Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Τεχνικές βελτίωσης της αποτελεσματικότητας εύρεσης λαθών σε προγράμματα μέσω στατικής ανάλυσης

Διπλωματική Εργασία

του

Σταύρου Αρώνη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιανουάριος 2011

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Τεχνικές βελτίωσης της αποτελεσματικότητας εύρεσης λαθών σε προγράμματα μέσω στατικής ανάλυσης

Διπλωματική Εργασία

του

Σταύρου Αρώνη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7^{η} Ιανουαρίου, 2011.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2011

....
Σταύρος Αρώνης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η ανίχνευση και διόρθωση λαθών σε προγράμματα είναι μια διαδικασία που καταναλώνει σημαντικό μέρος του χρόνου κάθε προγραμματιστή. Εργαλεία που διευκολύνουν τον εντοπισμό λαθών είναι χρήσιμα τόσο στον περιορισμό των τελικών λαθών, όσο και στην διαδικασία εντοπισμού και διόρθωσης κατά την ανάπτυξη του προγράμματος. Στην παρουσα εργασία παρουσιάζεται η επέκταση των δυνατότητων του Dialyzer, ενός προγράμματος που χρησιμοποιεί στατική ανάλυση για την ανίχνευση λαθών σε προγράμματα στη γλώσσα Erlang, με την εισαγωγή *τύπων τομής* που βελτιώνουν σημαντικά την ακρίβεια του εργαλείου και την προσθήκη επιπλέον δυνατοτήτων σχετικών με τον εντοπισμό λαθών στην χρήση των *behaviours*, που αντιστοιχούν στις *αφηρημένες κλάσεις* των αντικειμενοστρεφών γλωσσών προγραμματισμού. Οι επεκτάσεις αυτές οδήγησαν στον εντοπισμό σημαντικών λαθών σε ήδη υπάρχοντα και ενδελεχώς ελεγμένο κώδικα.

Λέξεις Κλειδιά

Στατική ανάλυση, Συμπερασμός τύπων, Τύποι τομής, Αφηρημένες κλάσεις, Erlang, Dialyzer

Abstract

Detection and correction of bugs consumes a significant amount of every developer's time. Any tool designed with the intention to make this task easier is useful to both minimize the final bugs present in the code and to detect and correct them before release. This thesis describes the extension of DIALYZER, a static analysis tool designed to find discrepancies in ERLANG programs, with the introduction of *intersection types* that lead to impovements in its accuracy and the addition of a new module capable of finding errors in the use of *behaviours*, which correspond to *abstract classes* as they appear in object-oriented languages. These extensions lead to the detection of important errors in already thoroughly tested code.

Keywords

Static analysis, Type inference, Intersection types, Behaviours, Abstract classes, ERLANG, DIALYZER

Contents

List of Tables

List of Listings

Chapter 1

Introduction

DIALYZER is one of the most widely-used tools in the development of ERLANG programs. Its name stands for DIscrepancy anALYZer for ERlang and it does exactly that: in an inherently dynamically typed language such as ERLANG, DIALYZER is able to detect many type related discrepancies using static analysis. The initial version reported type errors using success typings but subsequent extensions allowed for verification of user contracts, detection of violations of the opaqueness of certain abstract data types and recently even warnings about race conditions [2, 10, 11, 12].

Altough taking part in the development of such a tool is a pleasure in itself, this thesis begun with a more concrete motivation: the extension of DIALYZER to detect discrepancies in the use of ERLANG's *behaviours* (more on these in Section 2.2), including simple type-related checks and enabling the new race detection analysis to pass through them undistracted. This goal was accomplished easily, as described in Chapter 3, but the cost was that DIALYZER produced some false warnings that couldn't be avoided using the existing type inference algorithm (see Section 3.2).

As this couldn't be tolerated (one of DIALYZER's cornerstones is that it is never wrong about a warning it emits) the second part of this thesis came into focus. When one designs a tool that promises to detect discrepancies in code soundly the main question that needs to be addressed is not what to include but what to leave outside. One of the greatest compromises made in DIALYZER's initial design was that it would work with a type system **without** *intersection types* for functions (see Section 2.3). This lead to its inability to report glaring errors as the one presented in Section 2.4 and was the reason behind the violation of soundness in behaviour analysis as well.

As in every happy story, this is no longer the case! Chapters 4 and 5 show how intersection types can be generated and used. DIALYZER emerges stronger than before, able to catch both abuses of behaviours and a whole new range of actual errors in code. What is more, further improvements are now easily attainable and are presented in Section 6.2.

Organization

After some preliminary knowledge presented in Chapter 2, this thesis is organized in two main parts. The first one (Chapter 3 describes the new handling of *behaviours* by DIALYZER. The second (Chapters 4 and 5) describes the design, implementation and evaluation of *intersection types* in DIALYZER's type system. Finally, Chapter 6 contains related work in other languages and suggests further work on the topics of this thesis.

Chapter 2

Preliminaries

2.1 Erlang and OTP

Erlang is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management and support for multiple platforms [1]. The number of areas where Erlang is actively used is increasing. However, its primary application area is still in large-scale embedded control systems developed by the telecom industry.

The main implementation of the language, the Erlang/OTP (Open Telecom Platform) system from Ericsson, has been open source since 1998 and has been used quite successfully both by Ericsson and by other companies around the world to develop software for large commercial applications.

Nowadays, applications written in the language are significant both in number and in code size making Erlang one of the most industrially relevant declarative languages.

2.2 Behaviours

As expected from an industrially used framework, OTP provides components that make the use of the language easier. Using these components developers are able to take advantage of the aforementioned language features. An excellent guide on how to develop a fault-tolerant, distributed and concurrent application is available to help new developers learn the language quickly [5].

One of the key elements of the framework are the *behaviours*. These correspond to the *abstract classes* or *interfaces* found in object-oriented languages, like Java, as they divide the functionality of a component into a generic part (the *behaviour* module) and a specific part (the *callback* module).

OTP provides many behaviour modules. To implement a process such as a server for example, the user only has to implement the callback module which should export a predefined set of functions, the callback functions. It is also possible for a developer to design his own behaviour, either by splitting the functionality of the program or by following a known design pattern [3].

2.3 Dialyzer

Dialyzer is a static analysis tool included in the OTP since 2007. It can detect a wide variety of discrepancies (i.e., type errors, software defects such as exception-raising code, hidden failures, unsatisfiable conditions, redundancies such as unreachable code, race conditions, etc.) in single modules or entire applications. Dialyzer is totally automatic, extremely easy to use and particularly successful in identifying software defects which may be hidden in Erlang code, especially in program paths which are not exercised by testing.

In the heart of Dialyzer lies a soft typing system. Its purpose is essentially to capture the biggest set of terms for which it can be proven that type clashes will occur. The type signatures that Dialyzer infers, called *success typings*, are the complement of that set of terms. Success typings are an over-approximation to the set of terms for which a function can evaluate: the domain of the signature includes all possible values that the function could accept as parameters, and its range includes all possible return values for this domain. Success typings are guaranteed to capture all intended uses of a function, along, perhaps, with some erroneous ones. Thus, any use of a function that is incompatible with its success typing will definitely fail. In effect, success typings approach the type inference problem from a direction opposite to that of type systems for statically typed languages.

