National Technical University of Athens

School of Electrical and Computer Engineering
Division of Computer Science

Detection Of Opaque Type Violations in Erlang Using Static Analysis

Diploma Thesis

by

Manouk-Vartan Manoukian

Supervisors:
N.T.U.A Associate Prof. Konstantinos Sagonas

Software Laboratory
Athens, September 2009

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Software Laboratory

Detection Of Opaque Type Violations in Erlang Using Static Analysis

Diploma Thesis

by

Manouk-Vartan Manoukian

Supervisors:
N.T.U.A Associate Prof. Konstantinos Sagonas

This thesis was approved by the examining committee on the 16th of September, 2009.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Kostas Kontogiannis	Nikolaos Papaspyrou	Konstantinos Sagonas
N.T.U.A Associate Prof.	N.T.U.A Assistant Prof.	N.T.U.A Associate Prof.

(Signature)

.....
Manouk-Vartan Manoukian
Graduate Electrical and Computer Engineer, N.T.U.A.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Software Laboratory

Abstract

Erlang is a general-purpose programming language designed at the Ericsson Computer Science Laboratory. Erlang has extensive dynamic libraries in which a plethora of abstract data types are defined. However, programming in Erlang suffers from the lack of opaque types. Opaque types are especially necessary in a production environment since they provide solid contractual guarantees. Due to heavy pattern matching operations though, violations of the structure of abstract data types are a common occurrence. To address this problem we used static type checking analysis to reliably detect and warn about opaque type violations as well as a myriad of other type related errors. We believe that we have in place a system that will prove helpful in the development of new applications as well as in the maintenance and easier extension of existing code.

Keywords

opaque type violations, Erlang, static analysis, testing, debugging

Acknowledgments

I would like to express my gratitude to Prof. Sagonas for his guidance, never ending patience and supervision of my work. His hard work and dedication to software engineering have been nothing short of inspiring.

Thanks also go to Prof. Papaspyrou. Through his courses I was introduced and fascinated by computer science.

Also my parents deserve special praise for their care and affection in my early steps and for their struggle to provide for me the best education possible.

Finally I would like to thank myself for my utter awesomeness.

Contents

List of Figures

Chapter 1

Introduction

Abstract data types (ADTs) are widely used in today's programming environment. Most programming languages have libraries that define common data structures, queues, trees, stacks and dictionaries. Efficient and productive programming depends on using predefined ADTs. Programs are built on top of abstract data types often defining new data types on the way. Thus many layers of code are created. These layers rely on the code below them to operate consistently.

There are many reasons why the underlying structure of an abstract data type may change; using a different operating system, porting to a different architecture or optimizing the code, just to name a few. This may have no effect on programs using the ADTs if abstract principle is followed correctly and only the interface functions are used. On the other hand it may prove catastrophic to programs that directly access the structures. Programs like that need to be refactored and recompiled which is unacceptable in a competitive mission-critical environment. To reduce the chance of errors numerous programming languages use opaque types, hiding the structure fields from clients and only presenting an interface.

Erlang is not one of these programming languages. It is a dynamically typed functional programming language with extensive pattern matching capabilities and supports concurrent programming. It has a large user base, including large companies that develop many commercial applications. However, what Erlang doesn't have, is an implementation of opaque types. As it would be expected, an extensive module library with various abstract data types is available. Nevertheless, all these data structures are exposed and accessible from any application. Making matters worse, pattern matching on function heads and case clauses is very commonly used in Erlang code. Programmers, often, because of neglect or convenience, use pattern matching on ADTs. Practices like the above may lead to perfectly functional code, but are vulnerable to code changes. This is a problem that is further intensified by the module hot swapping system used in Erlang. Hot swapping is used so code can be changed without stopping a system. Changing a data type in such a system can crash a vital non-stop application if the data type is incorrectly accessed.

The following elementary example illustrates problems that might occur and the direction we have taken to combat them.

A simple queue library in Erlang v1:

```
-module(my_queue).
-export([new/0, add/2, dequeue/1, is_empty/1, length/1]).

-type my_queue() :: list().

-spec new() -> my_queue().
new() ->
    [].

-spec add(term(), my_queue()) -> my_queue().
add(T,Q) ->
    Q ++ [T].

-spec dequeue(my_queue()) -> {term(),my_queue()}.
dequeue([H|T]) ->
    {H, T}.

-spec is_empty(my_queue()) -> boolean().
is_empty([]) ->
    true;
is_empty([_]) ->
    false.

-spec length(my_queue()) -> non_neg_integer().
length(Q)->
    erlang:length(Q).
```

An application that uses the queue ADT:

```
-module(queue_use).
-export([do_it/0]).

do_it() ->
    NewQueue = [],
    Data = some_module:data(),
    Queue = my_queue:add(Data, NewQueue).
```

The flaws in the above application are obvious. The programmer has associated the queue abstract data type with the concrete type list. The initialization using a user defined construct is a mistake that will not cause the program to crash but may lead to the program becoming inoperable in the future. If we suppose that the author of the queue code notices that the `queue:length/2` function is being used frequently and it is inefficient to traverse large queues every time the operation is requested; the decision is made to store the size of the queue along side with the queued elements. Such a change might look like this:

Queue abstract data type v2:

```
-module(my_queue).
-export([new/0, add/2, dequeue/1, is_empty/1, length/1]).

-type my_queue() :: {non_neg_integer(), list()}.

-spec new() -> my_queue().
new() ->
    {0,[]}.

-spec add(term(), my_queue()) -> my_queue().
add(T,{S,Q}) ->
    {S+1,Q ++ [T]}.

-spec dequeue(my_queue()) -> {term(), my_queue()}.
dequeue({S,[H|T]}) ->
    {H,{S-1,T}}.

-spec is_empty(my_queue()) -> boolean().
is_empty({0,[]}) ->
    true;
is_empty({_S,_Q}) ->
    false.

-spec length(my_queue()) -> non_neg_integer().
length({S,_Q})->
    S.
```

Succeeding the code revision, it is apparent that the `queue_use` module will fail to execute properly. Nonetheless, because Erlang is a dynamically typed language, compilation of the module is still possible. Checks at compile time are minimal; the problems in this case will only be revealed at runtime, although, there are examples with subtler changes that even during execution, potential bugs may pass undetected.

To compensate for the lack of any systematic static checking, Dialyzer was created. Dialyzer is a static analysis tool developed by Tobias Lindahl and Konstantinos Sagonas. It uses type checking and dataflow analysis to identify software defects in Erlang programs. The version of Dialyzer we started from only uses Erlang's built-in types. Abstract data types (dictionaries, trees, sets) were treated as structured entities made up from the basic Erlang types and not as opaque "black boxes". Ergo it was not possible for Dialyzer to reveal abuses of the opaqueness of ADTs. Using dialyzer's default type checking — before the changes to the queue abstract data type — yields no warning since, structurally at least, the types match. After the changes dialyzer is able to detect points where type clashes will occur.

Dialyzer without opaque type information (my_queue v1):

```
Proceeding with analysis... done in 0m0.19s
```

Dialyzer without opaque type information (my_queue v2):

```
Proceeding with analysis...
queue_use.erl:7:
        The call my_queue:add(Data::term(),NewQueue::[])
        will never return since the success typing is
        (any(),{number(),[any()]}) -> {number(),[any(),...]}
        and the contract is (any(),my_queue()) -> my_queue()
done in 0m0.17s
```

Discovering problems this manner is not ideal. Firstly, the situation could have been averted before the data type change, if the opaqueness violations were detected. Secondly, the warnings do not bring to the programmer's attention that he is in fact dealing with an opaqueness related issue. In response to these flaws we have modified Dialyzer and expanded its capabilities so as to make detection of opaque type violations possible. Additionally we defined several Erlang abstract data types as opaque and also allowed for new modules to implement opaque types. The aim of our new analysis is to reveal opaque type violations that might occur. Such violations are pattern matching and type checking on opaque types, deconstruction of opaque ADTs and use of structured constructs in the place of opaque types. In the following example we demonstrate how it is possible to declare the queue ADT as an opaque type. The process is straightforward and especially painless, if proper type contracts are already in place.

Queue opaque data type:

```
-opaque my_queue() :: {non_neg_integer(), list()}.
```

The only alteration that is needed in this example is the replacement of the "`-type`" attribute with the "`-opaque`" attribute. The rest of the work is done by the contracts that were in place beforehand. The implementation here is irrelevant; the code may be altered, optimized and revised in any way. As long as the interface described above remains constant, the analysis will work towards finding opaqueness violations. If we now use Dialyzer with the help of the extra opaque information, we will receive warnings about the abuses of the abstract data types even if the program is type correct and will not fail during execution.

Dialyzer with opaque type information (my_queue v1):

```
Proceeding with analysis...
queue_use.erl:7:
        The call my_queue:add(Data::any(),NewQueue::[])
        does not have an opaque term of type
        my_queue:my_queue() in position 2
done in 0m0.17s
```

Dialyzer with opaque type information (my_queue v2):

```
Proceeding with analysis...
queue_use.erl:7:
        The call my_queue:add(Data::any(),NewQueue::[])
        does not have an opaque term of type
        my_queue:my_queue() in position 2
done in 0m2.40s
```

Both the above analysis results are the same. This of course is our goal. The structure of the queue abstract data type is made invisible. The warnings are identical because only the interface is used for the analysis and the underlying structures are not exposed. In contrast with the analysis made without the opaque declaration, the opaque violation flaws are detectable, regardless of type correctness. Defects detected will not necessarily result in runtime errors, however they will definitely constitute a breach of the abstract data type's interface. Furthermore, the warnings clearly state that an opaque type `my_queue:my_queue` is expected at position 2 of the `my_queue:add/0` call, instead the variable `NewQueue` is used, which has a *list()* type. Using the well defined warnings, the developer can easily locate and mend the potential defect.

This brief prologue is meant to familiarize the reader with our work. The examples only represent a trifle of the work done on the detection of opaque type violations. In later chapters we present the complete extent of the defect detection capabilities as well as the sortcomings of the analysis process. Also we talk about abstract and opaque data types, Dialyzer, success typings and the results of our work.

Chapter 2

Abstract Data Types

2.1 Definition

An abstract data type is a conceptual, non concrete category of objects sharing common characteristics, in a form a computer can use [5]. An abstract data type is a data type, whose operations are defined at a formal, logical level, without being restricted by operational details. This formal definition or specification becomes the sole interface for both application developers and programmers who implement the abstract data type in a computer language. The most used type in any computer language is, without a doubt, the integer type. We can consider the type integer as an example of an abstract data type. Integers may seem very concrete because we are accustomed to working with numbers. However, we do not know how they are represented in a particular computer, how mathematical operations are accomplished between two integers, or what their range might be. All we know is what the syntax and operations (interface) of the language tell us.

