Precise Explanation of Success Typing Errors *

Konstantinos Sagonas
Department of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Josep Silva Salvador Tamarit
Department of Information Systems and Computation
Universitat Politècnica de València, Spain
{jsilva,stamarit}@dsic.upv.es

Abstract

Nowadays, many dynamic languages come with (some sort of) type inference in order to detect type errors statically. Often, in order not to unnecessarily reject programs which are allowed under a dynamic type discipline, their type inference algorithms are based on non-standard (i.e., not unification based) type inference algorithms. Instead, they employ aggressive forwards and backwards propagation of subtype constraints. Although such analyses are effective in locating actual programming errors, the errors they report are often extremely difficult for programmers to follow and convince themselves of their validity. We have observed this phenomenon in the context of Erlang: for a number of years now its implementation comes with a static analysis tool called Dialyzer which, among other software discrepancies, detects definite type errors (i.e., code points that will result in a runtime error if executed) by inferring success typings. In this work, we extend the analysis that infers success typings, with infrastructure that maintains additional information that can be used to provide precise (i.e., minimal) explanations about the cause of a discrepancy reported by Dialyzer using program slicing. We have implemented the techniques we describe in a publicly available development branch of Dialyzer.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Algorithms, Languages, Theory

Keywords Type inference, program slicing, Erlang

1. Introduction

Dynamically typed languages provide flexibility to programmers since they allow program variables to refer to values of different types at different points during program execution. Moreover, the compilers of these languages do not require programmers to write explicit type annotations on expressions, or to provide type signatures for functions. This ability to develop programs rapidly is currently exploited by many modern dynamic languages used in industry but it comes with a cost: many errors that would be caught by a static type system are detected only at runtime, potentially making programs written in dynamic languages less reliable. To ameliorate the situation, many dynamic languages nowadays come with various sorts of type inferencing approaches and tools, aiming to detect most type errors statically rather than dynamically [2, 5, 7]. Such tools allow these languages to combine the expressivity and flexibility of dynamic typing with the robustness of static typing.

Given a program P, we consider the type of an expression as the (possibly infinite) set of values to which this expression can be evaluated using P. Given two expressions e_1, e_2, such that e_1 is a subexpression of e_2, we say that e_1 produces a type error if the type of e_1 is τ_1, the type expected from e_1 by e_2 is τ_2, and $\tau_1 \cap \tau_2 = \varnothing$. Therefore, in this work the term *type error* refers to a specific point in the source code of the program.

Example 1. Assume that the following three Erlang functions appear in a file `ex1.erl` (for the time being ignore the boxes).

```
1  main(A) ->                    8  f(X) -> X+1.
2    X = f(A),
3    case X of                   10 g(42) -> ok.
4      0 -> g(X);
5      _ -> g(X-1)
6    end.
```

If we analyze this program with the type analysis of Dialyzer[1] we get the following warning pointing out a specific type error:

> ex1.erl:4: The call g(X::0) will never return since it differs in the 1st argument from the success typing arguments: (42).

Because Dialyzer performs an analysis that is sound for defect detection (i.e., it produces no false positives), rather than being sound for correctness, we can be sure that the call `g(X)` at line 4 will never succeed for `X` = `0`. Although the type analysis has very fine-grained type information (note that it employs *singleton types* instead of collapsing `0` and `42` to *integer*), Dialyzer does not provide any explanation about *why* this call will not succeed. Of course, in such small programs it is easy to see why, but in big programs this information is often insufficient and the programmer must track both forwards and backwards the values that manifest an error.

By using *program slicing* [14, 16], and taking `g(X)` in line 4 as the slicing criterion, we would get the portion of the program that has been boxed in the source code. Note that `X` at line 4 control-depends on the pattern and the variable of the case expression. Also, it data-depends on the definition of `X` in line 2. This definition, `X = f(A)`, in turn depends on `A` and the function call. Thus, the

* Work partially supported by the EU IST-2011-287510 project RELEASE and by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019. Most of this work was done during a three month visit of the second and third authors to Uppsala.

PEPM'13, January 21–22, 2013, Rome, Italy.