For the actual workings of Dialyzer more details can be found in the relevant bibliography [6, 9, 10, 11, 16]. Only some key features will be presented concisely as they are relevant with the modifications this thesis describes.

2.3.1 Analysis phases

Dialyzer operation can be split in two phases:

1. **Find success typings:** In this phase Dialyzer traverses the code of every function included in the analysis and finds the success typing of each function. This requires several iterations of simple constraint solving and dataflow analysis. In the end, every function is assigned a final success typing.

2. **Emit warnings:** After the success typings have been fixed the code is traversed one more time. In this run a warning is emitted whenever a discrepancy is detected.

2.3.2 Refinement of success typings

In the calculation of success typings, functions that are not exported undergo a further refinement. The actual calls to these functions are used to calculate the effective domain. This is possible, as all the calls are located in the module under analysis. Restricting the domain may tighten the final success typing and possibly render some clauses unneeded. In Listing 2.1 an example is provided. Even though `foo`'s initial type will be calculated as $number \rightarrow number$ the final success typing will be $42 \rightarrow 43$ allowing a stricter return type for *test* as well (initially *number*, finally 43).

```
-module(refine).
-export([test/0]).

test() ->
  foo(42).

foo(X) -> X + 1;
```

Listing 2.1: Refining a local function

2.3.3 Contracts

DIALYZER can take into account annotations placed by developers to further restrict the success typings. These are called *specs* [9] and are used for both type checking (by DIALYZER) and documentation (by EDOC). The contracts may have more than one clauses, as it can be seen in the examples in Listing 2.2. These clauses though should not overlap (a certain call must belong in exactly one of them).

```
%% Permitted spec with one clause
-spec foo(number()) -> number().

%% Permitted spec with two clauses
-spec foo(integer()) -> integer();
         (  float()) -> float().

%% Non-permitted spec. 'bar' is an atom and overlaps with the 2nd clause
-spec foo( 'bar') -> 'ok';
         (atom()) -> 'error'.
```

Listing 2.2: Permitted and non-permitted specs

2.4 Intersection types

Intersection types are types describing values that belong to both of two other given types. For example, in most implementations of C the signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting either signed or unsigned chars, because it is compatible with both types.

Most modern statically typed programming languages support overloaded functions. Such functions execute different code depending on the type of the arguments they receive. Intersection types are useful for describing the type of such functions: an example is a function with type $Int \rightarrow Int | Float \rightarrow Float$. This function will return Int if called with an Int argument and a $Float$ if called with $Float$. Type checking is an essential part of the semantic analysis of a statically typed language's compiler.

However, ERLANG is a dynamically typed language, so type checking is not part of the compilation. What is more, the relevant tool, DIALYZER, was designed without intersection types. In Listing 2.3 we see a simple overloaded function, `foo`. DIALYZER using *success* *typings* will find that the function succeeds for arguments a or b and the return may be 1 or 2. Lacking intersection types, this will be expressed as $a|b \to 1|2$. This is an overapproximation because `foo` will never return 2 if the argument is a and vice-versa. This will cause the error presented in the combination of `foo` with the other functions in the example to pass undetected [1].

```
foo(a) -> 1;
foo(b) -> 2.

bar(1) -> ok.

buz(X) ->
  bar(foo(X)).

test() ->
  buz(b).
```

Listing 2.3: Trivial error that can be detected with the use of intersection types

The goal of the second part of this thesis is the inference of intersection types and is presented in Chapters 4 and 5.

[1]In fact this will happen only if `buz` is exported. Otherwise the refinement described in Section 2.3 will find that buz will be called with b only and will refine `foo`'s type with this info, catching the error.

Chapter 3

Finding discrepancies in behaviour usage

As described in Section 2.2, behaviours are ERLANG's equivalent of *abstract classes*, as they appear in object-oriented languages like Java. When using *abstract classes* the developer might make trivial mistakes such as forgetting to implement a particular abstract method or implementing it incorrectly so that it "doesn't fit" with those already provided. In ERLANG the abstract methods are called *callbacks*. ERLANG's compiler detects only the lack of implementation of any *callbacks* but DIALYZER can be used to further aid the developer by verifying whether his implementations have the expected success typings, ensuring thus that they "fit well" with the already provided infrastructure. DIALYZER's recent feature, race analysis [2], can also be extended to detect races present in code that uses behaviours. Each extension will be presented in a separate section.

3.1 Usage of behaviours

ERLANG developers use behaviours heavily as they readily provide some of the key features of the language (concurrency, communication, distribution and fault-tolerance) and allow the developer to focus on the particular aspects of his implementation, ignoring these parameters. In Table 3.1 the most common behaviours in OTP are presented, along with some of the callbacks they require.

Developers may also write their own behaviours, whenever a common infrastructure may be used for many specific implementations.

3.1.1 Declaration of a behaviour

A module describing a behaviour exports a specific function: `behaviour_info(callbacks)`. This returns the expected callbacks in the form of a list of tuples containing the names of the callback functions as atoms and their arity as integers. The example in Listing 3.1 is taken from the `gen_server` behaviour.

Module	Description	Callbacks
gen_server.erl	Generic server behaviour. Contains a state that is manipulated by calls (that require a reply) and casts (that do not wait for a reply).	init, handle_call, handle_cast, terminate
gen_fsm.erl	Finite state machine behaviour. A finite number of states exist along with the messages each state accepts, the replies that are sent and the state change that may follow.	init, State1, State2, ..., StateN, terminate
gen_event.erl	Generic event handler. Event handlers register in a central event manager and are notified for any event that arrives.	init, handle_event, terminate
application.erl	ERLANG application. An application is a collection of modules that implement some specific functionality and can be started and stopped as a whole.	start, stop
supervisor.erl	A process which supervises other processes called child processes. A child process can either be another supervisor or a worker process.	init

Table 3.1: Common OTP behaviours

```
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
  [{init,1}, {handle_call,3}, {handle_cast,2},
   {terminate, 2}, {code_change, 3}].

%%% The user module should export:
%%%
%%%   init(Args)
%%%     ==> {ok, State}
%%%         {ok, State, Timeout}
%%%         ignore
%%%         {stop, Reason}
%%%
%%%   handle_call(Msg, {From, Tag}, State)
%%%
%%%    ==> {reply, Reply, State}
%%%        {reply, Reply, State, Timeout}
%%%        {noreply, State}
%%%        {noreply, State, Timeout}
%%%        {stop, Reason, Reply, State}
%%%              Reason = normal | shutdown | Term
%%%
%%% ....  MORE COMMENTS FOR THE OTHER THREE CALLBACKS HERE .....
```

Listing 3.1: Generic server's declaration of callbacks

3.1.2 Better declaration of a behaviour

Often, though not always, the behaviour module also contains some additional information in the form of comments, as shown in Listing 3.1. The problem with comments is that they are in free text form, often lacking some information as in the case above, and cannot be trusted or mechanically processed.