Some common ADTs, which have proved useful in a great variety of applications, are:

Container	Map	Priority queue	Stack
Deque	Multimap	Queue	String
List	Multiset	Set	Tree

2.2 Purpose

Abstract data types and their implementation are not ends in themselves. The blind (over)use of abstract data types does not guarantiee the success of a project. Rather, abstract data types are some of the basic tools for building correct, efficient, modifiable, reusable software.

Correct When using abstract data types, the work is broken in smaller, more manageable fragments. Reduction of the complexity leads in a reduction of flaws and common mistakes.

Efficient Abstract data types reduce bloating of code and increase productivity.

Modifiable Each abstract data type can be modified and optimized indepedently, without effecting the rest of the software project.

Reusable Reusability is the primary function of abstract data types. An abstract data type has to be implemented only once, but can be used many times by different applications.

2.3 Abstraction

Software engineering involves the development and application of careful methodologies for the writing of software. One of the most important principles in accomplishing this is the use of abstraction. Abstraction allows us to organize the complexity of a task by focusing on the logical properties of data and action, rather than on the implementation details. Two kinds of abstraction are of interest to computer scientists: procedural abstraction and data abstraction. Procedural abstraction is the separation of the logical properties of an action from the implementation details. Data abstraction is the separation of logical properties of data from the details of how the data are represented. Clearly, procedural abstraction and data abstraction are closely related: the operations within an abstract data type are procedural abstractions. An abstract data type encompasses both procedural and data abstraction; the set of operations are defined for any data type that might make up the set of values.

2.4 Specification of Abstract Data Types

When a person is implementing an algorithm on another's data structures, misunderstandings can occur, much time and money can be lost. Software engineering tries to keep this sort of problem from occurring by formulating ways to precisely define both procedural and data abstractions. These definitions (called interface specifications) describe the effect of these abstractions on the external environment. There are two common techniques for writing formal specifications: axiomatic (or algebraic) specifications and abstract models.

2.4.1 Axiomatic (Algebraic) Specification

The specification of an abstract data type can be described as a triple $\{D, F, A\}$. The set D contains the domains and ranges involved in the data types. F, the second set in the triple, contains the names of the allowable operations. A , the last set of the triple, contains the axions or rules that describe the semantics (the meaning) of the operations. The first part of the axiomatic specification for the stack data type is given below:

```
structure       Stack(of ItemType))
interface       Create                  -> Stack
                Push(Stack,ItemType)    -> Stack
                Pop(Stack)              -> Stack
                Top(Stack)              -> ItemType
                IsEmpty(Stack)          -> Boolean
end

D = {Stack, ItemType, Boolean}
F = {Create, Push, Pop, Top, IsEmpty}
```

The interface section (sometimes called the declaration section) lists the functions, their domains, and their ranges. There is no indication of what the functions do. The meaning of the functions is given in axioms.

Axiomatic specifications define the behavior of an abstract data type by giving axioms that relate these functions to one another. There are two types of functions: those that build, construct, or modify instances of the data type (called constructors) and those that provide some information about objects of the data type (called observers).

The axioms form the third set of the triple in the definition of an abstract data type. The axioms for the observer functions are written in terms of the constructor functions. The constructors that remove an item from the structure are defined in terms of constructors that add an item to the structure. The constructor functions that create new structures or put an item on the structure are not explicitly defined. The axiom for the stack example is given below.

```
axioms  for all S in Stack, i in ItemType let
        IsEmpty(Create) = True
        IsEmpty(Push(S,i)) = False
        Top(Create) = Error
        Top(Push(S,i)) = i
        Pop(Create) = Error
        Pop(Push(S,i)) = S
end
```

In our work we have used just the first part of the axiomatic specification (the interface section). We only need the domains involved and the domains and ranges of the interface functions. Axioms are difficult to produce for complex abstract data types and our analysis would not benefit from the axioms, as the information of what a function actually does, is not relevant in type checking process.

2.4.2 Abstract Model

Abstract models use the operations of another abstract data type to describe the semantics of the abstract data type being defined. The underlying model can be a well defined mathematical model or one that has been defined axiomatically. A different notation usually used to express the axioms in this technique. The operations are defined as procedure or function headings, with the axioms stated as preconditions and postconditions for the procedure or function. Preconditions define what the procedure or function can assume on entry. Postconditions define what is true on exit from the procedure or function. For the stack example, the non-indexed, unsorted list is used as the underlying model to define the abstract data type.

The properties of a stack are defined here in terms of lists. This in no way implies that the stack is implemented as a list, only that the behavior of the stack is defined in terms of the behavior of a list.

Notation:

S	a stack
S'	the stack S prior to the current operation
i	element of the stack
(i)	list with i as its only item
$()$	empty list
$//$	concatenation operator
$\perp$	undefined

Operations:

Create(VAR S:Stack)
Pre: $True$
Post: $S = ()$

Push(VAR S:Stack; i:ItemType)
Pre: $S' \neq \perp$
Post: $S = (i)//S'$

Pop(VAR S:Stack; VAR i:ItemType)
Pre: $(S' \neq \perp)$ AND NOT $(S' = ())$
Post: $S' = (i)//S$

IsEmpty(VAR S:Stack):Boolean
Pre: $S' \neq \perp)$
Post: IsEmpty $= (S = ())$

Figure 2.1: Abstract Model Definition for the Stack ADT

2.5 Visibility and Access (Opaque Types)

All types and variables defined and/or declared in an interface section of a module are visible to the programs using the module. A transparent data type, as they are called, is both visible and accessible. It is visible because the user can read the structure in the module listing, and it is accessible because the user can access parts of that type. An opaque data type is a data type whose name is listed in the interface section of a module, but whose actual type is defined in the implementation section of the module (as in Modula-2). A private data type is a data type defined in the interface section

of a module, but marked as being private (as in Ada). The users of opaque or private data types can declare variables of that type, but cannot access parts of them. Variables declared to be opaque or private can be passed as parameters and assigned to one another, but cannot be accessed in any other way.

Definition 1 ***Transparent Types*** *Data types whose description is visible. Component variables of that type can be accessed directly by the user.*

Definition 2 ***Opaque Types*** *Data types whose name only is visible. Access to parts of opaque types must be through operations defined in the interface section that defines the type name.*

Definition 3 ***Private Types*** *Data types whose description is visible, but access to variables of the type must be through the operations defined in the interface module.*

Opaque and private data types are language features that can be used with modules to provide true encapsulation. The user can declare variables of opaque or private types but cannot access them except through the interface provided.

2.6 Opaque Type Implementations

Abstract data types are essential in high level programming languages. The opaqueness of ADTs has been addressed in many of them and in others, like Erlang, it has been left on the programmers good intentions. In this section, we present how some languages employ opaque types in their operation.

2.6.1 Java, C++, C#

In object oriented languages, such as Java and C++, opaque types play a fundamental role, even though they do not appear as types, but rather as classes. Opaque types are actually objects that utilize information hiding. When defining a class, concrete data fields can be annotated as private; if an access attempt is made on a private field, the compiler will detect the violation and terminate the compilation. In fact, it is possible to hide some data fields and have others visible. On the other hand, functions that are needed to manipulate the data are defined as public and make up the interface that is presented to clients. Thus, opaque data types can be created encapsulated in a class, along with the necessary interface.

Opaque types example in Java:

```
private class Stack<Item> {
        private Node first;
        private class Node{
                private Item item;
                private Node next;
        }
        public void push(Item item) {...}
        public Item pop() {...}
}
```

2.6.2 C, Ada

C, Ada and C++ can implement opaque types by opaque pointers. Opaque pointers are nothing more than ordinary pointers, directed to a record or data type of unspecified type. If the language is strongly typed, programs and procedures that have no other information about an opaque pointer of type T, can still declare variables, arrays, and record fields of type T, assign values of that type, and compare those values for equality. However, they will not be able to de-reference such a pointer, and can only change the object's content, by calling some procedure that has the missing information [4]. The pointer to implementation idiom, as it is called, serves a dual purpose: not only does it hide the implementation details, but also, by hiding the details, recompilations of client code -because of changes in the declaration of the opaque types' structure- are made redundant.

Opaque pointer example in C:

```
/* stack.h */
typedef struct stack *stack_handle;
stack_handle create();
void push(stack_handle handle, int item);
int pop(stack_handle);


/* stack.c */
#include "stack.h"
struct node {
        int item;
        node *next;
};
struct stack {
        node *first;
};
stack_handle create() {...}
void push(stack_handle, int item) {...}
int pop(stack_handle) {...}
```

Bear in mind that `stack.h` is meant to be distributed with the compiled library `stack.so` and not the source file `stack.c`. Thus, we have made the compiler unaware of the structure of the type stack; but pointers to the structure can still be used with the matching functions.

2.6.3 Haskell

Haskell utilizes its module system and new type declarations to hide type implementation details [10]. It is possible to allow types, in the export list, but not their constructors. In this manner, the internal structure of abstract data types remains outside the scope of the client code. The only way to build or take apart the ADTs outside of the module, is by using the various abstract operations. The ability to have user-defined constructors -opposed to only primitive constructors, which are in the scope of any application-, is what makes Haskell's opaque types maintain their opaqueness.

Stack ADT in Haskell:

```
module Stack (Stack, new, push, pop) where

newtype Stack a = StackConst [a]

new :: Stack a
new = StackConst []

push :: Stack a -> a -> Stack a
push (StackConst s) element = StackConst (element:s)


pop :: Stack a -> (a, Stack a)
pop (StackConst (element:s)) = (element, StackConst s)
```

2.6.4 Standard ML

SML uses structures to collect relative definitions under a common name space, much like a module. However this doesn't help with making abstract data types opaque. In order to achieve this, SML requires for an abstract description to be provided. These abstract descriptions come in the shape of signatures. Signatures are essentially type masks that hide the real typings of the structure's members and replace them with specified type signatures [6]. Types defined in signatures can be declared without a representation; therefore SML considers them to be unique and not aliases of other types. As a consequence, applications that try to construct or de-construct such types, are met with type errors, because only members of the ADT are designated as capable to manipulate them.

