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

Πολυμορφικός συμπερασμός τύπων επιτυχίας στη γλώσσα Erlang

Διπλωματική Εργασία

της

Μαρίας Κοτσιφάκου

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

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Σεπτέμβριος 2012

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

Πολυμορφικός συμπερασμός τύπων επιτυχίας στη γλώσσα Erlang

Διπλωματική Εργασία

της

Μαρίας Κοτσιφάκου

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $17^{η}$ Σεπτεμβρίου, 2012.

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

Αθήνα, Σεπτέμβριος 2012

....
Κοτσιφάκου Μαρία

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

Περίληψη

Ο εντοπισμός λαθών σε προγράμματα κατά τη διαδικασία της ανάπτυξης καθώς και οι έλεγχοι σε ήδη υπάρχοντα κώδικα συνιστούν σημαντικό μέρος του χρόνου που απαιτείται για την ανάπτυξη και τη συντήρηση εφαρμογών. Συνεπώς η ανάπτυξη εργαλείων που βοηθούν τον προγραμματιστή στον εντοπισμό λαθών είναι σημαντική για τον περιορισμό του απαιτούμενου χρόνου και την αύξηση της αποτελεσματικότητας των ελέγχων. Αυτή η εργασία γίνεται στο πλαίσιο του Dialyzer, ενός εργαλείου που χρησιμοποιεί στατική ανάλυση για να προσδιορίσει λάθη σε προγράμματα στη γλώσσα Erlang. Η ανίχνευση λαθών βασίζεται στην εξαγωγή τύπων με χρήση τύπων επιτυχίας (success typings), η οποία όμως δεν υποστηρίζει πολυμορφικούς τύπους στα ορίσματα και στους τύπους επιστροφής των συναρτήσεων. Σε αυτή την εργασία επεκτείνονται οι δυνατότητες του Dialyzer με την εισαγωγή πολυμορφικών τύπων με στόχο την ανίχνευση, με μεγαλύτερη ακρίβεια, λαθών σε προγράμματα όπου χρησιμοποιούνται πολυμορφικές δομές δεδομένων.

Λέξεις Κλειδιά

Στατική ανάλυση, Συμπερασμός τύπων, Τύποι επιτυχίας, Πολυμορφικοί τύποι, Erlang, Dialyzer

Abstract

Error correction in programs during the development phase as well as in existing code tends to consume a significant fraction of programmers' time. Tools that address this problem by automating error detection result in less time consumed during development and testing as well as reduced number of bugs. This thesis is done in the context of the Dialyzer, a static analysis tool that detects programmer errors in Erlang programs such as definite type errors, unreachable code due to unsatisfiable conditions, concurrency errors, etc. To detect type errors, Dialyzer is using type inference of success typings, which albeit is currently restricted to inferring monomorphic types of arguments and return results of functions. This thesis presents the extention of this analysis to add parametricity to these types and thereby be able to possibly catch more errors in programs where polymorphic types such as sets, trees, etc. are used.

Keywords

Static analysis, Type inference, Success Typings, Polymorphic types, Erlang, Dialyzer

Contents

List of Tables

List of Listings

Chapter 1

Introduction

Erlang is a widely used programming language. Its main use is in the telecommunications industry, where fault-tolerance and support for concurrency is essential. It is dynamically typed, which allows fast development time and increased programmer productivity. However, the flexibility of a dynamic type system is often a drawback rather than an advantage when developping large scale applications. In this case, the failure of the type system to impose strict constraints about the allowed use of language constructs often leads to serious semantic errors being undetected at compile-time, which in turn results in time consuming debugging after the error has been identified at runtime.

Dialyzer (DIscrepancy AnaLYZer for ERlang) is a static analysis tool, which was developed in order to address this problem. It uses the type information that is implicit in Erlang programs to infer the widest possible input and return types for which functions may succeed. This approach leads to zero false positives, and only detects definite errors. Apart from this analysis, based on success typings, Dialyzer has been gradually extended in order to detect different classes of errors.