Instead of the form described in the previous Listing, the `behaviour_info(callbacks)` clause can be substituted with the attributes shown in Listing 3.2 which also specify the types which are expected from these callbacks.

```
-callback init(Args :: term()) ->
    {ok, State :: term()} |
    {ok, State :: term(), timeout() | hibernate} |
    {stop, Reason :: term()} |
    ignore.

-callback handle_call(Request :: term(), From :: {pid(), Tag :: term()},
                      State :: term()) ->
    {reply, Reply :: term(), State :: term()} |
    {reply, Reply :: term(), State :: term(), timeout() | hibernate} |
    {noreply, State :: term()} |
    {noreply, State :: term(), timeout() | hibernate} |
    {stop, Reason :: normal | shutdown | term(), Reply :: term(),
                                           State :: term()} |
    {stop, Reason :: term(), State :: term()}.

%%% .... MORE CALLBACK ATTRIBUTES .... %%%
```

Listing 3.2: Callback attributes

These attributes are identical with specs so Dialyzer can use these as a reference to compare the inferred types of the callbacks. Incidentally, the above example shows various interesting things:

1. Using the language of types and specs, one can provide information both for documentation purposes and for types as e.g. in `From :: pid(), Tag :: term()`
2. Comments are often incomplete or can easily become obsolete as e.g. the `hibernate` value is nowhere mentioned.

3.2 Finding discrepancies in callbacks

Dialyzer's extension to use these callback attributes was simple and straightforward. After the success typings were calculated, they were compared against the attributes and warnings were emitted when the latter were not subtypes of the former. In this way the discrepancies shown in Table 3.2 were found [1]

[1]Only definite results are presented. Some more results need verification from the OTP team as the documentation on which the callback attributes were based might be outdated.

Application	Description	Behaviour Used	Discrepancies
inets	Internet clients and servers	gen_server	1
		tftp	1
dist_ac	distributed application controller	gen_server	1
mnesia	distributed DBMS	gen_server	2
ssh	SSH application	gen_server	2
error_logger	Stdlib's error logger	gen_server	1
Total discrepancies			8

Table 3.2: Behaviour Discrepancies in OTP Applications

All these discrepancies correspond to cases where a callback has a wider return type than the one described in the relevant attribute. The most common warning was about the return value of `gen_server`'s callbaks `handle_cast` and `handle_info` which sometimes erroneously included `{reply, ...}`.

Some special attention was given to the `gen_server` module, as its API returns depended also on the success typings of the callbacks. Specifically for the API's `start` and `start_link` functions, *"If callback init/1 fails with Reason, the function returns {error,Reason}. If callback init/1 returns {stop,Reason} or ignore, the process is terminated and the function returns {error,Reason} or ignore, respectively."*. This was taken into account when calls to these functions were found, as otherwise false warnings were emitted for the inets and other applications.

Dialyzer emitted some other false warnings as well. These warnings came from situations where two or more callback functions used the result of a common underlying function as a reply, or when one callback called directly another. The example presented in Listing 3.3, from the mnesia application, falls into the latter category, as `handle_info` calls a specific clause of `handle_call`. Using the existing algorithm for type inference this will result in the inclusion of *all* of `handle_call`'s possible returns in `handle_info`'s return type. This causes the warning shown in the end of the Listing to be emitted, as `handle_call` might also return `{reply,…}` in other clauses, even though this will never happen in the call from `handle_info` (the returns of which are in lines 8, 11 and 23 and are all `{noreply,…}`).

This was the motivation for the second part of this thesis which deals with cases like this, where a specific call's return type is certainly narrower than the return type of the function. See Chapters 4 and 5.

3.3 Use of behaviour information to find more race conditions

Dialyzer was recently extended with the ability to detect data races in Erlang programs [2]. This extension makes heavy use of the dataflow analysis as race conditions appear when a value is obtained from two separate processes, modified and then written back.

In cases where OTP's behaviours are used, the flow of data is difficult to monitor because the behaviours' APIs use parameters obtained in runtime to make calls to functions in the

```
handle_call({connect_nodes, Ns}, From, State) ->
    ...
    case mnesia_monitor:negotiate_protocol(Check) of
        busy ->
            erlang:send_after(2, self(), {connect_nodes,Ns,From}),
            {noreply, State};
        [] ->
            gen_server:reply(From, {[], AlreadyConnected}),
            {noreply, State};
        GoodNodes ->
            mnesia_lib:add_list(recover_nodes, GoodNodes),
            cast({announce_all, GoodNodes}),
            case get_master_nodes(schema) of
                [] ->
                    Context = starting_partitioned_network,
                    mnesia_monitor:detect_inconcistency(GoodNodes, Context);
                _ -> %% If master_nodes is set ignore old inconsistencies
                    ignore
            end,
            gen_server:reply(From, {GoodNodes, AlreadyConnected}),
            {noreply,State}
    end;
...

handle_info({connect_nodes, Ns, From}, State) ->
    handle_call({connect_nodes,Ns},From,State);

%% Produces the warning:

mnesia_recover.erl:850: The inferred return type of the handle_info/2
callback includes the type {'reply','ok' | {'ok',_},_} which is not a
valid return for the gen_server behaviour
```

Listing 3.3: A false warning

callback modules. Therefore any values provided as arguments to behaviour API calls may end up in the callback module and cause a race condition that is impossible to detect as the call from the API to the callback is dependent on runtime parameters. A special "bypass" mechanism was added in the race detection that translates calls to the behaviour API of OTP's behaviours into the respective call to a callback function, as they are described in the documentation. Examples of such translations for the `gen_server` module are given in Table 3.3.

Call to API's	... is translated to a call to callback's
start_link, start	init
call, multi_call	handle_call
cast, abcast	handle_cast

Table 3.3: OTP's `gen_server` translations

This extension caught a specific bug that escaped the existing race condition analysis. It is currently being tested with other additions to the race analysis.

Chapter 4

Intersection Types Generation

As described in Section 2.3, DIALYZER has two distinct phases in its analysis: during the first it calculates the success typings of all the functions and during the second it finds the discrepancies in their use. In this chapter the calculation of intersectioned success typings will be presented, leaving their usage in discrepancy detection for Chapter 5.

4.1 Original type system and analysis

Before describing the design and implementation of the intersection types as well as the analysis needed to produce and use them, a brief overview of the existing ERLANG's type system will be given, focusing on the type of functions and the analysis performed by DIALYZER to generate them. Further details, especially for the analysis, are available in the Master thesis of Elli Fragkaki [6].

4.1.1 Type system

ERLANG's type system includes types for all the basic term sets. These types form a lattice, with the type *any()* being the top type and *none()* the bottom. Table 4.1 briefly describes these sets and Table 4.2 contain the most commonly used ERLANG types. More information on these can be found in the language manual and relevant publications and bibliography [4, 9, 15].

4.1.1.1 Function type

Some special attention should be given to the form of the function type. As shown in Table 4.2, a function's type consists of two parts:

1. The first part describes the type of the function's arguments. This can be either a product of specific length, with one type for each of the function's arguments, or the type *any()* if we have no information about the number of arguments.
2. The second part describes the return type of the function. This can be a regular type or the special type *unit()* for functions that are not supposed to return[1].