Stack Structure and Signature in ML:

```
signature STACK
sig
        type 'a stack
        val empty : 'a stack
        val push  : 'a * 'a stack -> 'a stack
        val pop   : 'a stack -> 'a
end

structure Stack
struct
        type 'a stack = 'a list
        val empty = []
        val push = op ::
        fun pop (tos :: rest) = tos
end :> STACK
```

Chapter 3

The Erlang language

Erlang is a strict, dynamically typed functional programming language, with support for concurrency, communication, distribution and fault-tolerance. The language relies on automatic memory management. Erlang's primary design goal was to ease the programming of soft real-time control systems commonly developed by the telecommunications (telecom) industry. Erlang's basic data types are atoms, numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. A notation for objects (records in the Erlang lingo) is supported, but the underlying implementation of records is the same as tuples. To allow efficient implementation of telecommunication protocols, Erlang nowadays also includes a binary data type (a vector of byte-sized data) and a notation to perform pattern matching on binaries. There are no destructive assignments of variables or mutable data structures. Functions are defined as ordered sets of guarded clauses, and clause selection is done by pattern matching. In Erlang, clause guards either succeed or silently fail, even if these guards are calls to builtins, which would otherwise raise an exception, if used in a non-guard context. Although there is a good reason for this behavior, this is a language feature which often makes clauses unreachable in a way that goes unnoticed by the programmer. Erlang also provides a catch/throw-style exception mechanism, which is often used to protect applications from possible runtime exceptions. Alternatively, concurrent programs can employ so called supervisors which are processes that monitor other processes and are responsible for taking some appropriate clean-up action after a software failure.

Erlang/OTP is the standard implementation of the language. It combines Erlang with the Open Telecom Platform (OTP) middleware. The resulting product, Erlang/OTP, is a library with standard components for telecommunications applications (an ASN.1 compiler, the Mnesia distributed database, servers, state machines, process monitors, tools for load balancing, etc.), standard interfaces such as CORBA and XML, and a variety of communication protocols (e.g., HTTP, FTP, SMTP, etc.)

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 Erlang/OTP system has so far been used quite successfully, both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks, etc.), to develop software for large (several hundred thousand lines of code) commercial applications. These telecom products range from high-availability ATM servers, ADSL delivery systems, next-generation call centers, Internet servers, and other such networking

equipment. Their software has often been developed by large programming teams and is nowadays deployed in systems which are currently in operation. Since these systems are expected to be robust and of high availability, a significant part of the development effort has been spent in their (automated) testing. On the other hand, more often than not, teams which are currently responsible for a particular product do not consist of the original program developers. This and the fact that the code size is large, often make bug-hunting and software maintenance quite costly endeavors. Tools that aid this process are of course welcome.

Chapter 4

Dialyzer

The work on detecting opaque type violations is solely based on dialyzer and comes as an expansion to its functionality. Dialyzer is a lightweight static analyzer for Erlang programs. It was developed to fill a gap in the development process. No tools were available to improve the reliability of applications; testing, no matter how extensive, cannot reveal all bugs, especially those hiding outside the major code paths. Furthermore, the Erlang compiler performs only trivial type checks. Dialyzer uses static analysis to identify discrepancies, such as type errors, unreachable or redundant code and unsafe machine bytecode. It has been used to reveal bugs in many commercial applications, consisting of millions lines of code, bugs that have gone undetected in years of operation and testing.

Dialyzer's ability to analyze Erlang code fast, reconstruct type information, and perform dataflow analysis, has spawned great interest to harness its potential. Such projects include TyPer. An automatic type annotator for Erlang code, expansions for detecting data races and, of course, our project, detection of opaque type violations.

A good knowledge of Dialyzer is necessary as we get in more technical territory. Next we try to explain some fundamental concepts in Dialyzer's operation.

4.1 Success Typings

Dialyzer employes success typings to help with the type annotating of Erlang functions. The following information on success typings has been attained from the paper "Practical Type Inference Based on Success Typings" by Tobias Lindahl and Konstantinos Sagonas [9]. Please refer to it for a more in depth and complete presentation.

4.1.1 Subtyping systems and the need for subtyping in Erlang

Subtyping systems amount to a broader domain than systems based on Hindley-Milner type inference do. Attempts to tailor a static type system on dynamically typed languages have been made with limited success. Because of the way Erlang is written, a type system for Erlang needs to be based on (unrestricted) subtyping. For example, consider the following Erlang function from the Erlang/OTP

```
send(Pg, Mess) when is_atom(Pg) ->
    global:send(Pg, {send, self(), Mess});
send(Pg, Mess) when is_pid(Pg) ->
    Pg ! {send, self(), Mess}.
```

This is a function with two arguments, `Pg` and `Mess`, consisting of two guarded clauses. The first clause handles the case when the `Pg` argument is an atom, the second clause when `Pg` is a process identifier (an Erlang *pid*). When `Pg` is an atom, it denotes a globally registered process and the library function `global:send/2` is used. When it is a *pid*, the Erlang built-in *send* function (denoted '!') is used to send the message (a 3-tuple). Typing this function is not possible with a constructor-based type system such as Hindley-Milner. The first argument needs to be described by unique constructors, one for the *atom* and another for the *pid* primitive type. This function would need to be rewritten to explicitly match these constructors, instead of performing type checks using guards. In short, imposing a Hindley-Milner type system on Erlang requires modifications to existing code and amounts to starting to program in a different language, not in Erlang as we currently know it. Given that it is a language with existing applications often consisting of more than one million lines of code, this is not a viable option. Subtyping is the answer to this problem. By adopting a subtyping system that allows for disjoint union types, we can characterize the first argument as a union of the *atom* and *pid* types. This indicates that the function can be called with any subtype of the union, that is with any specific *atom* or *pid*. In such a scheme, the second argument can then have the type which denotes the set of all terms i.e. the *any()* type.

4.1.2 Basic idea behind success typings

Success typings are not meant for proving type safety — this is already provided by the underlying implementation — nor are they meant for removing dynamic type tests. Their purpose is to capture the biggest set of terms for which it can be deduced that type clashes will definitely occur. A function's success typing is the compliment of this set of terms. A success typing is a type signature that over-approximates the set of types for which the function can evaluate to a value. 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. If the arguments of a function call are in the function domain, the call might succeed, but if they are not, the call will definitely fail. This unusual property gives dialyzer the unique characteristic of not providing false positives, meaning that errors discovered by dialyzer are sure to also cause runtime errors.

4.1.3 Examples

```
foo(Int) when is_integer(Int) ->
    Int + 1;
foo(List) when is_list(List) ->
    erlang:length(List) + 1;
```

The success typing for the function foo is

```
( integer() | list() ) -> integer()
```

This simple example demonstrates how the argument is either an *integer* or a *list*; any other argument will certainly lead to failure. The function always returns an *integer*. More interesting is a case with varied return types.

```
foo2(Int) when is_integer(Int) ->
    Int + 1;
foo2(List) when is_list(List) ->
    list_to_atom(List);
```

The success typing for the function `foo2` is

```
( integer() | list() ) -> (integer() | atom())
```

In this function the argument is, again, either an *integer* or a *list*, the return type is the union of the *integer* and *atom* types. Note that the success typing does not track the dependencies between the input and output types. On the other hand, the information maintained is enough to conclude that, if any other type is requested from the function, an error will manifest.

```
foo3(Int1, Int2) when is_integer(Int1) and is_integer(Int2) ->
    Int1 + Int2;
foo3(List) when is_list(List1) and is_list(List2)->
    list_to_atom(List1 ++ List2);
```

The success typing for the function foo3 is

```
((integer() | list()), (integer() | list()))  -> (integer() | atom())
```

In the preceding paradigm the success typing has overestimated the use of the function. For example the call *foo3(17, [l,i,s,t])*. will never evaluate to a value and will cause the thowing on an exception. Indeed, success typings have their sortcomings. But they will never fail to catch a use of a function or produce an invalid type.

4.1.4 Constraint Generation

Success typings are inferred in two steps. Constraints are generated by traversing the code bottom-up and using derivative rules for guards, pattern matching, case clauses etc. Then, the constraints are solved and the solution constitutes the success typing. Figure 4.1 shows the rules for constraints generation. The rules do not cover the entire Core Erlang.[1] Rules for the `trycatch` and `receive` language constructs are omitted, they can be handled with minor variations as `case` expressions. Also there is no rule for the sequence operators, they can be treated as `let` expressions, where the variable is never used. A represents an environment with bindings of variables of the form: $\{..., x \mapsto \tau_x, ...\}$ and `C` represents nested conjunctions and disjunctions of subtype constraints of the form: $C ::= (T_1 \subseteq T_2)|(C_1 \wedge ... \wedge C_n)|(C_1 \vee ... \vee C_n)$ Equality constraints, $T_1 = T_2$, are used as shorthands for $(T_1 \subseteq T_2) \wedge (T_2 \subseteq T_1)$. The judgment $A \vdash e : \tau, C$ should be read as "given the environment A the expression e has type $Sol(\tau)$ whenever Sol is a solution to the constraints in C".

[1] Core Erlang is the intermediate language that all Erlang programs are translated to by the Erlang/OTP compiler.

$$\frac{}{\{A \cap \{x \mapsto \tau\} \vdash x : \tau, \emptyset} \qquad [\texttt{VAR}]$$

$$\frac{A \vdash e_1 : \tau_1, C_1 ... e_n : \tau_n, C_n}{A \vdash c(e_1, ..., e_n) : c(\tau_1, ..., \tau_n), C_1 \wedge ... \wedge C_n} \qquad [\texttt{STRUCT}]$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \quad A \cup \{x \mapsto \tau_1\} \vdash e : \tau_2, C_2}{A \vdash \texttt{let}\ x = e_1 \text{in}\ e_2 : \tau_2, C_1 \wedge C_2} \qquad [\texttt{LET}]$$

$$\frac{A \cup \{x_i \mapsto \tau_i\} \vdash f_1 : \tau_1^{'}, C_1 ... f_n : \tau_n^{'}, C_n \quad e : \tau, C}{\begin{array}{r} A \vdash \texttt{letrec}\ x_1 = f_1, ..., x_n = f_n \text{ in } e : \tau, C_1 \wedge ... \wedge C_n \wedge C \\ \wedge(\tau_1^{'} = \tau_1) \wedge ... \wedge (\tau_1^{'} = \tau_1) \end{array}} \qquad [\texttt{LETREC}]$$

$$\frac{A \cup \{x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n \vdash e : \tau_e, C}{A \vdash \text{fun}(x_1, ..., xn) \rightarrow e : \tau, (\tau = ((\tau_1, .., \tau n) \rightarrow \tau_e \text{when } C))} \qquad [\texttt{ABS}]$$

$$\frac{A \vdash e_1 : \tau_1, C_1 ... A \vdash e_n : \tau_n, C_n}{\begin{array}{r} A \vdash (e_1(e_2, ..., e_n) : \beta, (\tau_1 = (\alpha_2, ..., \alpha n) \rightarrow \alpha) \wedge (\beta \subseteq \alpha) \\ \wedge(\tau_2 \subseteq \alpha_2) \wedge ... \wedge (\tau_n \subseteq \alpha_n) \wedge C_1 \wedge ... \wedge C_n \end{array}} \qquad [\texttt{APP}]$$

$$\frac{A \vdash p : \tau, C_p \quad A \vdash g : \text{true}, C_g}{A \vdash p \text{ when } g : \tau, C_p \wedge C_g} \qquad [\texttt{PAT}]$$

$$\frac{\begin{array}{r} A \vdash e : \tau, C_e \\ A \cup \{u \mapsto \tau_u \mid u \in \text{Var}(p_1)\} \vdash p_1 : \alpha_1, C_1^p, b_1 : \beta_1, C_1^b \\ ... \\ A \cup \{u \mapsto \tau_u \mid u \in \text{Var}(p_n)\} \vdash p_n : \alpha_n, C_n^p, b_n : \beta_n, C_n^b \end{array}}{\begin{array}{r} A \vdash \texttt{case}\ e \text{ of } p_1 \rightarrow b_1; ... p_n \rightarrow b_n \text{ end } : \beta, C_e \wedge (C_1 \vee ... \vee C_n) \\ \text{where } C_1 = ((\beta = \beta_i) \wedge (\tau_i = \alpha i) \wedge C_i^p \wedge C_i^b) \end{array}} \qquad [\texttt{CASE}]$$