Aiming to improve Dialyzer's accuracy even further, this thesis focuses on including polymorphic types in the analysis. Currently, Dialyzer is restricted to inferring monomorphic types of arguments and return results of functions. Type variables are eliminated during the analysis, thus information about polymorphic types is lost. The remaining information expresses only the widest possible domain and range for which a function call may succeed, and not the relationship between them. This analysis will improve the accuracy of error detection in programs where polymorphic data structures are used by maintaining type variables in the success typings of functions, as well as the corresponding constraints about their types. As a result, more accurate constraints will be available at the call sites of polymorphic functions, which will express the relationship between the types of the arguments and the result during the analysis.

The rest of this diploma thesis is organized as follows. In Chapter 2 we provide a general background for Erlang and Dialyzer, essential for the comprehension of the problem we address. In Chapter 3 we describe the changes that were required for the generation and analysis of polymorphic types. Finally, Chapter 4 contains related work and suggestions for further work related to this thesis.

Chapter 2

Background

2.1 Erlang

Erlang is a general-purpose declarative programming language with automatic memory management and support for distribition, fault-tolerance, soft-realtime execution and on-the-fly code reloading. It also supports concurrency, with a small but powerful set of primitives which allow the creation of processes and the communication between them. Its sequential subset is a functional language, with single assignment, strict evaluation and dynamic typing [2, 6].

Erlang was originally designed in order to improve the development of telephony applications. Its main implementation, the Erlang/OTP(Open Telecom Platform), developed by Ericsson, has been open source since 1988 and has been used by several companies worldwide, such as Ericsson, Nortel, T-mobile, for the development of large-scale applications. Nowadays, its main application remains in large-scale embedded control systems developed by the telecommucications industry. However, its popularity is growing due to the demand for concurrent fault-tolerant services.

2.2 Dialyzer

Dialyzer (DIscrepancy AnaLYZer for ERlang) is a static analysis tool used to detect discrepancies in single erlang modules or applications [5]. These discrepancies may be definite type errors, redundancies such as dead or unreachable code due to programmer error or unsatisfiable conditions, and recently concurrency errors (race conditions) ([8, 9, 4]).

The analysis performed in Dialyzer is sound for error, since it aims to identify the widest possible set of terms for which it can be proved that type clashes will occur. To this end, Dialyzer's inference algorithm is based on success typings, which is the set of terms for which the abovementioned cannot be proved. For any function, its success typing is an over-approximation of the set of terms for which the application will succeed. The domain of a function's success typing contains all the possible terms which the function can accept as parameters, and its range contains all the possible return values corresponding to this domain. The aim of the inference algorithm is to reduce the domain and the range of the success typing as much as possible without excluding any valid terms. This approach means that the success typing of a function contains all the correct uses of a function, as well as some mistaken. If a function is used in a way that is compatible with its success typing, then the call may or may not succeed, but if it is used in an incompatible way then it will definitely fail.

2.2.1 Analysis

In order to find the success typings, Dialyzer traverses the code of all functions included in the analysis in order to generate constraints. Dialyzer then iterates between constraint solving (bottom-up) and dataflow analysis (top-down) until it reaches a fix point or until the analysis fails. The fix point, if reached, constitutes the success typings.

2.2.2 Refinement

The module system of Erlang allows for specialising the success typings of module-local functions based on their actual instead of their possible uses. For the refinement procces, Dialyzer uses a dataflow analysis which propagates information forward in the control flow. This analysis starts at the entry point of all functions that are not module-local, assigning to their arguments the domains of their success typings, and takes into account the fact that all the functions' intended calls are known in order to tighten the existing success typings. The refined succes typings allows Dialyzer to locate more type clashes, case clauses that can never match, guards that will always fail, and other errors.

In Listing 2.1, the type of function `id/1` will be calculated as *any() -> any()*. However, since the function is not exported, its type will be further restricted based on its actual uses, and its final type will be *42 -> 42*.

```
-module(id).
-export([foo/0]).

id(X) -> X.

foo() -> id(42).
```

Listing 2.1: Refinement example

2.2.3 Type specifications

Dialyzer offers the programmer the ability to restrict the type of a function by providing a contract that captures its intended uses. Dialyzer uses this information to further restrict the success typing generated for the function. However, in order for Dialyzer to take the contract into account, it must be more specialized that the available success type for the above mentioned function.