[1]This is the case for example in a function that implements a server's main loop

Category	Description	Examples
Integer	A mathematical integer	−31, 0, 17, 42
Float	A floating point number	−0.123, 3.14
Atom	A named constant	hello, 'World'
Binary	An untyped series of bytes	«255,0,98», «42»
Bitstring	An untyped series of bits	«99,3:2», «1:1,0:1», «4»
Pid	A handle for referring to an Erlang process	–
Port	A handle for referring to an external program	–
Reference	A term unique within a runtime environment	–
Fun	A callable function object	fun(X) → X + 1 end, fun lists:reverse/1
Tuple	A compound term with a fixed number of elements	{0,alabama,3.14}, {answer,42}
List	A compound term with a variable number of elements (not necessarily of the same type)	[1,2,3], [42,answer], [for,whom,the,bell,tolls]

Table 4.1: Categories of Erlang terms

4.1.1.2 Type operations

Working with types requires special operators. The most important of them will be presented here (as these were the ones which were mainly affected by the introduction of intersection types), along with the particular usage of them on function types.

Supremum: The supremum of two types is the smallest type that contains them both. Erlang supports union types as shown in Table 4.2 as well as special unions for common types (like atoms, integers and tuples). Dialyzer may overapproximate a union type in cases where detail exceeds a certain level to keep the success typings analysis terminating and efficient. Such is the case with large sets of atoms for example: the type of the single-character atoms corresponding to the lowercase letters of the English alphabet is *atom()* and not `a | b | ... | z`.

The original calculation of two function types' supremum is the main reason for our inability to detect errors such as the one presented in Section 2.4. As the type system has one field for the success typings of the arguments and one for the function's return, supremum simply creates the union of the arguments and the result and stores them in the respective fields. This produces wider success typings than desired (in the example shown in Listing 4.1 a function that takes an *atom()* and returns a *number()* is included in the type, even though none of the original members includes it). If the functions have different arity the domain type collapses into *any()*.

Infimum: The infimum of two types is the biggest type that is contained in both.

In function types of the same arity this means that the infimum will have the infimum type for each argument and the infimum of the return types.

Equality: As there are no aliases in the representation of types, two types are equal if and only if they are syntactically equal. This applies even to the internal representation

Term Group	Related Types	Represented Terms
Integers	*<Int>*	only a specific integer, *<Int>* (singleton type)
	<Lo>..<Hi>	integers between *<Lo>* and *<Hi>*
	integer()	all integers
	non_neg_integer()	non-negative integers
	pos_integer()	positive integers
	neg_integer()	negative integers
Floats	float()	all floats
Atoms	*<Atom>*	only a specific atom, *<Atom>* (singleton type)
	atom()	all atoms
Binaries	binary()	all binaries
	«»	only the empty binary (singleton type)
	«_:*<Base>*»	binaries of length *<Base>* (in bytes)
Bitstrings	bitstring()	all bitstrings
	«»	only the empty bitstring (singleton type)
	«_:_**<Unit>*»	bitstrings of length $k \times$ *<Unit>* (in bits)
	«_:*<B>*, _:_**<U>*»	bitstrings of length *<B>*$\times$*<U>* (in bits)
Pids	pid()	all pids
Ports	port()	all ports
References	reference()	all references
Funs	fun()	all functions
	fun((...) $\rightarrow$ *Type*)	functions of any arity returning *Type*
	fun(() $\rightarrow$ *Type*)	zero-arity functions returning *Type*
	fun((T_1,...,T_N) $\rightarrow$ R)	N-arity functions accepting arguments of types T_1,...,T_N and returning R
Tuples	tuple()	all tuples
	{}	only the zero-size tuple (singleton type)
	{$Type_1$,...,$Type_N$}	tuples of N elements, of types $Type_1$,...,$Type_N$
Lists	[]	only the empty list (singleton type)
	[*Type*]	lists with elements of type *Type*
	[*Type*,...]	non-empty lists with elements of type *Type*
—	any()	all Erlang terms
	none()	no terms (special type)
	T_1 | T_2 | ... | T_N	the union of all terms represented by T_1, T_2, ..., or T_N

Table 4.2: Built-in Erlang types

```
Supremum:
Type A : fun((atom())   -> atom())
Type B : fun((number()) -> number())
Result : fun((atom() | number()) -> atom() | number())
```

Listing 4.1: Original function supremum example

which keeps every particular set (like *atom()* or *integer()*), as well as mixed unions, ordered.

Functions types are no exception and should also be syntactically equal to be equal.

Is Subtype: This operation is broken down to the calculation of the infimum of the two types and the check for its equality with the subtype candidate.

There are also special operations for function types:

Function domain: Returns a list of types, one for each of the function's arguments.

Function range: Returns the range of the function.

4.1.2 Original success typing analysis

Dialyzer begins the analysis by finding the calls between functions and creating the respective callgraph. From this callgraph a partial ordering of the functions is obtained and all functions' success typings are calculated beginning from those which have no calls and building on top of them.

The success typings analysis assigns a type variable to each of the code's original variables and stores a mapping from each variable to a type. Functions get two variables, one primary and a second one to be used in self-calls and calls within *strongly connected components* (SCCs)[2]. After that, the code is traversed to generate constraints and subsequently these are processed to obtain the final type for each function.

A brief presentation of the main types of constraints and the processing algorithm will be included here to show what needs to be changed. Further details are provided in the relevant bibliograhy [6, 11].

4.1.2.1 Constraints

The constraints belong to one of the following kinds:

Simple constraints: The simplest form of constraint states that a certain type should be equal to another or subtype of another. This is a natural requirement for function arguments for example, which must be subtypes of the corresponding success typing, calculated earlier in the analysis.

Conjunctive lists: The constraints generated from subsequent statements are stored in conjunctive lists as all must be satisfied at the same time. In simple functions the final constraint might be a conjunctive constraint list with simple constraints as elements.

Disjunctive lists: When branches of any kind are present in the code, each side of the branch generates a conjunctive list and all these lists are combined under a disjunction. After processing each of the conjunctions, the types for the disjunction are calculated by getting the supremum of the types for each variable on each branch.

[2]These correspond to functions that call each other forming a circle of calls

Constraint references: Funs without name generate these. These are special and not affected by the intersection types extension so they are simply mentioned for completeness.

```
%%Sample code

bar(1) -> 5;
bar(2) -> 10.

foo(a) -> b;
foo(X) ->
  Y = bar(X),
  Y*X.

%% Supposing we have alredy found the success typing:
%% bar(1 | 2) -> 5 | 10
%% The constraints for foo/1 are:

Conjunctive List 1:                   <- All the constraints for foo
 * var(1) eq fun(var(2)) -> var(3)  <- Tying foo type to it's args and ret
 * Disjunctive List 2:               <- Due to the two clauses
 *   * Conjunctive List 3:           <- Constraints for the first clause
 *   *   * var(2) eq a
 *   *   * var(3) eq b
 *   * Conjunctive List 4:           <- Constraints for the second clause
 *   *   * var(2) sub 1|2            <- X (var(2)) is used as argument of bar
 *   *   * var(4) sub 5|10           <- A hidden variable (var(4)) for the result
 *   *   * var(5) eq var(4)          <- Assign the result to Y (var(5))
 ...
```

Listing 4.2: Constraint examples

In Listing 4.2 an example is given. The constraints of the function `foo` are collected in a main conjunctive constraint list. In this list there exist some notable constraints:

1. **Generic function constraint:** This constraint has the form of the first element in the conjunctive list 1 of Listing 4.2. Its purpose is to bind the function's type variable to the ones of the arguments and the result. In the example *var(1)* is the type variable of a function with one argument (with type variable *var(2)*) whose return type is *var(3)*. This constraint is the actual constructor of the function's type.