Figure 4.1: Derivation Rules

The VAR, STRUCT and LET rules are standard. Constants can be typed by the STRUCT rule by viewing primitive types as nullary constructors. The ABS rule binds the constraints from the function body to its type, but exports no constraints. In this way the type of a function can be influenced by outer constraints on the free variables, but the constraints from the function body cannot affect the types of the free variables outside the function body. A `letrec` statement binds a number of function declarations to recursion variables. The scope of the recursion variables includes both the function declarations and the body

of the `letrec` statement. The `LETREC` rule assigns fresh type variables to the recursion variables and then adds equality constraints on the function types and the types of the recursion variables. The `PAT` rule slightly abuses notation. The guards in a pattern can be expressed as a conjunction of simple type constraints on variables such as `is_integer(X)`, `is_atom(X)`, etc. and by using equality constraints on variables. The translation of these into constraints on types and type variables is straightforward and omitted for brevity. The rule states that the guard must evaluate to true under the translated constraints, which is equivalent to stating that the constraints must have a solution. In `case` expressions, it is enough that one clause can be taken in order for the whole expression to have a success typing. This is captured by introducing a disjunction of constraints in the `CASE` rule. Each disjunction contains the constraints that need to be satisfied for the corresponding clause to contribute to the success typing. Intuitively, if a clause is taken at runtime, the type of each incoming argument and the corresponding pattern must be equal, and the constraints from the clause guard must be satisfied. The type of the whole case expression equals the type of the clause body. Finally, note that the `APP` rule is quite unorthodox. In traditional subtyping systems the type of an application is downwards bounded by the type of the functions range. This ensures that all possible return values are handled, possibly by inserting narrowers to make it a smaller type.

4.1.5 Constraint Solving

The second and final stage of infering a success typing, is solving the constraints generated in the first step. According to the derivation rules, disjuctions are only produced by the `CASE` rule. The constraints are kept in general form and not transformed into disjuctive general form. Such a transformation would cause the number of constraints to explode in the presence of several case expressions. Lets assume that Sol is the mapping from type expressions and type variables to concrete types. Concrete types include all type expressions with the exception of constraints and type variables. Sol is the solution to constraint set C ($Sol \models C$) if:

$$
\begin{aligned}
&Sol \models T_1 \subseteq T_2 \Longleftrightarrow none() \subset Sol(T_1) \subseteq Sol(T_2)\\
&Sol \models C_1 \wedge C_2 \Longleftrightarrow Sol \models C_1, Sol \models C_2\\
&Sol \models C_1 \vee C2 \Longleftrightarrow \begin{cases} Sol_1 \models C_1, Sol_2 \models C_2 \\ Sol = Sol_1 \sqcup Sol_2 \end{cases}
\end{aligned}
$$

where $Sol_1 \sqcup Sol_2$ denotes the point-wise least upper bound of the solutions. In words: a solution satisfies a subtype constraint if the mapping satisfies the subtype constraint and neither of its constituents is *none()*. A solution of a conjunction of constraints must satisfy all conjunctive parts and a solution to a disjunction of constraints is the point-wise least upper bound of the solutions of all disjuncts. Furthermore, if a constraint set has no solution, it can be assigned the solution $\perp$ which represents a solution that maps all type expressions to *none()*. Note that $\perp \sqcup Sol = Sol$. So, as long as the set of constraints from one clause in a case expression has a solution, other than $\perp$, the constraints from the whole case expression also have a solution other than $\perp$.

The constraint solver is written in Erlang. Type constraints are generated and solved at the granularity of a single function according to the algorithm in Figure 4.2. The basic

$$
\begin{aligned}
solve(\bot, \_) &= \bot \\
solve(Sol, \alpha \subseteq \beta) &= \begin{cases} Sol & \text{when } Sol(\alpha) \subseteq Sol(\beta) \\ Sol[\alpha \mapsto T] & \text{when } Sol(\alpha) \sqcap Sol(\beta) \neq \text{none}() \\ Sol[\alpha \mapsto T] & \text{when } Sol(\alpha) \sqcap Sol(\beta) = \text{none}() \end{cases} \\
solve(Sol, Conj) &= \begin{cases} Sol & \text{when } solve\_conj(solve(Sol, Conj)) = Sol \\ solve(Sol', Conj) & \text{when } solve\_conj(solve(Sol', Conj)) \neq Sol \end{cases} \\
solve(Sol, Disj) &= \begin{cases} \bigsqcup Sol' & \text{when } Sol' \neq \emptyset \\ \bot & \text{when } Sol' = \emptyset \end{cases} \text{where } \begin{cases} Sol' = \{S \mid S \in PS, S \neq \bot\} \\ PS = \{solve(Sol, C) \mid C \in Disj\} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
solve\_conj(\bot, \_) &= \bot \\
solve\_conj(Sol, C_1 \wedge ... \wedge C_n) &= solve_conj(solve\_conj(Sol, C_1), C_2 \wedge ... \wedge C_n) \\
solve\_conj(Sol, C) &= solve(Sol, C)
\end{aligned}
$$

Figure 4.2: Algorithm for solving constraints

idea is to iteratively solve all constraints in a conjunction until either a fixpoint is reached or the algorithm encounters some type clash and fails by assigning the type *none()* to a type expression. The starting point for *Sol* is a mapping where all type expressions are mapped to *any()*, with the exception of the types of all recursion variables that are mapped to *none()*.

4.2 Dataflow Analysis

Annotation of the functions with success typing is supersided by the dataflow analysis. The dataflow analysis is a top-down traversal of the Core Erlang code. It serves two purposes: detecting discrepancies in the code and refining of the success typings.

4.2.1 Detection of type clashes

The dataflow pass incorporates the inferred function signatures and uses them to warn about program points, where type clashes can occur. As a result of the "type signature" pass, all possible return values and arguments of call points are known. Types of explicit type constructions are also easy to compute. With the above type information, Dialyzer is in a position to detect the precise location and the specific nature of type clashes. For example, comparing the success typing of a function, with the actual arguments used in calling it, can reveal if the call is going to fail.

```
foo(List) -> List ++ "\n".

goo() -> foo(hello).
```

The success typings of the functions are:

```
foo(list()) -> list()

goo() -> none().
```

Dialyzer produces the warning:

```
The call mod:foo('hello') will never return since it differs
in argument position 1 from the success typing arguments: ([any()])
```

The allowed argument type for `foo/1` is only the *list()* type because of the texttt++ operator. It is being called with an atom argument by `goo/0` and it will definitely fail to execute. Note that the return type for `goo/0` is *none()*. This is after all normal, there are no disjunctions in the functions and the call `foo(hello)` will not return.

```
hoo(X) when is_atom(X) ->
   case X of
        zong -> ok;
        gazong -> norm;
        {error,_} -> error
   end
```

Dialyzer produces the warning:

```
The pattern {'error', _} can never match the type atom()
```

In this instance `hoo/1` will not fail to execute but contains unreachable code. The dataflow analysis deduces that `X` is of type *atom()* because of the `is_atom` guard. Without a doubt, the case {`error,_`} can not conceivably match with `X`.

In general, the dataflow pass can reveal the following defects:

- Ill typed function arguments.
- Unused functions.
- Functions that will never return or will terminate with explicit exceptions.
- Type tests that can never succeed.
- Guard tests that can never succeed.
- Invalid binary constructions.
- Patterns that will never match.
- Patterns that are completely covered by previous patterns.
- Invalid or overlapping contracts.

4.2.2 Refined success typings

Inference of the typings is done without any user input. As a result, dialyzer can be used immediately, on any piece of Erlang code, without any modifications. Commonly, success typings, due to their nature, can become overly general; this might appear weak and compromising the power of the analysis. However, this is not the case, thanks to refined success typings. Very helpful to this cause is Erlang's module system; all functions must be defined in modules and only a few functions are exported for use from other modules. This means that all the call locations of non-exported functions are known and limited inside the modules. Detection of the call locations leads to the identification of all possible argument types, which in turn leads to smaller, more precise typings —- these are referred to as refined success typings.

```
-module(test).
-export([foo/1]).

add1(X) -> X+1.

foo(X) when is_integer(X) -> add1(X);
foo(X) -> Y = round(X), add1(Y).
```

This example showcases how refinement of success typing works. If both `foo/1` and `add1/1` were exported the typings would be:

```
add1(number()) -> number().
foo(number()) -> number().
```

If the `add1/1` function is not exported, then dialyzer knows that it is only being called by `foo/1`. Exclusively *integer()* arguments are used on `add1/1`. Using this new information, the success typings can be recalculated with extra constraints. The result is:

```
add1(integer()) -> integer().
foo(number()) -> integer().
```

Observe that refinement does not effect only the annotation of `add1/1`. The "smaller" success typing induces a smaller success typing for `foo` also.

4.3 Contracts

Refining of success typing can also be performed by programmers via contracts or "specs" for functions. Contracts are Haskell-like type definitions, that are commonly used to capture the programmers' intent and can be used to assist the analysis and help refine exported functions.

```
swap({X,Y}) -> {Y,X}.
```

This function, for instance, is placing no restriction on its variables. It has a success typing of:

```
swap({any(),any()}) -> {any(),any()}.
```

If the textttswap function is bound for use solely with an *atom()* and *number()* tuple, that fact can be declared in a contract.

```
-spec swap({atom(),number()}) -> {number(), atom()}.
swap({X,Y}) -> {Y,X}.
```

The use of the contract allows the success typings to represent more closely the true goals of a function. As a result the analysis is strengthened. Furthermore contracts are a useful way of assuring that a function is indeed doing what it was intended to. If a contract is not satisfied, Dialyzer indicates that a function is not conforming to its predefined purpose.

```
-spec check(number()) -> {'ok'|'error', number()}.
check(X) ->
    case X > 34 of
        true -> ok;
        false -> error
    end.

Invalid type specification for function module:check/1.
The success typing is (_) -> 'error' | 'ok'
```

`Check/1` is described by the contract as a function that will return a tuple, the implementation however differs. Either the contract or the function is incorrect. Nonetheless, the use of the contract will reveal to the developer an inconsistency in his/her logic.

