spec append([any()],[any()])->[any()]
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

Two interpretations of the type signature:

1. Append can only take two lists and return a list.
2. Append will return a list if given two lists, otherwise the behaviour is undefined.

An Informal Specification for append

Consider the following implementation of append

```
%% @spec append([any()],[any()])->[any()]
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

Two interpretations of the type signature:

1. Append can only take two lists and return a list. FALSE
2. Append will return a list if given two lists, otherwise the behaviour is undefined.

An Informal Specification for append

Consider the following implementation of append

```
%% @spec append([any()],[any()])->[any()]
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

Two interpretations of the type signature:

1. Append can only take two lists and return a list. FALSE
2. Append will return a list if given two lists, otherwise the behaviour is undefined. LIMITED USE FOR ANALYSES

A Slightly More Formal Specification

... that might not be so different syntactically

```
-spec(append/2::([any()],[any()])->[any()]).
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

The interpretation of the new specification:

> Append will return a list if given two lists *and should not be used in any other way*

Differences:

- The specification is a contract
- The specification is an attribute rather than a comment

Some Benefits of Contracts

- Static analysis
  - Dialyzer
  - SOMETOOLNAME
- Testing
  - Instrument the code to log contract violations
  - Test case generation
- Documentation
  - The meaning of specifications become more clear
  - Edoc
- ...

The Type Domain

- All Erlang terms belong to the universal type *any()*
- However, some operations are only defined on subtypes of this type
- Basic types
  - *integer()*
  - *atom()*
  - *pid()*
  - ...
- Type unions
  - *atom()* | *tuple()*
  - *integer()* | *float()* (syntactic sugar: *number()*

The Type Domain (cont)

- Structured types
  - Lists: *[atom()]*, *[integer()]*
  - Tuples: *{atom(), integer()}*
  - Funs: *fun((tuple()) -> integer())*
- Some basic types have more precision (subtypes)
  - Atoms: *bool()*, *'foo'*, *'bar'*
  - Integers: *pos_integer()*, *byte()*, *1..42*, *42*
  - Tuples: *{'ok', integer()} | {'error', string()}*
  - Lists: *[]*, *[atom(),...]*

Basic Contracts

A contract for a function is given using the attribute `-spec`

```
-spec(length/1::([any()])->non_neg_integer()).
```

Optionally, the module can be given

```
-spec(my_lists:length/1::([any()])->non_neg_integer()).
```

Also optionally, the name of arguments can be given

```
-spec(length/1::(MyList::[any()])->non_neg_integer()).
```

Overloaded and Parametric Contracts

When a function have an overloaded behavior, this can be specified by using multiple clauses.

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

Types can be parametrized with type variables. Type variables have the same syntax as Erlang variables.

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

It is also possible to put constraints on type variables

```
-spec(my_hd/1::([X,...])->X when is_atom(X)).
```

Type Declarations

Type aliases can be declared using the attribute `-type`

```
-type(int_list() ::  [integer()]).
```

Type aliases can also be recursive

```
-type(tree(X) ::  {X,tree(X),tree(X)} | nil).
```

Types can also be declared in record declarations

```
-record(foo, {bar::integer(),
              baz::atom()}).
```

Records can be used as types in contracts

```
-spec(update_bar ::  (#foo{},integer())->#foo{}).
```

Dialyzer and Contracts

Dialyzer - A Discrepancy Analyzer of Erlang Programs.

- Finds software defects (discrepancies) in Erlang code.
- Warnings are sound (never wrong).
- Dialyzer's analysis is based on *Success Typings*
- Contracts can help refine the information.
- Dialyzer can find if a contract for a function describes the implementation.

Success Typings

Definition:

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

Intuition:

If the arguments of an application are in the function domain, the application *might succeed*, but if they are not the application will *definitely fail*.

A Success Typing for append

Using success typings we get a description that is closer to the truth in Erlang

```
-spec(append/2::([any()],any())->any())
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