[1] Dialyzer [9] is arguably the most advanced static analysis tool developed for Erlang; it is part of Erlang/OTP and is heavily used by Erlang projects.

slice also includes the function definition because it is needed to determine the value of X in line 2. Even though this program slice is complete (it includes the cause of the type error), it contains irrelevant information as it can be seen in the slice shown below:

```
1  main(A) ->                 8  f(X) -> X+1.
2    X = f(A),
3    case X of               10  g(42) -> ok.
4      0 -> g(X);
5      _ -> g(X-1)
6    end.
```

where the cause of the type error has been boxed in the source code. Basically, variable X will always have the value 0 due to the pattern but the only value in the domain of the success typing of g is 42.

A reader not familiar with program slicing might think that an easy way of providing a minimal slice would be to restrict the dependencies by only considering some subset of control or data dependencies instead of all of them. However, explaining a type error for some program could require to ignore some dependencies only in some cases but consider them in others. Let us see this with another example.

Example 2. Consider the following Erlang program (once again, initially ignore the boxes):

```
1  main(A) ->                 10  f(0) -> 0;
2    X = f(A),                11  f(1) -> 1.
3    Y = g(A,0),
4    Z = f(Y),                13  g(1,0) -> 0.
5    case Z of
6      0 -> g(X,A);
7      _ -> g(X-1,0)
8    end.
```

On this program, Dialyzer will report the following warning:

> ex2.erl:6: The call g(X::0|1,A::1) will never return since it differs in the 2nd argument from the success typing arguments: (1,0).

Dialyzer's analysis has inferred that the call to function g in line 6 will be either g(0,1) or g(1,1). But the only success typing arguments for this function are $(1, 0)$, thus the call will always fail. Here again, this information is useful but insufficient, because we do not know what part of the source code produces this problem. With program slicing and taking A in line 6 as the slicing criterion, we get the portion of the program that has been boxed in the code. But the cause of the type error could be explained as shown below:

```
1  main(A) ->                 10  f(0) -> 0;
2    X = f(A),                11  f(1) -> 1.
3    Y = g(A,0),
4    Z = f(Y),                13  g(1,0) -> 0.
5    case X of
6      0 -> g(X,A);
7      _ -> g(X-1,0)
8    end.
```

Observe how the type error gets produced: The argument of main (variable A) could initially have any value. However, after the call to f, its only possible values are 0 or 1 (otherwise, f(A) would not have succeeded). Similarly, after the call to g(A,0), the only possible value of A is 1. Note that this call is the cause of the type error, and this is a use of A, not a definition. Therefore, even uses of variables can restrict their values resulting in success typing errors.

In this work we describe a technique to automatically compute program slices that minimally explain success typing errors. Contrary to previous approaches [2, 7], we advocate for a parameterized technique where one of its inputs is the particular type error we want to explain. This means that the explanation of each different Dialyzer warning is computed differently. Roughly, our technique: (i) Analyzes the source code to automatically compute a collection of type constraints that must be satisfied. This is done by an instrumentation of a constraint generation system based on *subtype constraints* [10] that is able to propagate source position information during the analysis and attach these positions to the constraints. (ii) Uses a constraint solver to automatically find inconsistencies in the subtype constraints. When an inconsistency is found, all source positions needed to produce the inconsistency are collected by the constraint solver. (iii) Post-processes each particular inconsistency with a final analysis that automatically determines the exact source positions that are needed to explain the particular type error. All three phases are completely automatic. Note that the parameterization is done in the third phase that performs a post-processing for each type error detected during the second phase.

We present our formalization and our implementation for the functional language Erlang. The implementation has been performed in an experimental version of Dialyzer explaining errors using program slicing and its implementation is publicly available.

In summary, the contributions of this paper are the following: (i) Instrumentation of a constraint generation system to collect the source positions associated with subtype constraints. (ii) Definition of a constraint solver that is able to collect source positions associated with success typing errors. (iii) Definition of a parameterizable slicer for Core Erlang based on source positions associated with these errors. (iv) A prototype implementation in an experimental branch of Dialyzer, a widely used defect detection tool for Erlang.