4.4 Strongly Connected Components

Inference of the success typings doesn't happen in a per function basis as one might imagine, instead one strongly connected component of the callgraph is analyzed each time. Strongly connected components or SCCs consist of functions that call each other recursively, making it possible to reach any function in the component and then return to the starting point. This is a common occurrence in Erlang, even between functions of different modules. Nonetheless, these functions must be analyzed as a single entity in order to get the best possible success typing. It's also worth noting that SCCs that belong in disconnected callgraphs can be analyzed in parallel, making Dialyzer a strong candidate for a concurrent implementation; this work is already underway. One place where SCCs usually appear is in servers that loop perpetually and handle messages.

loop/1 and do_something/1 form a SCC

```
loop(Pid) ->
    receive
        {Pid, Msg} -> do_something(Msg)
    end.

do_something(Msg) ->
    Pid = module:getPid(Msg),
    ....
    loop(Pid).
```

4.5 Persistent Lookup Table

Although dialyzer is fairly fast most applications use large parts of Erlang's libraries. Analyzing static code every time would make dialyzer cumbersome and counterproductive. To speed up analysis runs, a number of fixed modules that remain unaltered can be defined, analyzed and stored on a persistent lookup table (PLT). The PLT is stored in the users home directory and contains type information for commonly used library modules, thus speeding up the analysis manyfold.

4.6 Dialyzer Operation Summary

Understanding of the inner workings of dialyzer was paramount in this endeavor. All the analysis steps had to be identified and altered to accommodate opaque types, without disturbing the normal operation of the program. This is a brief rundown of these steps.

1. First, the modules to be analyzed are specified by the user; and type and contract information is extracted from them.
2. The static callgraph is created.
3. Strongly connected components are identified and stored topographically in a code-server.
4. The SCCs are analyzed bottom-up using constraint based analysis; the success typings are created.
5. The SCCs are subjected to a top-down dataflow analysis. At this point discrepancies are identified and marked. The signatures of the internal functions are also refined.
6. If no refinements have been made to the signatures we have reached a fixpoint. Otherwise, the analysis is repeated from step 4 until a fixpoint is reached.

Chapter 5

Dialyzer Modifications

5.1 Institution of Opaque Types in Dialyzer

As it was mentioned earlier, for the purpose of detecting opaque violations we utilized Dialyzer. For Dialyzer to be able to perform type checking on opaque types, they had to be introduced in the type system. All the type information and type interactions, which are used by Dialyzer, exist in the `erl_types` module. Opaque types are defined there as a new primitive type, without any subtyping relations with the rest of the Erlang types, much like Erlang's identifier types (ports, references and pids). Figure 5.1 depicts the types used in Dialyzer. During the analysis, opaque types are embodied by records with 4 fields.

- The name of the module the opaque type is defined in.
- The name of the opaque type.
- A list of arguments that can be used in the future to implement polymorphic opaque types.
- The internal representation of the types.

```
-record(opaque,  {
        mod :: module(),
        name :: atom(),
        args = [],
        struct :: erl_type()}).
```

Multiple opaque types are grouped together in an ordered set, but no attempt is made to join them further, as is done on number ranges or tuples of the same arity.

With the representation finalized, a plethora of helper functions was implemented, setters, getters and pretty printers. The most important operation though is, by far, the one that defines the interaction of opaque types with the rest of the Erlang types. This behavior is determined mainly by the function `t_inf`, which is used to calculate the intersection of two given type arguments. Searching for a subtype, it's abundantly clear that any comparison between an opaque type and any other type must yield none();

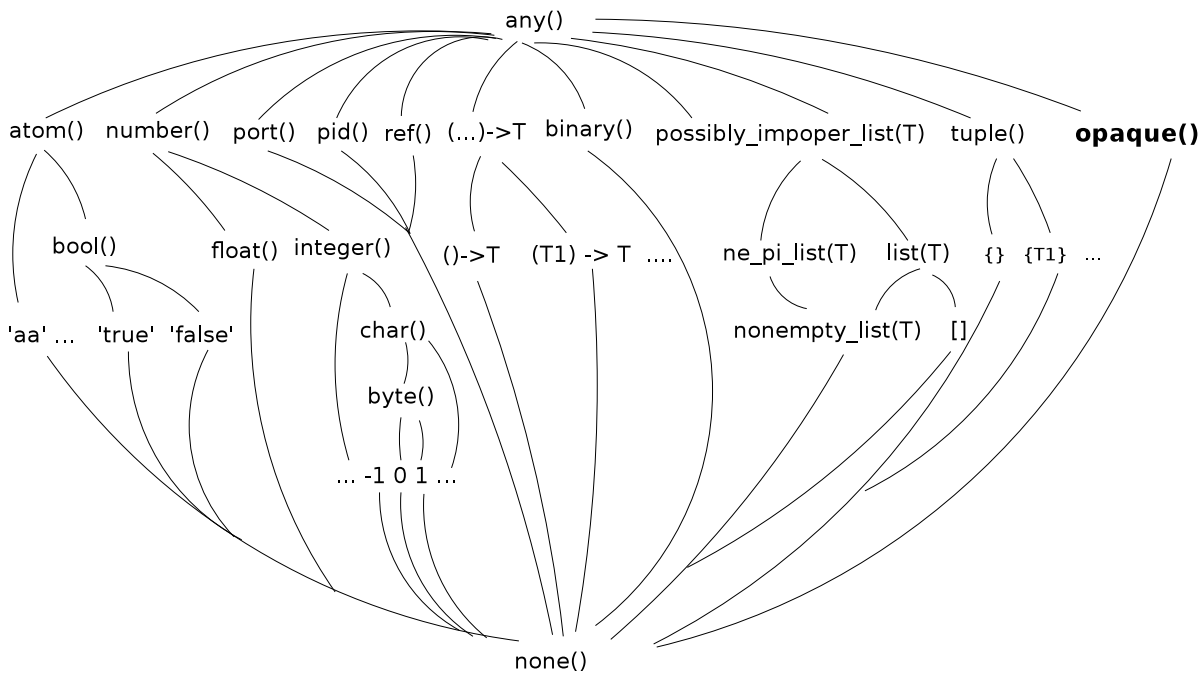

Figure 5.1: Type lattice

meaning that the types are foreign and it is not possible for them to have any association. Then again, we must consider the possibility that an opaque type is used in the module it has been defined in. In this case, the comparison must be allowed for the analysis to yield correct results. Hence, in such an occurrence, the internal representation is used to determine the subtyping relation and the opaque type is always considered to be the infimum, so it can get cascaded to the success typings. To achieve this split behavior we introduced an extra argument that makes the function aware of the behavior required.

```
t_inf(integer(), float()) = none()
t_inf(number(), float()) = float()
%% -opaque queue() :: list(any())
%% inside the queue module
t_inf(queue(), list(any())) = queue()
t_inf(queue(), list(atom()) = queue()
%% outside the queue module
t_inf(queue(), list()) = none()
```

The above example illustrates the behaviour of `t_inf` in different scenarios. The fourth example demostrates how the actual infimum can be overwritten inside an opaque module by the opaque type. This, as stated previously, is done to preserve the opaqueness of the exported signatures. It is possible that this action might compromise the type checking by returning the more general types and not the true infimum. As a result, we urge programmers, when defining opaque types, to provide detailed typings and, if possible, to define opaque types as typed records, which are easily recognizable in the analysis and their type cannot be overestimated.

Another important function is `t_sup`; its purpose is to determine the union of two types. In contrast with `t_inf`, here we have been more lenient. We have allowed type unions which contain opaque types. This is something that is not normally permitted, but it was deemed necessary to suit Erlang's dynamic nature.

5.2 Opaque Type Declarations

Dialyzer uses type declaration to alias complex structured types to simple type names.

```
-type TYPE_NAME :: TYPE_STRUCTURE.
```

In order to have explicit opaque type declaration, we modified `erl_lint`, the Erlang code parser, so that it will recognize the following construct:

```
-opaque TYPE_NAME :: TYPE_STRUCTURE.
```

We have already defined the following Erlang standard library types as builtin opaque types:

Array()

Functional, extendible arrays. Arrays can have fixed size, or can grow automatically as needed. A default value is used for entries that have not been explicitly set.

Arrays uses zero based indexing. This is a deliberate design choice and differs from other Erlang datastructures, e.g. tuples

```
-record(array, {size :: non_neg_integer(),
                max  :: non_neg_integer(),
                default,
                elements
               }).
-opaque array() :: #array{}.
```

Dict()

Dict implements a Key - Value dictionary. The representation of a dictionary is not defined.

```
-record(dict,
        {size=0               :: non_neg_integer(),
         n=?seg_size          :: non_neg_integer(),
         maxn=?seg_size       :: non_neg_integer(),
         bso=?seg_size div 2  :: non_neg_integer(),
         exp_size=?exp_size   :: non_neg_integer(),
         con_size=?con_size   :: non_neg_integer(),
         empty                :: tuple(),
         segs                 :: tuple()
        }).
-opaque dict() :: #dict{}.
```

Digraph()

The digraph module implements a version of labeled directed graphs. What makes the graphs implemented here non-proper directed graphs is that multiple edges between vertices are allowed. However, the customary definition of directed graphs will be used in the text that follows.