2. **Constraints from clauses:** If the function has clauses, the list contains a disjunctive list of the constraints generated in each of them (an example is the disjunctive list 2 in Listing 4.2). If the function has only one clause the constraints of it are added in the main conjunctive list as is.

3. **Refined function constraint:** This constraint comes from the dataflow analysis and restricts the whole type of unexported functions according to the actual calls that are present within the module. This constraint is omitted when the function is exported or dataflow has not yet been performed (the example has no such constraint).

The previous constraints are present in the form described above in every main conjunctive list. Some other forms of constraints that are present in almost every function are these:

1. **Branches:** Branches such as `case` statements generate disjunctive lists, just like clauses do.
2. **Function calls:** These produce a conjunctive list, requiring both the result's and the actual parameters' type variables to be subtypes of the respective success types.
3. **Self and SCC calls:** These are treated specially: Initially these calls are supposed to fail. On subsequent iterations the types calculated in the previous step are used to extend the types of the argument and the result. In this way we begin to extract the type from clauses that are sure to return and build on top of them to find wider success typings. The overapproximations mentioned in Section 4.1.1.2 (Supremum operator) make this procedure efficient, as after a certain limit the types collapse into generic ones.

4.1.2.2 Processing

Processing is a fixpoint procedure when it comes to anything but simple constraints. The latter simply restrain any variables they contain according to the operation they contain (equality or subtype) and store the result in the mapping. Lists and refs store the old mappings and compare the new ones against them to find a fixpoint as each element may affect others. Self-recursive functions and SCCs also require special treatment as the previously calculated types are fed back in to be further processed in the self or scc-related calls.

4.1.2.3 Storing

After the success typings for all the functions in a module have been calculated, they are stored in a *Persistent Lookup Table* (PLT) in order to be used by calls from other modules. The PLT may be stored in a file and imported in subsequent analyses to serve as a trusted starting point to analyze other modules that have calls to already analyzed ones.

4.2 Intersection types

In order to generate and use intersection types changes were required in the type representation, operators and Dialyzer's analysis.

4.2.1 Changes in the type system

4.2.1.1 Structure

We need the ability to store multiple domains with the respective ranges. Therefore we will substitute the original two fields in the function type with a list of tuples of arity 2. Each tuple will contain a domain and a range and will be referred as a *type clause*.

DIALYZER took advantage of the simple old form and stored the type of the function in the PLT using a tuple containing the two old parts. This had to be changed and a combined function type to be stored instead.

4.2.1.2 Semantics

For simplicity the order of the clauses will have no special meaning. This is the main difference with the ordinary ERLANG's function clauses, where pattern matching is used to select one and execute it while the others that follow are ignored. The consequence is that for every operation described all the clauses have to be taken into account.

This also changes the semantics of the final type. The syntax is similar to that of specs, **but** while in specs overlapping is not permitted (see Section 2.3), here it is allowed and the return type of each specific call is calculated by taking the supremum of the return types of **every** clause whose domain overlaps [3] with the inferred types of the arguments in the call.

4.2.1.3 Operations

The operations described in Section 4.1.1.2 are modified as follows, with regard to function types:

Supremum: If the functions have different arities we maintain the old behaviour, collapsing the domains to *any* and the range to the supremum of ranges. If the functions have the same arity, we simply add their clauses together to form the supremum. This produces an exact supremum. No issue arises in cases of duplicate domains as all the clauses are taken into account for further calculations (in fact clauses with equal domains are combined into one, as described later in this section).

Infimum: As each function might have more than one clauses, infimum is performed per clause. This means that each clause is compared against all the clauses of the other function and those who have infima that do not have *none* as an argument or return are kept in the result.

Reduction of clauses: The calculation of supremum and infimum is almost certain to produce types that are verbose. This makes the clauses list big, requirng both memory to store it and time to perform further calculations. An extra step was therefore introduced to reduce the number of clauses. Three separate methods of reduction are used:

1. **Combine domains:** Clauses with the same domain should combine their ranges with supremum. An example of this technique is given in Example 1 in Listing 4.3.
2. **Combine ranges:** Clauses with the same range may also be combined if this does not overapproximate the domains. This happens when the domains differ in exactly one position and the supremum of the types that differ contains

[3]This means that the infimum of every argument's type in the call and the respective type in the clause is not *none*

only the original types (we don't have overapproximations). This computation requires a fixpoint termination condition as further reduction may be possible in a successive pass. Examples 2 and 3 in Listing 4.3 illustrate this technique.

3. **Remove subclauses:** If both the domain and the range of a clause are subtypes of another clause's respective domain and range we can remove the clause, as anything using it will also use the superclause. This causes intersections to lose power in cases where catch-all clauses with return type *any* are present in the code, as these will cause all the rest to be absorbed in them. A solution to this issue is proposed in Section 6.2.2.1.

```
Example 1: Same domains in supremum
-----------------------------------
Type A : fun((a) -> b)
Type B : fun((a) -> c)
Result : fun((a) -> b; (a) -> c)
Reduced: fun((a) -> b | c)
-----------------------------------
Example 2: Same ranges in supremum
-----------------------------------
Type A : fun((a) -> c)
Type B : fun((b) -> c)
Result : fun((a) -> c; (b) -> c)
Reduced: fun((a | b) -> c)
-----------------------------------
Example 3: Same ranges, 2 passes
-----------------------------------
Initial  : fun((a,c) -> e; (a,d) -> e; (b,c) -> e; (b,d) -> e)
1st pass : fun((a,c|d) -> e; (b,c|d) -> e)
2nd pass : fun((a|b,c|d) -> e)
```

Listing 4.3: Clauses reduction examples

Sorting of clauses: The clauses are sorted according to the default ordering of Erlang's terms both before and after the reductions are performed to control both the order of the reductions and the final result. This destroys any relation between the original clauses in the code and the resulting success typings but it's important as it normalizes the type and maintains the desired property of "equality requires syntactic equality".

Equality and subtyping: Equality maintains its simple, syntax based check. The reasons behind this will become clearer after the presentation of the changes in the inference algorithm. Clause sorting is essential to maintain this property. Subtyping is also calculated as described in Section 4.1.1.2.

Function range: As described in Section 4.2.1.2, when asking for a function's range we may provide information about the argument types and retrieve a narrower type.

Other special type functions had to be modified as well to be compatible with the new representation. What is important to mention is that in any case where the inner types might be modified reduction had to be performed as well to ensure the syntactic equality.

4.2.2 Analysis

The initial steps of the analysis are not changed. Functions are sorted as described in Section 4.1.2 and the success typings are calculated per SCC. Changes are introduced in both the generation of constraints and their processing to produce success typings. As the most important change is implemented in the processing we will reverse the order of the presentation.

4.2.2.1 Changes in constraint processing

As we already described in Section 4.1.2.1, each function has all the constraints organized in a conjunctive list. This list contains disjunctive lists whenever a branch is present in the code. Moreover each of these has conjunctive lists with local maps which are checked for fixpoint and remain unaffected from the other sides of the branch. Therefore, the simplest and most natural way to maintain the relation between the various type variables (including those belonging to the arguments and the result) in each branch is within the local maps themselves.