The rest of the paper is organized as follows. In Section 2 we discuss previous approaches to explaining type errors and how we differ from them. In Section 3 we recall the syntax of Core Erlang and introduce some notation used in the rest of the paper. Section 4 presents the slicing technique we have developed to explain success typing errors; it is divided into three subsections that present a calculus to generate type constraints, a constraint solving system, and a slicing process. We state properties of our technique in Section 5, briefly describe the implementation of the Dialyzer extension in Section 6, and conclude in Section 7.

2. Related Work

By now there exist many techniques and implementations devoted to improve the quality of messages provided by debuggers and compilers and explaining type errors. A good representative in this line is Helium [8], a compiler designed especially for learning Haskell. Helium provides error messages that include hints suggesting improvements that are likely to correct the program. It uses heuristics to identify the most likely part of the program that produced the error—often a different place from where it was detected—and it shows specific messages for different errors.

Another line of research to explain type errors concentrates on showing to the user the part of the source code that is responsible for the type error using program slicing. Program slicing [14, 16] is a general technique of program analysis and transformation whose main aim is to extract the part of a program (the so-called *slice*) that influences or is influenced by a given point of interest. Thus, program slicing can be used to obtain the parts of the source code that influence a type error. The first to use program slicing to explain type errors were Tip and Dinesh [17] who adapted program slicing to the context of term rewriting systems. Unfortunately, as shown in Examples 1 and 2, standard program slicing produces slices that are too big, because they are constructed by traversing all data and control dependencies, and many of their components are unnecessary to explain a type error.

A different approach for the explanation of types has been developed by Choppella and Haynes [2] based on the use of *type constraint graphs*, which are graphs able to represent all type constraints in a program. Their approach constructs a graph where each

node represents an expression of the program and edges represent (equality) type constraints between expressions. A type error is found when a path in the graph connects two different types, meaning that one expression has been used with different types. Once a type error has been found, it is explained by collecting all expressions in the path between the two conflicting types. Of course, these expressions can be mapped to the source code producing a slice. Even though, this approach is conceptually similar to our work, the processes of finding minimal slices are very different. The minimal slices computed using type constraint graphs correspond to a minimal path in the graph. Computing all minimal slices has an exponential cost in that setting, and thus the same work [2] defines an algorithm to only concentrate in a proper subset. Moreover, in contrast to our approach, all type errors are treated in the same way (processing a path in the graph) with a consequent loss of precision.

Haack and Wells were the first to use the term *type error slicing* in their technique to explain type errors in a reduced subset of ML programs [7]. That work was later extended by Rahli *et al.* [11] to cover almost all SML. This implied a number of innovations and generalizations that are in part specific for SML and in part applicable to other languages. Even though, they focus on a very different language, their work is probably the closer one to ours. In fact, they used a similar approach to ours to get the variable constraints, but they instrumented a system based on Damas' type inference algorithm, which is based on unification (i.e., equality) rather than on subtyping constraints. Besides this major technical difference, our final post-processing stage also allows us to identify problems not treated by their system. In particular, we can identify and report other kinds of discrepancies in the code which are related to types but are not proper type errors. For instance, we can detect that a branch in a case expression will never be executed because its associated pattern will never match the argument of the case expression, or it *could* match, but the previous patterns completely cover the type of the argument. In our setting, these problems can also be explained with the information about source positions collected by the constraint generation system.

Stuckey *et al.* [15] presented the Chameleon Type Debugger that performs a type error slicing very similar to the one by Rahli et al. but for a Haskell-style language (Chameleon) with a different overloading style. There is an interesting idea implemented by Chameleon: producing slices that are the intersection of all slices that explain a type error. These slices are in general neither complete nor minimal, but they often contain the portion of the source code with the highest probability to contain the cause of the error.

A complementary approach to *constraint-based* type error slicing is *source-based* type error slicing [13]. Instead of generating constraints associated with the source code and producing a minimum set of unsolvable constraints, this approach iteratively replaces any term t in the source code by a term that type checks in any context. If the modified program type checks or produces a different type error, then t does belong to the slice. This process is repeated for all terms until a minimal slice is produced.