A directed graph (or just "digraph") is a pair (V, E) of a finite set V of vertices and a finite set E of directed edges (or just "edges"). The set of edges E is a subset of $V \times V$ (the Cartesian product of V with itself). In this module, V is allowed to be empty; the so obtained unique digraph is called the empty digraph. Both vertices and edges are represented by unique Erlang terms.

```
-record(dict,
-record(digraph, {vtab = notable :: ets:tab(),
                  etab = notable :: ets:tab(),
                  ntab = notable :: ets:tab(),
                  cyclic = true  :: boolean()}).
-opaque digraph() :: #digraph{}.
```

Gb_set()

An implementation of ordered sets using General Balanced Trees. This can be much more efficient than using ordered lists, for larger sets, but depends on the application.

```
-type gb_set_node() :: 'nil' | {term(), _, _}.
-opaque gb_set() :: {non_neg_integer(), gb_set_node()}.
```

Gb_tree()

An efficient implementation of General Balanced Trees. These have no storage overhead compared to unbalaced binary trees, and their performance is in general better than AVL trees.

```
-type gb_tree_node() :: 'nil' | {_, _, _, _}.
-opaque gb_tree() :: {non_neg_integer(), gb_tree_node()}.
```

Queue()

This module implements (double ended) FIFO queues in an efficient manner. The data representing a queue as used by this module should be regarded as opaque by other modules. Any code assuming knowledge of the format is running on thin ice.

```
-opaque queue() :: {list(), list()}.
```

Set()

Sets are collections of elements with no duplicate elements. The representation of a set is not defined.

```
-record(set,
        {size=0              :: non_neg_integer(),
         n=?seg_size         :: non_neg_integer(),
         maxn=?seg_size      :: pos_integer(),
         bso=?seg_size div 2 :: non_neg_integer(),
         exp_size=?exp_size  :: non_neg_integer(),
         con_size=?con_size  :: non_neg_integer(),
         empty               :: seg(),
         segs                :: segs()
        }).
-opaque set() :: #set{}.
```

Timer()

This module provides useful functions related to time. Unless otherwise stated, time is always measured in milliseconds. All timer functions return immediately, regardless of work carried out by another process.

```
-opaque tref() :: any().
```

5.3 Opaque Type Assignment

Of course, declaring an opaque type doesn't mean anything without the means to identify where the type is used. For that purpose we use contracts. Contracts specify the type of the arguments and also the range of a function. The programmer is required to provide contracts for all the exported functions in the form of the following statement:

```
-spec FUN_NAME(ARG_TYPE_1,...,ARG_TYPE_N) -> RET_TYPE.
```

Figure 5.2 illustrates exemplary contracts for a hypothetical point ADT. Suffice to say we expect the programmer to provide the correct opaque type information. The only check we can make is that the structured type we get from the structured type analysis matches the internal representation of the opaque type.

The first approach that was used to apply the contracts' opaque type information on the functions, was to wait after the analysis had produced the type signature, meaning that the exact type information of the function has been deduced. This methodology was straightforward and worked for most simple cases, but that was not successful when we came across strongly connected components. SCCs consist of functions that call each other recursively; as means to get the correct type information, they need to be analyzed together as one unit. Therefore, it is necessary to also apply the contracts during the dataflow analysis of a function. More specifically the opaque information is applied at the call points of each function.

```
-module(point).
-export([new/2, rotate/2, getCoordinates/1]).

-opaque point() :: {float(), float()}.

%% The specs indicate which arguments and return values
%% should be treated as opaque. This knowledge is crucial.
%% Specs can also be located directly above their corresponding funtions
-spec new(float(), float()) -> point().
-spec rotate(point(), float()) -> point().
-spec getCoordinates(point()) -> {float(), float()}.
-spec azimuth(float(),float()) -> float().

new(X, Y) ->
    R = math:sqrt(X*X+Y*Y),
    A = azimuth(X,Y),
    {R, A}.

rotate({R,A}, Rad) -> {R,A + Rad}.

getCoordinates({R,A}) ->
    X = R*math:cos(A),
    Y = R*math:sin(A),
    {X,Y}.

azimuth(X,Y) ->
    math:atan(Y/X).
```

Figure 5.2: Example Declaration of a Point opaque type in Erlang

In addition to using contract specifications to denote opaqueness, we wanted a method that could deduce the opaqueness of a type automatically, without requiring any programmer input, other than the opaque type declaration. In that direction, if an opaque type is declared with an internal representation that has a unique constructor — such as a record or a specific atom — during the analysis, this constructor is intercepted and the result is altered from a structured type to an opaque type instead. This allows us to have opaque modules that do not require from the programmer to provide signatures for all the exported functions. Such a module is `dict.erl`, which uses a record (dict) for the internal representation. We can rewrite the point module example (Figure 5.2) and use a record as a data structure (Figure 5.3). The opaqueness of the module will be deduced only by the declaration of the point opaque type.

In Dialyzer the point type is completely opaque. The inferred signatures shown in Figure 5.4 are identical to the contracts that we manually defined in Figure 5.2. That goes to show how effective and painless the recognition of opaque types is, if explicit constructors are used to represent them.

```
-module(point).
-export([new/2, rotate/2, getCoordinates/1]).

-record(point,
        {radius = 0 :: number(),
         azimuth = 0 :: number()
        }).

%% Opaque type declaration
-opaque point() :: #point{}.

%% No need for contracts in this version.
%% The #point{} record costructor is tracked
%% through the application and is marked as opaque.

new(X, Y) ->
    R = math:sqrt(X*X+Y*Y),
    A = azimuth(X,Y),
    #point{radius = R, azimuth = A}.

rotate(Point = #point{azimuth = A}, Rad) ->
    Point#point{azimuth = A+Rad}.

getCoordinates(#point{radius = R, azimuth = A}) ->
    X = R*math:cos(A),
    Y = R*math:sin(A),
    {X,Y}.

azimuth(X,Y) ->
    math:atan(Y/X).
```

Figure 5.3: Example of the automatic opaque type detection

```
-spec new(number(),number()) -> point:point().
-spec rotate(point:point(),number()) -> point:point().
-spec getCoordinates(point:point()) -> {float(),float()}.
-spec azimuth(number(),number()) -> float().
```

Figure 5.4: Type signatures for the point module

5.4 Remote Types

Until now we have focused mainly on modules that define and implement opaque types. Slowly we will move outside the modules, were the structure of the types is no longer in scope. When an opaque type is used, the program shouldn't have access to the opaque

type's internal representation. The type can be used as an argument or as a return value, as long as the interface functions are not circumvented. The need is created to use opaque types in contracts outside their native module. For instance, the function `newline/0` needs such a contract.

```
-module(line).
%% Hypothetical Contract:
%% -spec newline() -> {point(),point()}
newline() ->
    A = point:new(0,0),
    B = point:new(1,1),
    {A,B}.
```

The contract that is commented out in the example is not valid unless we define the *point()* type inside the `line` module. Designating the structure of an opaque type in every module that it is used, is not only ineffective but also contradicts the type's opaque nature. To solve this problem we implemented remote types. The ground work for remote types was already in place but it was never completed. A remote type uses first the module name and then the type name to describe a certain type. Dialyzer then refers to the designated module for the type definition, instead of looking in the current module. By adopting remote types we can rewrite `newline/0` with a proper contract.

```
-module(line).

-spec newline() -> {point:point(), point:point()}
newline() ->
    A = point:new(0,0),
    B = point:new(1,1),
    {A,B}.
```

With the use of remote types specific precautions have to be taken. For example, we check that the referred modules and types actually exist. And most importantly we make sure that types do not create definition cycles which could put Dialyzer in an infinite type inference loop.

5.5 Detection of Violations

After the deduction of the type signatures with the help of opaque constraints, the dataflow analysis takes place. Using the inferred typings we are able to detect several opaqueness breaches. The dataflow pass recognizes irregularities; locations where opaque types come in contact with structured types in an erroneous manner. We process these abnormalities and produce a warning that describes the situation and can be easily comprehended. The defects Dialyzer can detect are listed in the next section.

5.5.1 Violation Categories

Use of an opaque type where a structured type is expected. The success typing of a function may indicate that it receives only structured arguments of some kind. If an

opaque type argument is used instead, it will be treated by the function as if it was of the type indicated in the type signature; thus ignoring the argument's opaque attribute and violating its opaqueness.

Use of a structured type where an opaque type is expected.

These defects refer to structured types being used with the interface functions of opaque types.

Inspecting the type of an opaque type via a type test. Type tests are builtin Erlang functions. Type tests have `(any()) -> boolean().` signatures which means they accept any type of argument. Even by their name we can deduce that they are type inspecting functions. Therefore, opaque types must be excluded as possible arguments. The type tests we check for are the following:

`is_atom/1`	`is_boolean/1`	`is_port/1`
`is_binary/1`	`is_bitstring/1`	`is_reference/1`
`is_float/1`	`is_function/1`	`is_tuple/1`
`is_integer/1`	`is_list/1`	
`is_number/1`	`is_pid/1`	

Running a guard test on an opaque type. Guard tests are used to control the program flow based on the structure of their argument. In fact, guards use the same type tests mentioned above, as well as boolean and term comparison expressions. Thus, guard tests on opaque types are warned against by Dialyzer, for the same reasons.

Equality and inequality tests. Equality and inequality tests between opaque and structured types also constitute opaque type violations.

Matching of opaque types against patterns. Patterns are used for controlling the program flow based on the type structure and for deconstructing types. Both uses are improper when used on opaque types.

5.5.2 Violation detection examples

Consider the opaque type `my_queue`:

```
-module(my_queue).
-export([new/0, add/2]).
-opaque my_queue() :: list().

-spec new() -> my_queue().
new() -> [].

-spec add(term(), my_queue()) -> my_queue().
add(E, Q) -> Q ++ [E].
```

The concrete type used is a list and we have defined two interface functions for this example.

When comparing a function's success typing with the actual arguments used in its call locations, we recognize three types of errors. The first one is calling a function that expects a structured type as an argument with an opaque type argument. The second is using a structured type in the place of an opaque argument. Finally the last errors detected in this fashion are type tests on opaque types; type tests by default expect any argument, therefore violations are not caught by previous checks. Special care is taken to warn when the structure of opaque types is probed by type tests. These three different cases are demonstrated below.

- ```
  wrong1() ->
      length(my_queue:new()).
  ```

  The `erlang:length` function is used to count the elements of a list and has a success typing:

  ```
  length([any]) -> non_neg_integer().
  ```

  In other words it expects a list argument. Calling it with an opaque type will not cause any runtime errors, but is a clear violation of the opaqueness of the queue abstract data type. Dialyzer will warn us with the message below:

  ```
  The call erlang:length(my_queue:my_queue())
  contains an opaque term in position 1
  when a structured term of type [any()] is expected
  ```

- ```
  wrong2() ->
      my_queue:add('element', []).
  ```

  In this example we have the opposite case: the programmer has tried to use the interface to add an element to the list, but has used a structured type (empty list) to initialize the queue. There is no guarantee about the representation of the queue, thus construction of opaque types outside the ADT module can lead to problems. This is the warning we are presented with:

  ```
  The call my_queue:add('element',[]) does not have
  an opaque term of type my_queue:my_queue() in position 2
  ```

- ```
  wrong3() ->
      is_list(my_queue:new()).
  ```

  Before the introduction of opaque types, type tests were allowed on any type. This has changed and type tests are permitted only on structured types.

  ```
  The type test is_list(my_queue:my_queue())
  breaks the opaqueness of the term my_queue:my_queue()
  ```
  ```

Similar to the type test violations are the guard violations. In the current erlang version, only erlang's builtin functions can be used as guards. All the builtin functions are intended for testing structured types, therefore guard tests on opaque types are warned against.