The contracts can be overloaded, but if they overlap, or if they are not specific enough for the analysis to choose which clause to consider, they are collapsed by taking the union of their clauses. Listing 2.2 contains a simple and an overloaded, thus more restrictive, spec for a function which perfors a numerical operation to one argument.

```
-spec numeric_function(number()) -> number().

-spec numeric_function(integer()) -> integer();
                      (float())   -> float().
```

Listing 2.2: Contract example

2.3 Type System

2.3.1 Sets of terms

A piece of data of any data type is an Erlang term. Erlang provides more specific data types for all the basic term sets. These types constitute a lattice, with the type *any()* being the top type and the type *none()* being the bottom type. Table 2.1 describes these sets.

Set of terms	Description	Examples
Integer	A mathematical integer	-17,0,1,42
Float	A floating point number	-3.14,2.718
Atom	A constant with name	hello,'hi'
Bitstring	An untyped series of bits	«3:1,7:6»
Binary	An untyped series of bytes	«1, 17, 42», «"abc"»
Reference	A term unique in Erlang runtime system	—
Function	Functional object	fun(X) -> X + 1 end. fun lists:map/2
Port Identifier	A handle for referring to external programs	—
Proccess Identifier	A handle for referring to Erlang processes	—

Tuple	A compound term with a fixed number of elements of any type	{adam, 2}
		{tree, left_leaf, right_leaf}

List	A compound term with a variable number of elements of any, not necessarily the same, type	[1, 2, 3] [42] [1, foo, [3, 4, bar]]

Table 2.1: Common sets of terms

2.3.2 Built-in Erlang Types

The Erlang types that describe the previous sets of terms are included in Table 2.2.

Set of terms	Related Erlang Type	Description
Integers	integer()	all integers
	neg_integer()	negative integers
	non_neg_integer()	non-negative integers
	pos_integer()	positive integers
	<Int>	a specific integer (singleton type)
	<Lo>..<Hi>	integers between Lo and Hi
Floats	float()	all floats
Atoms	<Atom>	a specific atom (singleton type)
	atom()	all atoms
Bitstrings	bitstring()	all bitstrings
	«»	the empty bitstring (singleton type)
	«_:_*<U> »	bitstrings of length elements_number * U (in bits)
	«_:<B>,_:_*<U> »	bitstrings of length B*U (in bits)
Binaries	binary()	all binaries
	«»	the empty binary (singleton type)
	«_:<B> »	binaries of length B (in bytes)
References	reference()	all references
Functions	fun()	all functions
	fun((...) -> Type)	functions of any arity returning Type
	fun(() -> Type)	functions of zero arity returning Type
	fun((T1,...,Tn)-> R)	functions of arity N, accepting arguments of types T1, ..., Tn respectively and returning R
Ports	port()	all ports
Pids	pid()	all pids
Tuples	tuple()	all tuples
	{}	the zero-size tuple (singleton type)
	{T1,...,Tn}	N-arity tuple with elements of types T1, ..., Tn respectively
Lists	[]	the empty list (singleton type)
	list(T),[T]	lists with elements of type T (proper)
	[T,...]	lists with elements of type T (proper)
	T1|...|Tn	the union of terms represented by types T1, ..., Tn
	any()	all Erlang terms
	none()	no terms

Table 2.2: Built-in erlang types

2.4 Analysis

For efficiency reasons, the analysis begins by creating the global function callgraph, which is a directed graph with functions as nodes. The callgraph represents the dependencies between functions, using the notation that if function f calls function h then the graph contains an edge *(f,h)*. Calls between mutually dependent functions form cycles in the call graph, which constitute strongly connected components. The callgraph is then condensed to its SCCs, ending up to be a directed acyclic graph. The resulting DAG is sorted topologically in order to determine functions with no dependencies (which belong to the SSCs that are the leafs of the DAG). These functions are the starting point of the inference algorithm, which continues bottom-up. During the analysis phase, each variable detected in the code is assigned a type variable, each function is assigned two type variables (one general and one for self-calls and calls within the function's SCC), and this mapping is stored. After that, the code is traversed in order to generate constraints. Dialyzer iterates between constraint solving (bottom-up) and dataflow analysis (top-down) until it reaches a fix point or until the analysis fails. The fix point of the analysis is the success typings for the functions of the SCC.