The work of El Boustani and Hage [4] introduces a method for improving type error messages for generic Java programs by providing better information about type errors to the programmer (with respect to existing compilers such as Eclipse's or Sun's). In some sense, we want to do the same (improve Dialyzer's warnings), but there are important differences. First, they face the problem of parametric polymorphism in the context of the object-oriented language Java. We work with subtype polymorphism on the higher-order functional language Erlang. Moreover, they do not obtain slices of the source code. They just explain errors by improving the messages and the information contained in them.

A diploma thesis [6], supervised by the first author of this paper, had as its goal to explain Dialyzer warnings. It is therefore work that is quite similar to ours. However, that approach was not tightly integrated with the constraint generation system or the constraint solver. It basically implemented an extension to Dialyzer that was able to collect the line numbers associated with the constraints produced by Dialyzer. Although useful, line numbers are insufficient and very imprecise (e.g., they cannot usually distinguish between the arguments in a function definition). In our work, we can report the exact (sub)expressions that explain a type error.

Other approaches exist that address the problem of explaining type errors. But some of them, e.g., [19, 20], do not produce slices to explain the errors; or the slices produced are not minimal (e.g, [3, 18]). Other important differences are the programming language targeted and the underlying formalism used in the type inference system. Unlike previous works, we focus on explaining inference of success typings which is a constraint-based formalism based on subtyping. This is a context which is significantly different both from unification-based type inferencing algorithms and of type debuggers based on soft typing (such as MrSpidey/MrFlow and the subsequent work for DrScheme [5]).

3. Preliminaries

Core Erlang When an Erlang program is compiled with the Erlang/OTP compiler, it is first translated to an intermediate language called Core Erlang [1]. Core Erlang is a functional higher-order language with strict evaluation. In this paper, we will consider the following subset of Core Erlang which is expressive enough to cover a wide variety of real programs:

e	::=	$v \mid c(\overline{e_n}) \mid e(\overline{e_n}) \mid f$	(Expression)
	|	$\texttt{let}\ v = e_1\ \texttt{in}\ e_2$	
	|	$\texttt{letrec}\ \overline{v_n = f_n}\ \texttt{in}\ e$	
	|	$\texttt{case}\ e\ \texttt{of}\ \overline{gp_n \to e_n}\ \texttt{end}$	
f	::=	$\texttt{fun}\ (\overline{v_n}) \to e\ \texttt{end}$	(Function)
gp	::=	$p\ \texttt{when}\ g$	(Guarded Pattern)
p	::=	$v \mid c(\overline{p_n})$	(Pattern)
g	::=	$g_1\ (\texttt{andalso} \mid \texttt{orelse})\ g_2 \mid \texttt{true} \mid \texttt{false}$	
	|	$v_1\ (= \mid \neq \mid > \mid <)\ v_2$	(Guard)

In the following, we will use $\overline{a_n}$ to refer to the sequence $a_1, \ldots, a_n$. An expression can be a variable (variables always start with uppercase letters or _), a data constructor (e.g., a literal, a list or a tuple), an application, a function, a `let` expression, a `letrec` expression or a `case` expression. A pattern in a `case` expression is represented with a variable or a data constructor together with a guard. Guards are boolean conditions, so they can be logical operations between two guards or comparison operations between two variables; or also the special atoms `true` and `false`. In the rest of the paper, we will refer indistinguishably to an Erlang program, and its associated Core Erlang representation.

In order to uniquely identify each syntactical construct of a program, in a first step, we label the programs in such a way that each element of the program that can cause a type error (e.g., patterns, guards, expressions, etc.) is assigned an identifier that we call *source position* or just *label*.

Example 3. This is the labeling of a function:

```
(fun(X^l2) ->
   (case X^l4 of
        (2^l6 when true^l7)^l5 ->
            (case X^l9 of
                 (1^l11 when true^l12)^l10 -> a^l13
                 (2^l15 when true^l16)^l14 -> b^l17
                 (Y^l19 when true^l20)^l18 -> Y^l21
             end)^l8
    end)^l3
end)^l1 .
```