- ```
  wrong4() ->
      case (my_queue:new()) of
          X when is_list(X) -> 'something'
       end.
  ```

  ```
  Guard test is_list(X::my_queue:my_queue())
  breaks the opaqueness of its argument
  ```

Another possible erroneous behavior that Dialyzer was modified to detect is equality and inequality tests. Such tests must be permitted among opaque types of the same kind, but should raise warnings when comparing opaque with structured types.

- ```
  ok() ->
      A = my_queue:new(),
      B = my_queue:new(),
      A =:= B.
  ```

  The above is acceptable behavior since both A and B are of the same type. On the other hand the following are incorrect.

- ```
  wrong5() ->
      [] =:= my_queue:new().
  ```

  ```
  Attempt to test for equality between a term of
  opaque type my_queue:my_queue() and a term of type []
  ```

- ```
  wrong6()->
      Queue = my_queue:new(),
      Queue1 = my_queue:add('element', Queue),
      case Queue1 =/= [] of
          true  -> ok;
          false -> weird
      end.
  ```

  ```
  Attempt to test for inequality between a term of type []
  and a term of opaque type my_queue:my_queue()
  ```

The last warning that Dialyzer can give on opaque types is about improper pattern matching attempts. This is a very common occurrence since pattern matching is widely used in erlang programs.

- ```
  wrong7(X) ->
      ADT = my_queue_adt:add('buddy',X),
      case ADT of
          ['buddy'|_] -> module:buddy(ADT);
          ADT -> module:stranger(ADT)
      end.
  ```

  ```
  The attempt to match a term of type my_queue:my_queue()
  against the pattern ['buddy' | _] breaks the opaqueness of the term
  ```

The developer uses pattern matching to pick at the head of the queue and see if his buddy is first in line. This pattern tries to directly match with the opaque type. Patterns however can get more complicated with multiple nested structures, which might not all match directly to an opaque type. In order to identify if the pattern does not match because of an opaque type or because it is just incorrect, we perform an exhaustive comparison between pattern and matching type. Thus we can pinpoint exactly why the pattern matching has failed. Here is an example:

- ```
  wrong8() ->
      ADT = my_queue:new(),
      Pattern = {ok,{adt, ADT}},
      case Pattern of
          {ok,{adt, []}} -> 'empty'
          {ok,{Queue}} -> Queue;
      end.
  ```

In the first clause the programmer incorrectly tries to match the queue abstract data type against an empty list to discern if it is empty. He is warned by Dialyzer that the opaqueness of the term is compromised.

```
The attempt to match a term of type  {'ok',{'adt',my_queue:my_queue()}}
against the pattern {'ok', {'adt', []}} breaks the opaqueness of my_queue:my_queue()
```

In the second case clause it seems that the programmer has forgotten that he is expecting an {`adt, Queue`} pattern. He just uses the variable Queue assuming that it will bind with the queue abstract data type. This is not of course an opaque type violation but rather an error that will prohibit the pattern from matching. Therefore the warning is of a different nature.

```
The pattern {'ok', Queue} can never match the type
{'ok',{'adt',my_queue:my_queue()}}
```

5.5.3 Deficiencies

The analysis employed is sound for defect detection, meaning that it produces no false positives but it does not guarantiee that all opaque violations can be unearthed. Our efforts are not focused at systematically proving or disproving opaque type violations, but rather at detecting as many as possible, using success typings and dataflow analysis.

There are still obvious violations that we fail to catch. Such a violation is the use of opaque arguments in certain type inspecting functions. Type inspecting is a function that uses type checks, guards and pattern matching on its arguments.

```
foo(X) when is_atom(X) -> 'atom';
foo({_,_}) -> 'tuple2'.

goo(X) -> {shazam, X, X}.
```

`foo/1` is a structure inspecting function where as `goo/1` is not. `foo's` success typing is:

```
foo(atom()|{any(),any()}) -> 'atom' | 'tuple2'.
```

Signatures such as this do not pose a problem. By comparing the signature with the function's argument type we can ascertain the validity of the argument. Unfortunately, clear success typings as the above are not always the case. The structure inspecting attribute is not always reflected on the success typing. For example, by adding an extra clause to `foo/1` we can dramatically change the type signature and throw off the analysis.

```
foo(X) when is_atom(X) -> 'atom';
foo({_,_}) -> 'tuple2';
foo(_) -> 'error'.
```

The success typing now changes to:

```
foo(any()) -> 'atom' | 'tuple2' | 'error'.
```

The signature suggests that any argument is acceptable by the function — which is of course true in Erlang's scope but not in the scope of our analysis. The *any()* type masks potential violations which can go undetected. We are planning to extend the analysis and keep information about structure inspecting functions, in order to overcome this problem. Dialyzer is always expanding, we are improving its functionality continuously in the hope of making the analysis stronger and detecting even more software defects.

Chapter 6

Operation Walkthrough

Having explained Dialyzer's operation and the modifications made, we will demostrate how all the steps tie together using an example for our test suite.

6.1 The Opaque Data Type Module

The process starts with a module. The module `rec_adt` implements the abstract data type `rec`. The `rec` ADT is used to store an atom and an integer, the data structured utilized to do this is a record with two fields. For each field we have a setter and a getter interface function.

```
-module(rec_adt).

-export([new/0, get_a/1, get_b/1, set_a/2, set_b/2]).

-record(rec, {a :: atom(), b = 0 :: integer()}).

new() -> #rec{a = gazonk, b = 42}.

get_a(#rec{a = A}) -> A.

get_b(#rec{b = B}) -> B.

set_a(R, A) -> R#rec{a = A}.

set_b(R, B) -> R#rec{b = B}.
```

If we wish to analyze the `rec_adt` as an opaque type implementing module, then we have to declare the `#rec` record as an opaque type and provide signatures for the interface functions. Note that in this example, it is not necessary to present contracts for the functions; Dialyzer can automatically ascertain the opaqueness of a term by detecting the record constructor. However, it is always better to use contracts on functions, even for documentation and clarification purposes only. The opaque `rec_adt` module will look like this:

```
-module(rec_adt).

-export([new/0, get_a/1, get_b/1, set_a/2, set_b/2]).

-record(rec, {a :: atom(), b = 0 :: integer()}).

-opaque rec() :: #rec{}.

-spec new() -> rec().
new() -> #rec{a = gazonk, b = 42}.

-spec get_a(rec()) -> atom().
get_a(#rec{a = A}) -> A.

-spec get_b(rec()) -> integer().
get_b(#rec{b = B}) -> B.

-spec set_a(rec(), atom()) -> rec().
set_a(R, A) -> R#rec{a = A}.

-spec set_b(rec(), integer()) -> rec().
set_b(R, B) -> R#rec{b = B}.
```

6.2 The Module To Be Analyzed for Violations

Besides the `rec_adt` module, another module is needed, a module that employes the *rec()* opaque type and in which opaque type violations may be present. For this purpose we define the `rec_use` module:

```
-module(rec_use).

ok1() ->
    rec_adt:set_a(rec_adt:new(), foo).

ok2() ->
    R1 = rec_adt:new(),
    B1 = rec_adt:get_b(R1),
    R2 = rec_adt:set_b(R1, 42),
    B2 = rec_adt:get_b(R2),
    B1 =:= B2.

wrong1() ->
    case rec_adt:new() of
        {rec, _, 42} -> weird1;
        R when tuple_size(R) =:= 3 -> weird2
    end.
```

```
wrong2() ->
    R = list_to_tuple([rec, a, 42]),
    rec_adt:get_a(R).
```

6.3 Analysis Preprocessing

The analysis is invoked from the command line:

```
dialyzer -c rec_adt.erl rec_use.erl
```

The Erlang source code is compiled in to Erlang Core code. The core code is traversed; type, opaque type, record, and contract information is extracted and stored for later use. Then a callgraph is created to determine the direction of the analysis. In our example the callgraph looks something like Figure 6.1.

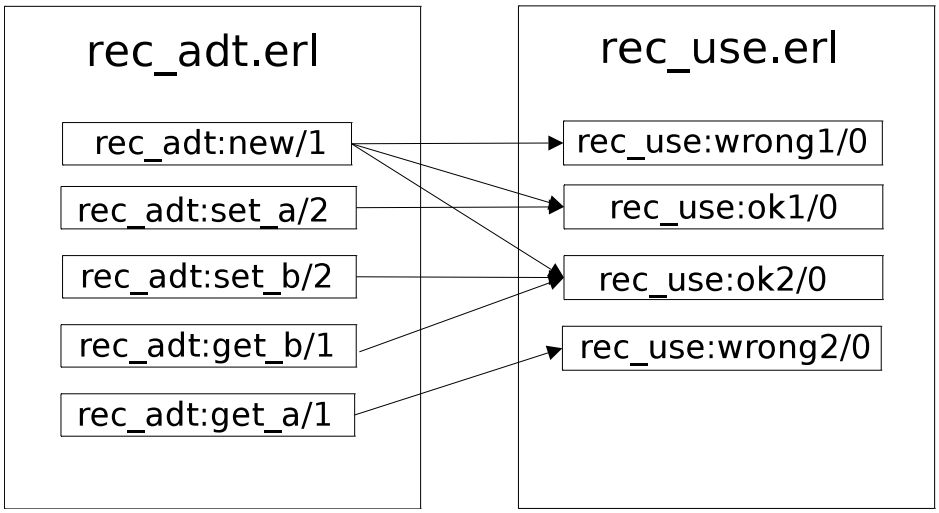

Figure 6.1: Callgraph

6.4 Success Typing Inference

The graph is acyclic, that means that we have no strongly connected components to worry about. The program flows from the `rec_adt` towards `rec_use`. Thus the analysis must start from the `rec_adt` functions, when their success typings are inferred, the `rec_use` functions can then be analyzed, using the previously inferred signatures.

The constraints are generated for the `rec_adt` functions and then they are solved to produce success typings. Normally we would get success typings with structures types like the following:

```
-spec new() -> #rec{a::'gazonk',b::42}.
-spec get_a(#rec{}) -> atom().
-spec get_b(#rec{}) -> integer().
-spec set_a(#rec{},atom()) -> #rec{}.
-spec set_b(#rec{},integer()) -> #rec{}.
```

The typings produced now are different. The opaque declarations act as extra constraint and cause the success typings of the *rec()* abstract data type functions to reflect their opaque characteristics.