2.4.1 Constraint Generation

At the beginning of the analysis phase, Dialyzer traverses the code and constraints are generated. The constraints are of the following types:

```
-type constr() :: constraint() | constraint_list() | constraint_ref().
```

Listing 2.3: Constraint types

Simple Constraint: This form of constraint expresses the relationship or equality of subtyping between the left hand side and the right hand side of the constraint. An example of its generation during the initial phase of the analysis is a function call, to state that the types of the actual arguments and the result of the call must be subtypes of the types of the formal arguments and the result of the function according to its success typing.

Conjuctive List: This form of constraints expresses the fact that all the constraints which are contained must be satisfied simultaneously. Typically, conjuctive lists of constraints are generated from traversing simple functions with no branches, and contain simple constraints.

Disjunctive List: This form of constraints is used to represent constraints whose disjunction must hold. This is the case when any kind of branches is present at the code, for example functions with multiple clauses, `if` statements and `case` statements. For this case, the constraints for each clause/branch form a conjuctive list, and and these lists are joined in a disjunctive list.

Constraint References: This type of constraint is generated by anonymous or higher order functions.

The module demo in Listing 2.4 will be used to demonstrate the main types of constraints that are generated.

```
-module(demo).
-export([bar/1]).

foo(1) -> 3;
foo(2) -> 4.

bar(X) -> Y = foo(X), Y.
```

Listing 2.4: Constraint example

The constraints for function `foo/1` (Listing 2.5) are:

```
Conjunctive List 3:
  % Assosiates the type of the function with its arguments and return type:
  fun((var(4)) -> var(11)) eq var(12)
  % Every function with one argument must be subtype of the type any() -> any():
  var(12) sub fun((_) -> any())
  % Generated because foo/1 has two clauses:
  Disjunctive list 2:
    % Constraints for the first clause:
    Conjunctive List 0: 1 eq var(4), 3 eq var(11)
    % Constraints for the second clause:
    Conjunctive List 1: 2 eq var(4), 4 eq var(11)
    ...
```

Listing 2.5: Constraints for function `foo/1`

The success typing for function `foo/1` at the end of the analysis (Listing 2.6) is:

```
Succ typings:
  {demo,foo,1} :: fun((1 | 2) -> 3 | 4)
```

Listing 2.6: Success type of `foo/1`

Dialyzer uses this result to analyze function `bar/1`. The constraint list can be seen in Listing 2.7.

In this example, we notice some general constraints:

Function constraint: This constraint has the general form that is shown at the first element at lists 0 and 3. Its purpose is to assosiate the type of the function with its arguments and the result. Function foo is assosiated with the type variable *var(12)*, which is the type variable of a function with arity 1, one argument assosiated with the type variable *var(4)* and return type assosiated with the type variable *var(11)*. Solving this constraint actually binds the function to its type.

Contraints from clauses: List 2 is a disjunctive list of constraints corresponding to the two clauses. Generally, there will be as many elements in the disjunction as many

```
Conjunctive List 0:
  % Assosiates the type of the function with its arguments and return type:
  fun((var(13)) -> var(23)) eq var(24)
  % Every function with one argument must be subtype of the type any() -> any():
  var(24) sub fun((_) -> any())
  % Result of function call:
  fun((var(45)) -> var(44)) eq var(43)
  % Result of function call:
  var(43) sub fun((1 | 2) -> 3 | 4))
  % Result of function call:
  var(15) sub 3 | 4
  % Result of function call:
  var(15) sub var(44)
  % Result of function call:
  var(14) sub 1 | 2
  % Result of function call:
  var(14) sub var(45)
  ...
```

Listing 2.7: Constraint example for function `bar/1`

clauses the function has. Similarly, disjunctive lists are the result of other branches, like `case` and `if` statements.