Although the values of the type variables are being kept separate in that way, the type of the function itself is constructed in the main conjunctive list taking into account the supremum of all the values, as this is the correct way to handle the types in a disjunctive list. We need to "push" the constraint that binds the type variables of the arguments and the result with the type variable of the function into each local map. Taking the *disjunctive normal form* of the original constraint list accomplishes this goal in a natural way and separates all the interleavings where nested disjunctive lists are present (for example a case statement in a branch of a multi-clause function). An example is provided in Listing 4.4. In this way we generate the correct partial type in every branch and combine them all in the end using supremum which maintains the separation. The disjunctive normal form is already used in the generation of constraints from *guards* to gain precision. Another benefit we gain is that whenever a function is too complicated, the calculation of the disjunctive normal form can detect it and return the original constraint list which when further processed will return the old-fashioned collapsed type.

4.2.2.2 Changes in constraint generation

To gain the benefits of the disjunctive normal form when function calls are present we need to generate a disjunctive list as a constraint when processing them. The way this should be done is obvious: for every clause in the function type a separate conjunctive list is to be generated, binding the argument and result type variables to the respective success types in the clause. These conjunctive lists are then placed in a disjunctive list and the constraint is ready to be handled by the normalization (see Listing 4.5 for an example).

This is simple in cases where the success typing of the called function is calculated and fixed, as the generation of the disjunctive constraint can take place immediately. The hard case is the self-recursive functions along with those that belong in SCCs. For these we introduced a new *dynamic constraint* which is to be substituted before the calculation of the disjunctive normal form by the disjunctive list derived by the latest success type of the respective function.

```
%%Sample code

bar(1) -> 5;
bar(2) -> 10.

foo(a) -> b;
foo(X) ->
  Y = bar(X),
  Y*X.

%% Supposing we have alredy found the success typing:
%% bar(1 | 2) -> 5 | 10
%% The constraints for foo/1 are:

Conjunctive List 1:
 * var(1) eq fun(var(2)) -> var(3)
 * Disjunctive List 2:
 *  * Conjunctive List 3:
 *  *  * var(2) eq a
 *  *  * var(3) eq b
 *  * Conjunctive List 4:
 *  *  * var(2) sub 1|2
 *  *  * var(4) sub 5|10
 *  *  * var(3) eq var(4)

%% Disjunctive normal form of the list:

Disjunctive List 1:
 * Conjunctive List 2:
 *  *  var(1) eq fun(var(2)) -> var(3)
 *  *  var(2) eq a
 *  *  var(3) eq b
 *  Conjunctive List 3:
 *  *  var(1) eq fun(var(2)) -> var(3)
 *  *  var(2) sub 1|2
 *  *  var(4) sub 5|10
 *  *  var(3) eq var(4)
```

Listing 4.4: Normal form example

The usage of intersections simplified the generation of constraints from *contracts* as well, as a similar disjunctive list can be used for them as well.

4.2.2.3 Changes in refinement

The use of intersectioned types allowed for a small improvement in the refinement of success typings as well. The previous approach was *monovariant* in the sense that all the calls to the unexported functions were found and the types of the actual arguments were combined in a union that restricted the success typing. This restriction was taken into account in a new calculation of the success typing by solving the default constraints with the addition of the refinement constraint.

Allowing for intersection types, we can keep each call separate and expect better results from the refinement. This is closer to a *polyvariant* control flow analysis. The argument

```
%%Sample code

bar(1) -> 5;
bar(2) -> 10.

foo(a) -> b;
foo(X) ->
  Y = bar(X),
  Y*X.

%% Supposing we have alredy found the success typing:
%% bar(1) -> 5; (2) -> 10
%% The new constraints for foo/1 are:

Conjunctive List 1:
 * var(1) eq fun(var(2)) -> var(3)
 * Disjunctive List 2:
 *  * Conjunctive List 3:
 *  *  * var(2) eq a
 *  *  * var(3) eq b
 *  * Conjunctive List 4:
 *  *  * Disjunctive List 5:
 *  *  *  *  Conjunctive List 6:
 *  *  *  *  *  var(2) sub 1
 *  *  *  *  *  var(4) sub 5
 *  *  *  *  Conjunctive List 7:
 *  *  *  *  *  var(2) sub 2
 *  *  *  *  *  var(4) sub 10
 *  *  * var(3) eq var(4)
 ...
```

Listing 4.5: Disjunction for function calls

types from each call generate a conjunctive list and all these lists are placed under a disjunctive list where they can be handled by the normal form transformation in the usual way.

As an example take the simple reversal of lists, presented in Listing 4.6. The one-argument function is exported while the other is kept local and can therefore be refined. The initial success typing is very generic because the first clause doesn't restrict the second argument. Using the union of the types from the two calls we learn that the second argument is a list, so the result must be a list as well but no distinction is made. Only by separating the calls and solving each case separately can we obtain the maximum information from this code.

```
reverse(List) ->
  reverse(List, []).

reverse(    [], Acc) -> Acc;
reverse([H| T], Acc) -> reverse(T, [H| Acc]).

%% Initial success typing for reverse/2 is:
-spec reverse([_], _) -> any().

%% Using (_,[_]) as a refinement for the arguments yields:
-spec reverse([_], [_]) -> [_].

-spec reverse([_]) -> [_].

%% Using separate (_,[]) and (_,[_,...]) as a refinement
%% for the arguments yields:
-spec reverse([_,...],[_]    ) -> [_,...];
             ([]     ,[_,...]) -> [_,...];
             ([]     ,[]     ) -> [].

-spec reverse([_,...]) -> [_,...];
             ([]     ) -> [].
```

Listing 4.6: Refinement of the success typing of reverse/2

Chapter 5

Using Intersection Types

5.1 Testing with PropEr

Testing the implementation of intersection types proved to be an excellent opportunity to show the shine of another tool developed in SoftLab: PropEr [15]. Types are inherently an abstract data type with its own operators and properties that should be satisfied by them. A brief overview of the testing using PropEr will be given in this section.

5.1.1 Generating random function types

Property-based testing requires a generator for random input for the tests. Using PropEr we were able to create a generator for intersectioned function types with ease taking into consideration parameters as:

- Covering all the simple types for both arguments and result
- Testing operators for both same arity and differing arity functions
- Helping PropEr's reduction with simple primitive types (like 'a' for atoms)
- Pretty printing of failing tests using PropEr's ?WHENFAIL directive

Using PropEr was a very creative and fun experience as ideas could be tested quickly against the properties required.