```
-spec new() -> rec().
-spec get_a(rec()) -> atom().
-spec get_b(rec()) -> integer().
-spec set_a(rec(),atom()) -> rec().
-spec set_b(rec(),integer()) -> rec().
```

6.5 Dataflow Analysis

The next step is the inference of the `rec_use` functions' success typings. These signatures are not important in the context of this example. The ensuing dataflow analysis is the step that concerns us. Dataflow analysis of the `rec_adt` module does not reveal any errors nor does it generate any refinements for the module's functions. The critical part of the process is the dataflow analysis of the `rec_use` module. Examination of the `ok1/0` and `ok2/0` uncover no type clashes. Three clashes are detected in total in the next two functions, `wrong1/0` and `wrong2/0`.

wrong1/0 While traversing the case clause we find that the result of `rec_adt:new()` is matched against the pattern {`rec, _, 42`}. The success typing we calculated earlier shows that the function returns an opaque type. Comparing the opaque type with the pattern we conclude that this is a breach of *rec()*'s opaqueness. In the next case clause we also detect an inconsistency; `R` is assigned the opaque type *rec()* and then a type test, `is_tuple()`, is used on R. This is also marked as an opaque type violation.

wrong2/0 Here the variable `R` is bound to the *tuple()* by the builtin `list_to_tuple/1` function. Later `R` is used on the `rec_adt:get_a/1` function as an argument. From the type signature analysis we are aware that `get_a/1` expects an opaque argument. Therefore we conclude that this call is abusing *rec()*'s abstract data type interface.

Note that this program does not actually have type clashes that will cause it to fail. This is the case because, in reality, Erlang does not have an implementation for opaque types. All the abnormalities detected are programming errors; opaque type violations that could have negative impact in the future.

No refinements were made during the dataflow pass. Thus, there is no need to repeat the analysis; we have reached a fixpoint. If refinements had been made, the success typing analysis would be repeated to produce "better" typings and the dataflow analysis would search for more errors. This cycle would continue until a fixpoint was achieved. With the

completion of the dataflow analysis the process comes to an end. Dialyzer terminates and prints out the warning it has gathered.

```
rec_use.erl:17:
  The attempt to match a term of type rec_adt:rec() against
  the pattern {'rec', _, 42} breaks the opaqueness of the term
rec_use.erl:18:
  Guard test tuple_size(R::rec_adt:rec())
  breaks the opaqueness of its argument
rec_use.erl:23:
  The call rec_adt:get_a(R::tuple()) does not have an opaque
  term of type rec_adt:rec() in position 1
```

Chapter 7

Results

Dialyzer is a valuable tool that can aid in the development and testing of Erlang applications. Expanding Dialyzer's analysis with the ability to detect opaque type violations is not a novelty upgrade. Opaque type violations are a real problem that can cost time and money. We can define new opaque types at will using Erlang's module system and we are able to reliably detect many types of opaque violations through the marriage of success typings and dataflow analysis.

When we set out to test our work we did not expect to find errors in the thouroughly tested and maintained Erlang library. However, after only defining a few opaque types (*array(), dict(), digraph(), gb_set(), gb_tree(), queue(), set(), tref()*) we located several defects related to opaque type violations. Most violations where tied to the `gb_set` module. Specifically, use of the `is_list/1` guard on *gb_sets* was the most common mistake but also the confusion of the interface between *gb_set()* and *gb_trees()* as well as pattern matching on the *queue()* abstract data type raised several warnings. With the introduction of more opaque types and the adaptation of opaque types by Dialyzer's considerable user base, the benefits will become even more apparent.

The project is not by any reasonable margin flawless. We have no proof that all opaque violations will be detected. As a matter of fact we are aware of cases where obvious defects go unchecked. Also the warnings we produce although they are well founded and correct, do not always reflect critical situations, the way originally Dialyzer did. This at first might seem strange to the user accustomed to previous Dialyzer versions, but we believe that familiarization is a matter of understanding the concept of our work and the importance of opaque types even in an environment such as Erlang, where opaque types are not actually a part of the language.

Opaque types have slowly started to show up in new applications; we are already receiving feedback and bug reports. We hope that opaque type specifications will be used in new applications, since no drawbacks are introduced. Our work compliments Dialyzer operation unintrusivley. Thanks to the extensive Erlang library we were able to test the changes in a very large code base. A large number of bugs was eliminated and the analysis now has a very stable behavior. As a matter of fact, while debugging Dialyzer, we even discovered underlying Dialyzer issues.

Dialyzer is a exciting sophisticated project. Working on the program and making a useful contribution has been most fullfiling. In the future we plan to maintain the code and improve on it. There is a lot of work to be done in ironing out the code and optimizing

certain aspects of the analysis, as well as adding the capacity for detection of structure inspecting functions.

Chapter 8

More Examples

Example with the array() builtin opaque data type

```
-module(array_use).

-export([ok1/0, wrong1/0, wrong2/0]).

ok1() ->
  array:set(17, gazonk, array:new()).

wrong1() ->
  {array, _, _, undefined, _} = array:new(42).

wrong2() ->
  case is_tuple(array:new(42)) of
    true -> structure_is_exposed;
    false -> cannot_possibly_be
  end.
```

Analysis Results

```
array_use.erl:12: The type test is_tuple(array())
        breaks the opaqueness of the term array()
array_use.erl:9: The attempt to match a term of type array()
        against the pattern {'array', _, _, 'undefined', _}
        breaks the opaqueness of the term
```

Example with the dict() builtin opaque data type

```
-module(dict_use).

-export([ok1/0, ok2/0, ok3/0, ok4/0, ok5/0]).
-export([middle/0]).
-export([w1/0, w2/0, w3/0, w4/1, w5/0, w6/0, w7/0, w8/1]).
```

```
-define(DICT, dict).

%%---------------------------------------------------------
%% Cases that are OK
%%---------------------------------------------------------

ok1() ->
  dict:new().

ok2() ->
  case dict:new() of X -> X end.

ok3() ->
  Dict1 = dict:new(),
  Dict2 = dict:new(),
  Dict1 =:= Dict2.

ok4() ->
  dict:fetch(foo, dict:new()).

% ok5 is OK since some_mod:new/0 might
% be returning a dict()
ok5() ->
  dict:fetch(foo, some_mod:new()).

middle() ->
  {w1(), w2()}.

%%---------------------------------------------------------
%% Cases that are problematic w.r.t. opaqueness of types
%%---------------------------------------------------------

w1() ->
  gazonk = dict:new().

w2() ->
  case dict:new() of
    [] -> nil;
    42 -> weird
  end.

w3() ->
  try dict:new() of
    [] -> nil;
    42 -> weird
```

```
  catch
    _:_ -> exception
  end.

w4(Dict) when is_list(Dict) ->
  Dict =:= dict:new();
w4(Dict) when is_atom(Dict) ->
  Dict =/= dict:new().

w5() ->
  case dict:new() of
    D when length(D) =/= 42 -> weird;
    D when is_atom(D) -> weirder;
    D when is_list(D) -> gazonk
  end.

w6() ->
  is_list(dict:new()).

w7() ->
  dict:fetch(foo, [1,2,3]).

w8(Fun) ->
  dict:merge(Fun, 42, [1,2]).
```

Analysis Results

```
dict_use.erl:38:
        The attempt to match a term of type dict() against
        the pattern 'gazonk' breaks the opaqueness of the term
dict_use.erl:42:
        The attempt to match a term of type dict() against
        the pattern [] breaks the opaqueness of the term
dict_use.erl:43:
        The attempt to match a term of type dict() against the
        pattern 42 breaks the opaqueness of the term
dict_use.erl:48:
        The attempt to match a term of type dict() against the
        pattern [] breaks the opaqueness of the term
dict_use.erl:49:
        The attempt to match a term of type dict() against the
        pattern 42 breaks the opaqueness of the term
dict_use.erl:55:
        Attempt to test for equality between a term of type
        maybe_improper_list() and a term of opaque type dict()
```

```
dict_use.erl:57:
        Attempt to test for inequality between a term of
        type atom() and a term of opaque type dict()
dict_use.erl:61:
        Guard test length(D::dict()) breaks
        the opaqueness of its argument
dict_use.erl:62:
        Guard test is_atom(D::dict()) breaks
        the opaqueness of its argument
dict_use.erl:63:
        Guard test is_list(D::dict()) breaks
        the opaqueness of its argument
dict_use.erl:67:
        The type test is_list(dict()) breaks
        the opaqueness of the term dict()
dict_use.erl:70:
        The call dict:fetch('foo',[1 | 2 | 3,...]) does not
        have an opaque term of type dict() in position 2
dict_use.erl:73:
        The call dict:merge(Fun::any(),42,[1 | 2,...]) does
        not have opaque terms in positions 2 and 3
```

Example with the tref() "timer" builtin opaque data type

```
%%--------------------------------------------------------
%% A test case with:
%%   - a genuine matching error
%%   - a violation of the opaqueness of timer:tref()
%%   - a subtle violation of the opaqueness of timer:tref()
%%--------------------------------------------------------

-module(timer_use).
-export([wrong/0]).

-spec wrong() -> error.

wrong() ->
  case timer:kill_after(42, self()) of
    gazonk -> weird;
    {ok, 42} -> weirder;
    {Tag, gazonk} when Tag =/= error -> weirdest;
    {error, _} -> error
  end.
```

Analysis Results

```
timer_use.erl:16:
        The pattern 'gazonk' can never match the type
        {'error',_} | {'ok',tref()}
timer_use.erl:17:
        The attempt to match a term of type {'ok',tref()}
        against the pattern {'ok', 42} breaks the opaqueness
        of tref()
timer_use.erl:18:
        The attempt to match a term of type
        {'error',_} | {'ok',tref()} against the pattern
        {Tag, 'gazonk'} breaks the opaqueness of tref()
```

Bibliography

[1] Abstract data types. http://en.wikipedia.org/wiki/Abstract_data_types.

[2] Erlang documentation. http://www.erlang.org/doc/.

[3] Erlang man pages. http://www.erlang.org/man/.

[4] Opaque pointers. http://en.wikipedia.org/wiki/Opaque_pointer.

[5] *Abstract Data Types: Specifications, Implementations and Applications*. D.C. Heath and Company, 1996.

[6] Robert Harper. Introduction to standard ml.

[7] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium*, volume 3302 of *LNCS*, pages 91–106, Berlin, Germany, 2004. Springer.

[8] Tobias Lindahl and Konstantinos Sagonas. Typer: A type annotator of erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17 – 25, New York, NY, USA, 2005. ACM Press.

[9] 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.

[10] John Peterson Paul Hudak and Joseph Fasel. A gentle introduction to haskell, version 98. http://www.haskell.org/tutorial/index.html, 2000.

[11] Konstantinos Sagonas. Experience from developing the dialyzer: A static analysis tool detecting defects in erlang applications. In *ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[12] 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.