Function Calls: Function calls generate a conjunctive list which requires the types of all actual arguments and the actual return type of the function call to be subtypes of the respective types in the success type of the function.

2.4.2 Constraint Solving

Constraint solving is an iterative process, and continues until it reaches a fixpoint. For simple constraints, solving simply requires restricting the type of the type variables involved according to the operation that is described by the costraint (equality or subtyping) and updating the mapping with the current result. Solving constraint lists or refs require storing the previous mappings and compare to with the result of another iteration in order to determine whether changes occur.

2.4.3 Storing

When a fixpoint is reached, the success types are stored in a Persistent Lookup Table (PLT). The PLT can be used for future reference, as a base of trusted and type-checked code in order to analyze other erlang functions and modules which contain calls to analyzed code.

Chapter 3

Polymorphic Types

Currently, Dialyzer is restricted to inferring monomorphic types of arguments and return results of functions. At the beginning of the analysis, the most general type for every type variable is assumed, that is *any()* for every variable and *fun(any(), ..., any()) -> any()* (*any()* is repeated *N* times as an argument type) for every function with arity *N*. While solving a simple constraint, Dialyzer maps the left hand side of a subtyping constraint or both sides of an equality constraint with the widest possible type that is contained in both types (`erl_types:t_inf/2`), as they derive from the current mapping, and updates the mapping. As a result, the types are gradually restricted, but information about equality between type variables is lost. It is this problem that is addressed here, in that managing to collect and use information about the equality between type variables that is included in the constraints will result in detecting polymorphic types, which in turn will increase the accuracy of error detection due to more strict constraints derived from the call sites of polymorphic functions.

3.1 Storing

In order to produce and use information about polymorphic types in the analysis, keeping information about type variables is essential. The current representation of function types in the PLT is in the format *{Result_ Type, [Argument_ Type]}*, where the *Result_ Type* and the list of elements of type *Argument_ Type* are restricted to constant erlang types. This needed to be extended, in order to include both the constant type of the function, for consistency of the current analysis, and the polymorphic type as well as the constraints that apply to the type variables that it contains. As a result, an entry to the PLT may be *{Result_ Type, [Argument_ Type]}* or *{Result_ Type, [Argument_ Type], {Polymorphic_ Function_ Type, [Constraints]}}*. For simplicity, since these constraints always express subtyping, each of them is represented as a tuple *{Type Variable, Corresponding Type}*.

3.2 State Definition

During the analysis of an SCC, information required for the analysis are stored in a record of type *state*. This record will be referred as *State*. The fields that are of interest are mentioned:

cmap: A dictionary which after traversing the code of each function in the SCC contains the constraint list for every function in the SCC.

next_label: A field of type *label*, whose value is higher that any label used in the code.

In order to generate and use information abut polymorphic types, the record *State* is used to store some additional information.

p_labels: For the purposes of the analysis, a field of type *[label]* is kept, and is initialized as an empty list when the record *State* is created. This field will be used to determine which type variables were inserted due to polymorphic function constraints, thus will be treated differently in constraint solving.

fun_vars: A dictionary which maps each polymorphic function to its polymorphic type, if one has been generated.

3.3 Constraint Generation

In order for the analysis to use information about polymorphic types, two elements are required.

Firstly, unique type variables must be assigned to the variables that are used to capture information about each call. This is essential, because although a polymorphic type variable is bound to a type during a call, this type is generally subject to change between different calls. For this reason, the type variables related to the constraints generated from one call are given a unique identifier, rather than using a symbolic name, permanently stored in the PLT. This is accomplished by the function `create_pvars` (Listing 3.1), which returns a mapping between the plt type variables and the unique ones that will be used for analysis. This function also returns the new record *State* from which the constraint generation will continue, to ensure the uniqueness of type variables and to provide information about the created polymorphic type variables.

```
create_pvars([],Mapping,State) -> {dict:from_list(Mapping),State};
create_pvars([Hvar|Tvars],Mapping,State) ->
  {NewState, NewVar} = state__mk_var(State),
  OldId = erl_types:t_var_name(Hvar),
  NewId = erl_types:t_var_name(NewVar),
  create_pvars(Tvars,[{OldId,NewId}|Mapping],
               NewState#state{p_labels = [NewId|p_labels]}).
```

Listing 3.1: Generation of unique type variables

Secondly, appropriate constraints must be generated when acquiring information about the function type from the PLT.