5.1.2 Properties of function types

As the actual implementation was a result of experimentation, trivial properties as well as stronger ones were tested. Some of them:

1. Simple function types combine correctly into an intersection
2. The supremum/infimum of F with F is equal to F
3. Subtracting F from F yields *none*

4. F is subtype of F
5. If Inf is the infimum of F1 and F2 then Inf is a subtype of both of them
6. If Sup is the supremum of F1 and F2 then F1 and F2 are subtypes of Sup
7. If Sup is the supremum of F1 and F2, Inf1 is the infimum of Sup and F1, Inf2 is the infimum of Sup and F2 then Inf1 is equal to F1 and Inf2 is equal to F2
8. The supremum/infimum of F1 and F2 is equal to the supremum/infimum of F2 and F1
9. If Inf is the infimum of F1 and F2 then the infimum of F1 and Inf is equal to Inf
10. If F1 is subtype of F2 and F2 is subtype of F1 then they are equal

5.1.3 Side results

Using these properties and some early implementations with no syntactic equality (as presented in Section 4.1.1.2) we found a two minor omission in the type system of Erlang, in an operator that returned all the simple types contained in a composite type:

1. Lists did not break down in simpler types as the empty list and the nonempty list with the same contents.
2. The *number()* type did not break down to *integer()* and *float()* types.

Fixing the first omission led to the detection of many loose contracts in OTP (where nonempty list was sure to be returned whereas the contract included the empty list as a possible return).

5.2 Performance issues

Sparsely in the previous secrions we mentioned the need for limits in the geneneration of intersections. These limits were imposed when the analysis was under risk to become needlessly time consuming. The recent parallelization of Dialyzer [17] enables a more efficient utilization of the modern multicore machines and should push these limits further.

Number of clauses: In the calculation of the disjunctive normal form, in cases where too many clauses or deep nesting of branches is present it is possible for the normal form to have too many branches. A limit was put to the number of them to maintain both efficiency and usability, as a very long success typing would alse be impractical to present to the user. This limit requires that the *disjunctive normal form* has no more than 100 clauses.

Size of SCCs: The analysis of SCCs requires a separate fixpoint and the substitution of the *dynamic constraints* mentioned in Section 4.2.2.2 each time using the latest type. This becomes impractical when the SCC is particularly big so another limit was placed in the size of it. This limit requires that any SCC has no more than 30 members.

Iterations in SCCs and self-recursive: Before intersection types DIALYZER's overapproximations guaranteed that fixpoint would be reached in a reasonable amount of iterations. This is no longer the case. Consider the example in Listing 5.1. The iterative process will infer that the function returns 0 for input 0, 1 for input 1 and so forth, without any reason to stop or any mean to find a fixpoint (previously after a few iterations both success types would collapse into *integer()*). A limit of 10 tries for SCCs and 30 for self-recursive functions is applied here.

```
    id(0) -> 0;
    id(N) -> 1 + id(0)
```

Listing 5.1: A self recursive numeric identity function

In all these cases we simply skip the tranformation to the normal form after a fixed number of iterations. This causes the success typing to collapse as the combination of the arguments' and the return types happens in the end (as described in Sections 4.1.2 and 4.2.2.1).

5.3 Intersection analysis results

The actual usage of the success typings for discrepancy detection happens in a final dataflow pass on the code under inspection. We won't go into detail here on the various warnings that may be emitted as these are covered in detail in the relevant publications [6, 11, 10].

The only change we implemented is the substitution of the generic lookup for the return type of function calls with a lookup that takes into consideration the types of the arguments. In this way we can easily detect discrepancies like the one in the initial example (Section 2.4).

5.3.1 Generic discrepancies

Using the extended DIALYZER the results presented in Table 5.1 were found. They are divided in the following categories:

Failing calls: These are calls that are certain to fail. This is usually the result of a particular combination of arguments. This category includes calls that are supposed to fail but no spec is provided so that dialyzer knows not to worry (See Listing 5.2).

Unneeded cases: These are *case* statements that have an unneeded error-catching or catch-all clause (See Listing 5.3.

Exit calls: Calls that result in an *erlang:exit*. These come from error-handling functions that do not always fail. When such calls are present, user should specify that the function may not return (See Listing 5.4).

Nonmatching clauses: These are nonmatching clauses that are not catch-alls.

Deriving warnings: In some cases a root failure may cause several more warnings to be emitted. These are listed with this category. An example is functions that won't be called due to an error earlier in the flow of control. Fixing the root cause will eliminate these warnings as well.

Application	Description	Category	Discrepancies
asn1	Abstract Syntax Notation 1 tools	Exit call	1
auth	Network Authentication Server	Deriving warnings	9
edoc	Documentation generator	Failing call	1
erts	Erlang Run-Time System	Failing calls	2
		Deriving warnings	17
file	File Interface Module	Deriving warnings	25
hipe	High Performance compiler	Unneeded case	1
inets	Internet clients and servers	Failing calls	2
		Exit call	1
		Nonmatching clause	1
mnesia	distributed DBMS	Exit calls	7
		Unneeded case	1
ssh	SSH application	Unneeded case	1
ssl	Interface for Secure Socket Layer	Failing calls	2
		Unneeded case	1
		Deriving warnings	1
tv	graphical examination of ETS and Mnesia tables	Unneeded case	1
Total original errors			22

Table 5.1: New Discrepancies in OTP Applications

5.3.2 Behaviour related results

The introduction of intersection types removed all the false positives from the previously collected results on behaviour usage. The reason for this is precisely the one described in Section 3.2: intersection types were assigned to the functions that handled all the requests and the analysis was able to discern whether a particular call could end in each result instead of assuming all results were possible. None of the other warnings were affected.

5.3.3 Bonus results

In Section 5.1.3 we mentioned how the correction of a small omission produced warnings about overspecified functions. Another such small error regarded the relation between the *none* and *unit* types. The infimum of the two was considered to be *unit*. The correction of this error unearthed a heap of *"Function X has no local return"*.

```
httpd_request_handler.erl:439: The call
httpd_response:send_status(ModData::#mod{data::[], method::[any()],
  request_line::nonempty_maybe_improper_list(),
  parsed_header::[any()], connection::boolean()},501, [1..255,...])
will never return since it differs in the 2nd and/or 3rd argument from
the success typing arguments: (#mod{socket_type::'ip_comm' |
  {'essl',_} | {'ossl',_} | {'ssl',_}},100 | 304 | 400 | 408 | 413 |
416 | 500 | 503,any()) or (#mod{socket_type::'ip_comm' | {'essl',_} |
  {'ossl',_} | {'ssl',_}},301 | 403 | 404 | 414,[any()]) or
(#mod{socket_type::'ip_comm' | {'essl',_} | {'ossl',_} |
  {'ssl',_}},400 | 401 | 412,'none') or (#mod{socket_type::'ip_comm' |
  {'essl',_} | {'ossl',_} | {'ssl',_}},501,{atom() |
  [any()],[any()],[any()]})

%% A rather esoteric warning about the 501 Not Implemented HTTP status message
```

Listing 5.2: A call that will surely fail

```
con_desc(E) ->
    case cerl:type(E) of
        cons -> {?cons_id, 2};
        tuple -> {?tuple_id, cerl:tuple_arity(E)};
        binary -> {?binary_id, cerl:binary_segments(E)};
        literal ->
            case cerl:concrete(E) of
                [_|_] -> {?cons_id, 2};
                T when is_tuple(T) -> {?tuple_id, tuple_size(T)};
                V -> {?literal_id(V), 0}
            end;
        _ ->
            throw({bad_constructor, E})
    end.

%% Produces the warning

cerl_pmatch.erl:338: The variable _ can never match since previous clauses
completely covered the type 'binary' | 'cons' | 'literal' | 'tuple'
```