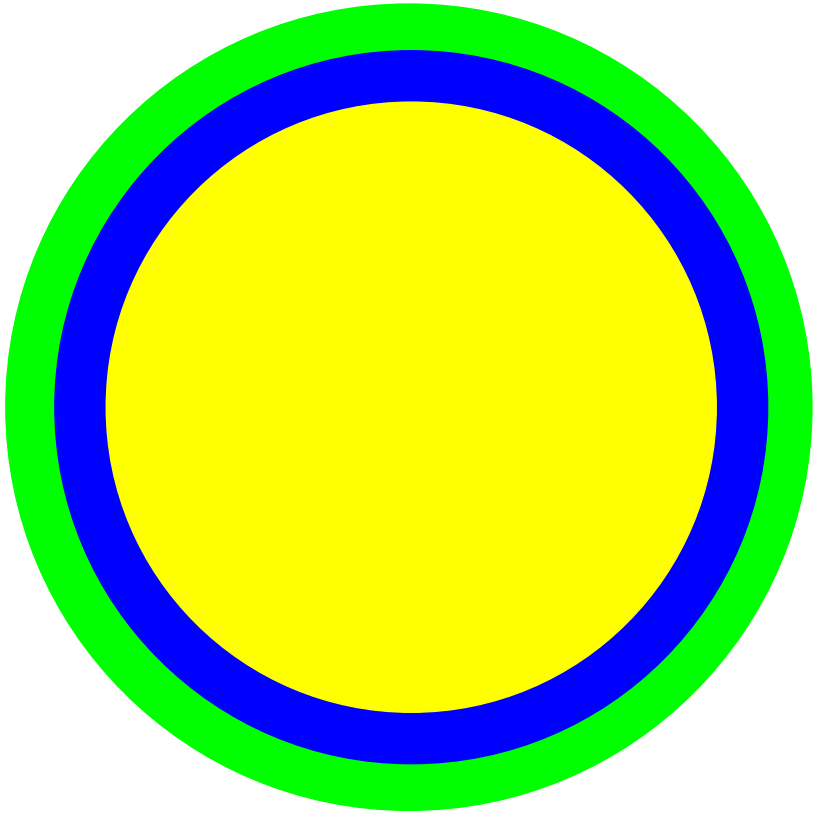
A Success Typing for append

Using success typings we get a description that is closer to the truth in Erlang

```
-spec(append/2::([any()],any())->any())
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

Success Typings are sound for failure

Function Domains



Static type domain

Dynamic type domain

Success typing domain

A Contract for append

We wanted to specify that append can only be used with lists.

```
-spec(append/2::([any()],[any()])->[any()])
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

This specification is not a success typing

Refined Success Typings

Definition:

Let f be a function with success typing $(\vec{\alpha}) \rightarrow \beta$. A refined success typing for f is a typing on the form, $(\vec{\alpha}') \rightarrow \beta'$, such that

1. $\vec{\alpha}' \subseteq \vec{\alpha}$ and $\beta' \subseteq \beta$.
2. For all $\vec{p} \in \vec{\alpha}'$ for which the application $f(\vec{p})$ reduces to a value, $f(\vec{p}) \in \beta'$.

Intuition:

A refined success typing is also a success typing, but for one reason or another, we have constrained the domain and can thus possibly constrain the range as well.

Contracts as Success Typings

Our contract is a refined success typing for append

```
%% Success Typing:  ([any()],any())->any()
-spec(append/2 ::  ([any()],[any()])->[any()]).
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

Contracts as Success Typings

Our contract is a refined success typing for append

```
%% Success Typing:  ([any()],any())->any()
-spec(append/2 ::  ([any()],[any()])->[any()]).
append([], List) -> List;
append([H|T], List) -> [H|append(T, List)].
```

Basic intuition about contract violations

- C is an instance of ST OK
- ST is an instance of C OK
- C and ST are overlapping OK
- C and ST are not overlapping NOT OK

Checking Contracts with Dialyzer

Some observations:

- Dialyzer warnings should be sound. Only warn when:
  - The contract is not possible for the function
  - The arguments at call sites violates the contract
- However, making mistakes is easy
  - Dialyzer can warn about other contract discrepancies if asked to.
- Dialyzer is not a type checker
  - You only know that Dialyzer cannot find a violation.

Concluding Remarks

Contracts

- Expose intentions with code
- Can be checked
- Serves as documentation

Current and future work

- Erlang Extension Proposal (EEP)
- Contracts for Erlang/OTP libraries
- Tools:
  - TypEr
  - Dialyzer
  - Dynamic contract checking

UPPSALA UNIVERSITET

ERICSSON TAKING YOU FORWARD