These goals are accomplished by the following procedure (Listing 3.2), which is now included in function `dialyzer_typesig:get_plt_constr/4`. While describing this procedure, we refer to the function's polymorphic type in the PLT as `PltPolType` and to the list of constraints that contains the information about its type variables as `ConList`.

- collect the type variables that are used in the function success typing
- for each type variable that is required for the representation of the polymorphic type, generate a new type variable. The result of the function is a mapping between the plt type variables and the unique ones that will be used for analysis. This function also returns the new record *State* from which the generation will continue, to ensure the uniqueness of type variables
- use the polymorphic type instead of the constant type in order to determine the *Result_ Type* and *[Argument_ Type]*
- isolate the types of the polymorphic type variables
- express the constraint by using the unique type variable for each constraint
- create a conjunctive list of constraints about the function's actual arguments, result and type variables.

```
VarList = erl_types:t_collect_vars(PltPolType)
{NewVarDict,NewState} = create_pvars(VarList,[],State)
ActualParType = erl_types:t_replace_vars(PltPolType, NewVarDict)
ActualRetType = erl_types:t_range(ActualParType)
ActualArgTypes = erl_types:t_fun_args(ActualParType)
{PltVars,Types} = lists:unzip(ConList)
ActualVars = lists:map(fun (X) -> erl_types:t_replace_vars(X, NewVarDict) end,
                       PltVars)
state__store_conj_lists(lists:append([Dst|ArgVars],Types), sub,
                       lists:append([ActualRetType|ActualArgTypes],ActualVars),
                       NewState)
```

Listing 3.2: Constraint generation for call to a polymorphic function

3.4 Solving

3.4.1 Find polymorphic types

In order to detect polymorphic types, we need to find which types are related with equality constraints within a conjunctive list of constraints. It is required that the equality constraints are not part of disjunctive lists of constraints. If this is the case, then there might be a clause where two types are equal, but there is no guarantee that equality holds in general, thus it is not safe to include information from these constraints.

To accomplish that, we use a union-find approach for each function [1], whose steps are dercribed briefly.

- The constraints for each function in the current analysis phase are gathered from the current record *State* (field `cmap` of record *State* containts the mapping between a function's identifier and the respective list of constraints).

- The type variables that are bound with the function's domain and range are inserted in a union-find data structure. The constraint list is traversed, in order to gather information about the types that are detected in equality constraints. These types are also inserted in the union-find data structure as singleton sets.
- The constraint list is traversed again, and for every equality constraint that we detect the types of the left hand side and the right hand side of the constraint are unified. The new representative of the set is determined by the following rule:
  - Between two type variables, the type variable with the smaller identifier is chosen.
  - Between a type variable and any other type, the latter is chosen.
- The domain and the range of the function are the types of the representatives of their sets. The type variables that are contained in the domain and the range are extracted, and if every type variable in the function's range also occurs in its domain, the function is polymorphic and its type is calculated using the interface provided in `erl_types`, and specifically the function `t_fun(Domain,Range)`.

3.4.2 Storing polymorphic types

After completing the analysis of a function or an SCC, a mapping between the identifiers and the types of each analyzed function is returned. This information is later stored in the PLT for future reference. Currently, a function's type is looked up in a mapping with its unique function key as identifier. In order to provide information about polymorphic types as well, the previously created field `fun_vars` is required. The function's type is looked up in both the original mapping and `fun_vars`. If `fun_vars` contains a polymorphic type, then the type variables that are assosiated with it are looked up in the same mapping in order for their types to be found in the context of the function/SCC. Then, the appropriate list of subtyping constraints (in the form discussed earlier) are produced, and the function returns a tuple with two elements: the function's constant type, and the function's polymorphic type along with the subtyping constraints for its arguments and range. Otherwise, the function's constant type is returned.