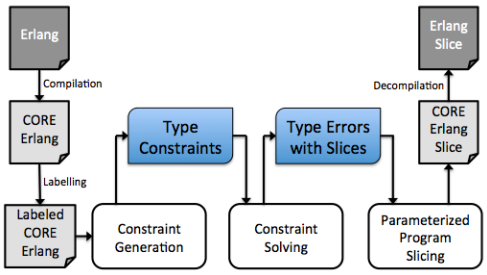

Figure 1. Block diagram of the technique

Types A type represents a set of possible values, and they are represented by type expressions. Their syntax is as follows:

$$\begin{array}{lcll} \tau & ::= & \alpha \mid c(\overline{\tau_n}) \mid (\overline{\tau_n}) \to \tau \mid \tau_1 \cup \tau_2 \mid p\tau & \text{(Type)} \\ p\tau & ::= & \mathit{none}() \mid \mathit{any}() \mid \mathit{integer}() \mid \mathit{boolean}() & \\ & \mid & \mathit{atom}() \mid \texttt{42} \mid \texttt{foo} \mid \ldots & \text{(Primitive Type)} \end{array}$$

A type expression can be a type variable—represented in the following by Greek letters (e.g., α, β, τ...) to distinguish type variables from usual variables—structured types, function types, unions of two types or primitive types. Primitive types include singleton types (e.g., `42` and `foo`) or finite (e.g., *boolean*()) and infinite (e.g., *integer*(), *atom*(), etc.) unions of sets of Erlang terms. There are also the special types *any*() and *none*() used to denote the set of all Erlang terms and the empty set of terms, respectively. We define a *concrete type* as a type that does not contain any type variable. Subtyping is represented by set inclusion, i.e., $\tau_1 \subseteq \tau_2$.

4. Explanation of Success Typing Errors

In this section we describe our technique to extract the part of a program that minimally explains a success typing error. Its three main phases are depicted in Figure 1 with white blocks: constraint generation, constraint solving and program slicing. These three phases are independent and must be executed sequentially because each phase takes as input the output of the previous.

The first phase takes as input a labeled Core Erlang program that can be automatically produced from the Erlang source after a phase of compilation and labeling. This phase automatically infers the type of each expression in the program and produces a collection of type constraints enhanced with additional information about their associated source positions. In the second phase, all constraints are solved identifying the success typing errors and combining the source positions in the constraints to collect those that are the cause of the errors. The third phase analyzes the errors and produces a final slice associated with each error that ensures minimality (redundant positions or different causes of the same type error are reduced to obtain a minimal slice that explains the type error). This phase can produce slices for all type errors, or it can take as input one particular error and produce a specific slice for it. The information provided in the second phase is rich enough as to allow the third phase to explain other kinds of discrepancies in addition to type errors. This means that the tool can detect parts of the code where something fishy may be occurring: for example code which is clearly unreachable or is code that can definitely not execute.

The rest of this section explains each phase separately.

4.1 Instrumentation of a Constraint Generation System

We start from a calculus [10] that computes the set of type constraints associated with a given (labeled) Core Erlang program. Without labels, each type constraint is a subtype relation of the form $\tau_1 \subseteq \tau_2$ where τ_1 and τ_2 represent the types of some expression in the program. We will use the shorthand $\tau_1 = \tau_2$ to denote the conjunction of constraints $\tau_1 \subseteq \tau_2$ and $\tau_2 \subseteq \tau_1$.

We will instrument the derivation rules of this calculus to get additional information from the labels. In particular, we will calculate the set of labels that have influenced each type constraint. As a result, the calculus attaches to each inferred type constraint a set of syntactic labels. Therefore, a type constraint will be a tuple formed by a subtype relation as described above, and also a set of labels that have defined this constraint; in other words, those program labels needed to infer each type appearing in the constraint and also the label of the expressions that form the constraint itself. Formally,

Definition 1 (Type constraint)**.** *Let P be a labeled Core Erlang program. A* simple type constraint *for P is a pair $(\tau_1^{l_1} \subseteq \tau_2^{l_2}, L)$ where τ_1 and τ_2 are types, l_1 and l_2 are labels, and L is a set of labels from P. A* composite type constraint *for P is a conjunction or disjunction of type constraints.*

The extended calculus shown in Figure 2 automatically produces the type constraints associated with a Core Erlang program. Each rule in this calculus contains a judgment $A \vdash e : \tau, L, C$ that should be read as "given the environment A, the expression e has type τ with associated labels L provided that the type constraint C holds". The environment A is a set of variable bindings of the form $v \mapsto \tau$ where τ is the type of v. Finally, the set of associated labels L are those needed to assign the type τ to e.