Listing 5.3: A redundant catch-all clause

```
...
check_if_valid_tag(<<>>, _, OptOrMand) ->
    check_if_valid_tag2(false,[],[],OptOrMand);
...

check_if_valid_tag2(_Class_TagNo, [], Tag, MandOrOpt) ->
    check_if_valid_tag2_error(Tag,MandOrOpt);

...

check_if_valid_tag2_error(Tag,mandatory) ->
    exit({error,{asn1,{invalid_tag,Tag}}});
check_if_valid_tag2_error(Tag,_) ->
    exit({error,{asn1,{no_optional_tag,Tag}}}).

%% Produces the warning

asn1rt_ber_bin.erl:3: The call
asn1rt_ber_bin:check_if_valid_tag2('false',[],[],OptOrMand::any()) will never
return since it differs in the 2nd argument from the success typing arguments:
('false' | {'APPLICATION',_} | {'CONTEXT',_} | {'PRIVATE',_} |
{'UNIVERSAL',_},nonempty_maybe_improper_list(),[] | {_,_,_},any())
```

Listing 5.4: A path that ends in a call to *exit*

Chapter 6

Related and Further Work

6.1 Related work

There is almost no other known cases where tools need to make checks like the ones implemented for ERLANG's *behaviours*, either in their generic usage or in the detection of races through them. Object-oriented languages that use the equivalent *abstract classes*, *virtual methods* and *interfaces* have static typing (C++, Java, OCaml, ...) which ensures the fitting of implementations in every case.

On the subject of intersection types in a dynamically typed programming language, only DRuby is a known analog and is presented below. Other related work has been centered around Dylan and JavaScript and is briefly mentioned as well.

Finally, the changes in the refinement procedure are related to a formal approach to control flow analysis, also described below

6.1.1 Diamondback Ruby (DRuby)

Diamondback Ruby (DRuby)[7] is a recent tool that blends Ruby's dynamic type system with a static typing discipline. It uses a similar approach as DIALYZER generating constraints for the variables then applying a set of rewrite rules exhaustively. Intersection types are included from the beginning in it's type system, but it cannot infer them automatically. As a result they need to be annotated by the developer placing them on the same level as DIALYZER's contracts (*specs*).

6.1.2 Dylan and JavaScript

Dylan is a dynamically typed object-centered programming language inspired by Common Lisp and ALGOL. In a recent publication Mehnert proposed an extension providing function types and parametric polymorphism to the language[13]. The function types specialize from the previous generic ones but do not include intersections. Powerful parametric polymorphism is provided though.

JavaScript is the main scripting language for Web browsers, and it is essential to modern Web applications. Applying type analysis to JavaScript is a subtle business because, like most other scripting languages, JavaScript has a weak, dynamic typing discipline which resolves many representation mismatches by silent type conversions. In their publication[8] Jensen, Møller and Thiemann develop such a type analyzer which like DIALYZER is fully automatic *but* is designed for *soundness* with regard to the absense of certain errors.

DIALYZER's success typings approach is mentioned in both these attempts to provide static typing and discrepancy detection to dynamically typed languages.

6.1.3 Refinement using control flow analysis

In their publication, Palsberg and Pavlopoulou [14] propose an approach to control flow analysis that goes one step further than the changes proposed in this thesis (Section 4.2.2.3). Instead of using the types of the arguments as they are presented in the various calls, a computation is performed to obtain "covers" from them and use these covers instead in the polyvariant refinement. For this idea to be applicable we need to figure out if the actual calls can be reduced to some elementary sets such that each call is exactly covered by some of them.

6.2 Further work

6.2.1 Behaviours

6.2.1.1 Automatic bypass of API for race detection

The "bypass" mechanism was designed to be extensible, allowing other behaviour API's to be connected with the respective callbacks. Anyone with a better understanding of the other behaviours may document the rest OTP's behaviours easily by adding code in *dialyzer_behaviours* module. It would be even better if this connection was tranferred in each behaviour's file, as a new attribute or as an extension on the *callback* attribute that was introduced in this thesis.

6.2.2 Intersections

6.2.2.1 Negative types

The next logical extension to the type system would be negative types. Examples would be *"any term except the integer 42"* or *"any atom except a"*. With this infrastructure, when DIALYZER generates disjunctive lists it will be able to eliminate the types already covered in previous clauses in the following ones. Thus a "catch-all" will not have type *any* but *"anything but X, Y and Z"* where *X, Y* and *Z* will be the types already covered by previous clauses.

6.2.2.2 Tighter coupling between type and code

Using normal form and sorting on the function's type has an impact on the relation between the type of the function and the actual code that generated it. Even though it was easy to find the cause of all the warnings emitted when intersection types were used, it might be better to narrow down a warning to a particular clause instead of the generic pointer to the first line of the function.

6.2.2.3 Better refinement

In the current form, if a lot of calls are present in the code the refinement analysis might fail due to the added effect of the disjunctive refinement constraint. The ideas presented in Palsberg and Pavlopoulou [14] might reduce the elements of the constraint in such cases, maintaining strictness.

Bibliography

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, second edition, 1996.

[2] Maria Christakis and Konstantinos Sagonas. Static Detection of Race Conditions in Erlang. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages (PADL'2010)*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer, January 2010.

[3] Ulf Ekström. Design Patterns for Simulations in Erlang/OTP. Master's thesis, Computing Science, Dept. of Information Technology, Uppsala University Sweden, 2000.

[4] Ericson AB. *Erlang Reference Manual User's Guide*, December 2010. Version 5.8.1, http://www.erlang.org/doc/design_principles/users_guide.html.

[5] Ericson AB. *OTP Design Principles User's Guide*, December 2010. Version 5.8.1, http://www.erlang.org/doc/reference_manual/users_guide.html.

[6] Elli Fragkaki. Explanation of Success Typing Violations in Erlang Programs. Undergraduate thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, 2010.

[7] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[8] Simon Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer Berlin / Heidelberg, 2009.

[9] Miguel Jimenez, Tobias Lindahl, and Konstantinos Sagonas. A Language for Specifying Type Contracts in Erlang and its Interaction with Success Typings. In *Proceedings of the 2007 SIGPLAN Workshop on Erlang*, pages 11–17. ACM, 2007.

[10] Tobias Lindahl and Konstantinos Sagonas. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In Wei-Ngan Chin, editor, *Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer Berlin / Heidelberg, 2004.

[11] Tobias Lindahl and Konstantinos 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.

[12] Manouk-Vartan Manoukian. Detection of Opaque Violations in Erlang Using Static Analysis. Diploma thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, 2009.

[13] Hannes Mehnert. Extending Dylan's type system for better type inference and error detection. In *Proceedings of the 2010 international conference on Lisp*, ILC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[14] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 197–208, New York, NY, USA, 1998. ACM.

[15] Emmanouil Papadakis. Automatic Random Testing of Function Properties from Specifications. Undergraduate thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, 2010.

[16] Konstantinos Sagonas and Daniel Luna. Gradual Typing of Erlang Programs: A Wrangler Experience. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 73–82, New York, NY, USA, September 2008. ACM Press.

[17] Ypatia Tsavliri. Parallelizing Dialyzer: a Static Analyzer that Detects Bugs in Erlang Programs. Undergraduate thesis, Department of Electrical and Computer Engineering, National Technical University of Athens, 2010.