3.4.3 Analysis of polymorphic types

The heart of the analysis algorithm is the function `dialyzer_typesig:solve_one_c/3`, which solves a simple constraint (first argument) in the context of a given mapping (second argument), and returns a new mapping with the types assosiated with the type variables of the constraint. As mentioned before, a function call generates subtyping constraints between its actual arguments and result and its corresponding success type. The problem addressed here is that in the case of polymorphic functions, we can further restrict the polymorphic type of the function, for a specific call. Since the function is polymorphic, and unique type variables are used to express the constraints for this call, then we can safely

- narrow the type of the function's formal argument for the specific call to be equal to the types of its actual parameters. If this type is more general that the one specified for the function by its success type, then the type of the actual parameter will be restricted instead, as usual. For example, a constraint generated for an actual argument with type *var(x)* and corresponding formal argument *var(y)*, restricted to *integer()*, would be `{lhs = var(x), op = sub, rhs = var(y)}`, `{lhs = var(y), op = sub, rhs = integer()}`. Assume that *var(x)* is bound to type *0*. Currently, the solution of this set of constraints will match the type variable *var(y)* to the type *integer()*, when clearly it can be further restricted to the type *0* for this call.

- in the same way, narrow the type of the function's result for the specific call to be equal to the type of the actual result.

In other words, when constraints related to calls of polymorphic functions occur, the subtyping constraints can be treated as equality constraints.

This is accomplished by introducing the reverse constraint when solving constraints whose right hand side is a type variable of a polymorphic type. The variables that were introduced due to polymorphic function types can be detected by using the field `p_labels` that is kept in the record *State* for the current SSC. Since it was updated with the corresponding identifier for every type variable that was introduced during constraint generation for calls of polymorphic functions, it contains an exhaustive list of all these variables during the solving phase. As a result, a new function `solve_one_c/4` is defined, whose fourth argument is the record *State*. After invoking `solve_one_c/3`, it checks whether the constraint has a type variable which was introduced during a call to a polymorphic function as its right hand side. If this is the case, then this constraint expresses the abovementioned relationship between a polymorphic function's actual and formal arguments, and as a result `solve_one_c/3` is invoked again for the reverse costraint.

3.5 Evaluation of proposed analysis

The proposed analysis is demonstrated using a small erlang module, shown in Listing 3.3.

```
-module(foo_bar).
-export([foo/2,bar/0]).

foo(X,Y) -> {Y,X}.

bar() -> {A,B} = foo(1,'hi'), A+B.
```

Listing 3.3: Definite type error

As mentioned earlier, the analysis currently performed by Dialyzer does not detect the type error, thus runtime failure will occur. This type error cannot be detected because the success typing of function `foo/2` is *fun((any(),any()) -> {any(),any()}*, which fully captures the constraints about the expected inputs and the output but does not express the relationship between the input arguments and the result. The constraints generated for function `bar/0` (Listing 3.4) demonstrate the fact that the actual arguments of the function call are restricted to be subtypes of the function's formal arguments (type variables *var(55)* and *var(56)*) and also the result at the call site of the function (type variable *var(19)*) is restricted to be subtype of the function's result (type variable *var(54)*).

```
Conjunctive list 0
% Generic function constraint
fun(() -> var(33)) eq var(34)
% A function with arity 0 is subtype of fun(() -> any())
var(34) sub fun(() -> any())
% Type of foo/2
fun((var(55),var(56)) -> var(54)) eq var(53)
% Type of first actual argument of foo is subtype of first formal argument
1 sub var(55)
% Type of second argument of foo is subtype of second formal argument
'hi' sub var(56)
% The return result of the function call is subtype of the function's result type
var(19) sub var(54)

{var(57),var(58)} eq var(19)
var(19) sub Fun@L781(var(19))
% A tuple with two elements is subtype of {any(),any()}
var(19) sub {_,_}
...
```

Listing 3.4: Constraints for function `bar/0`

Our analysis, using the union-find algorithm that was described earlier (3.4.1) will generate the polymorphic success typing *fun((var(1),var(2))->{var(2),var(1)})* for function `foo/2`. This type will be stored in the PLT, as well as the corresponding PLT constraints *[{var(1),any()},{var(2),any()}]*. Given the polymorphic type of function `foo/2`, the constraint generation process that is presented in the above section (3.3) will produce the following constraints (Listing 3.5):

```
Conjunctive list 0
% Generic function constraint
fun(() -> var(33)) eq var(34)
% A function with arity 0 is subtype of fun(() -> any())
var(34) sub fun(() -> any())
% Type of foo/2
fun((var(55),var(56)) -> var(54)) eq var(53)
% Type of first actual argument of foo is subtype of first formal argument
1 sub var(55), var(55) sub any()
% Type of second argument of foo is subtype of second formal argument
'hi' sub var(56), var(56) sun any()
% The return result of the function call is subtype of the function's result type
var(19) sub var(54), var(54) sub {var(56),var(55)}
...
```

Listing 3.5: Constraints for function `bar/0`

During the constraint generation, the type variables *var(54), var(55)* and *var(56)* will be added in the field `p_labels` of the record *State*, in order to indicate the fact that they are generated to describe the application of a polymorphic function.

During the analysis of function `bar/0`, described in section 3.4.3, the constraints `{lhs = 1, op = sub, rhs = var(55)}` and `{lhs = 'hi', op = sub, rhs = var(56)}` will be treated as equality constraints, and the types of the type variables *var(55)* and *var(56)* will be bound to the types *1* and *'hi'* respectively. As a result, the result of the function call at this site will be restricted to *{'hi',1}*. The analysis will then be able to detect the type error that occurs due to the use of operator `'+'` with non-arithmetic operands.

This example, however small, demonstrates the significant type information that can be captured using polymorphic types. Applying this analysis in larger modules, where polymorphic data structures are used, will have similar effect in the accuracy of error detection.

Chapter 4

Related and Future Work

4.1 Related Work

As far as Erlang is concerned, Tobias Lindahl and Konstantinos Sagonas have proposed and developed a static analysis tool, Dialyzer, which is based on the concept of success typings, and is used to extract type information that is implicit in Erlang programs and to detect definite errors [8], as opposed to the usual approach of static type systems, which are sound for correctness. Additionally, a language for specifying type contracts at the level of individual functions has been developed [7] by Miguel Jimenez, Tobias Lindahl and Konstantinos Sagonas. This language allows the user to provide information about the intended uses of a function despite its more general type, and as a result allow further specialization of the inferred type. Dialyzer has recently been extended to emmit warnings about different classes of errors, like data races ([4]). Also, the type system has been extended to include intersection types, which improved the accuracy of the analysis and the ability to detect actual errors ([3]).

4.2 Future Work

4.2.1 Sophisticated analysis for polymorphic types

Currently, the analysis is restricted to generate polymorphic types for simple functions. Future work can focus on developing a more sophisticated approach, that can generate polymorphic types for functions with many clauses or self-recursive functions.

4.2.2 Polymorphic contracts

The constraint generation and analysis for polymorphic functions can easily be extended in order to include user-provided information. As a result, polymorphic type specifications may also be used in order to further improve the accuracy of the analysis.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] S. Aronis. Techniques to improve the efficiency of error detection in programs using static analysis. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, January 2011. `http://http://artemis-new.cslab.ece.ntua.gr:8080/jspui/handle/123456789/5443`.

[4] M. Christakis and K. Sagonas. Static detection of race conditions in erlang. In *Proceedings of the 12th international conference on Practical Aspects of Declarative Languages*, PADL'10, pages 119–133, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Dialyzer Documentation. `http://www.erlang.org/doc/apps/dialyzer/index.html`.

[6] Erlang Official Site. `http://www.erlang.org/`.

[7] M. Jimenez, T. Lindahl, and K. Sagonas. A language for specifying type contracts in erlang and its interaction with success typings. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, ERLANG '07, pages 11–17, New York, NY, USA, 2007. ACM.

[8] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM.

[9] K. Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In *Proceedings of the 10th international conference on Functional and Logic Programming*, FLOPS'10, pages 13–18, Berlin, Heidelberg, 2010. Springer-Verlag.