Each rule of the calculus is explained below:

`[VAR]`: If we have a variable v^l, we can consult the type τ of this variable in the environment A of the assumptions. Then, the set of labels that are assigned to v only contains l, that represents the use of variable v. No constraint is defined in this rule.

`[STRUCT]`: The type of a data constructor $c^l(\overline{e_n})$ is $c(\overline{\tau_n})$ if the type of all arguments $\overline{e_n}$ can be reduced to $\overline{\tau_n}$. The labels assigned to the final type τ are the union of all labels needed to explain the types of the arguments $(\overline{L_n})$ and the label l of the data constructor c. The constraint defined is the conjunction of all constraints defined in the arguments.

`[LET]`: The type of the `let` expression is the same type τ_2 of the expression e_2 that has been computed by adding to the environment a new binding between variable v and a fresh type variable α. Clearly, α should have the same type as τ_1 computed for the let expression e_1. This is represented with a new constraint $(\alpha^{l_v} = \tau_1^l, L_1 \cup \{l_v\})$, where the labels of the constraint are l_v of v together with the labels L_1 needed to compute τ_1. Note that this constraint is necessary to associate the labels l_v and l with types α and τ_1, and thus, if the constraint later produces an error, be able to report $v^{l_v} = e_1^l$ as responsible. The labels of the let expression are the labels computed for expression e_2 (L_2). The constraints of this rule are the constraints computed to find the types τ_1 and τ_2, together with the new constraint.

`[LETREC]`: This rule is a generalization of the previous rule, but in this case, the rule creates a new constraint for each defined variable. Each constraint states that the type $\tau_i, 1 \leqslant i \leqslant n$, of a variable v_i is equal to the type inferred τ_i' for the corresponding function f_i. The labels associated with the new constraint are those needed to infer type τ_i (l_i) and type τ_i' (L_i).

`[ABS]`: The type of a function is a fresh type variable α. In order to infer the type τ of the output of the function (e), for each parameter of the function $(\overline{v_n})$, this rule adds to the environment a new binding $\overline{v_n \mapsto \beta_n}$ where β_n are fresh type variables. To ensure that α represents a function, a new constraint $(\alpha^l = (\overline{\beta_n^{l_n}}) \to \tau^{l_e}, L \cup \bigcup \overline{l_n})$ is defined that makes α be equal

$$\frac{}{A \cup \{v \mapsto \tau\} \vdash v^l : \tau, \{l\}, \varnothing} \quad \texttt{[VAR]}$$

$$\frac{A \vdash \overline{e_n : \tau_n, L_n, C_n}}{A \vdash c^l(\overline{e_n}) : c(\overline{\tau_n}), \bigcup \overline{L_n} \cup \{l\}, \bigwedge \overline{C_n}} \quad \texttt{[STRUCT]}$$

$$\frac{A \vdash e_1 : \tau_1, L_1, C_1 \quad A \cup \{v \mapsto \alpha\} \vdash e_2 : \tau_2, L_2, C_2}{A \vdash \texttt{let}\ v^{l_v}\ =\ e_1^l\ \texttt{in}\ e_2 : \tau_2, L_2, (\alpha^{l_v} = \tau_1^l, L_1 \cup \{l_v\}) \wedge C_1 \wedge C_2} \quad \texttt{[LET]}$$

$$\frac{\overline{A \cup \{v_n \mapsto \alpha_n\} \vdash f_n : \tau_n, L_n, C_n} \quad e : \tau, L, C}{A \vdash \texttt{letrec}\ \overline{v_n^{l_n}\ =\ f_n^{l'_n}}\ \texttt{in}\ e : \tau, L, \bigwedge(\overline{C_n \wedge (\alpha_n^{l_n}\ =\ \tau_n^{l'_n}, L_n \cup \{l_n\})}) \wedge C} \quad \texttt{[LETREC]}$$

$$\frac{A \cup \{\overline{v_n \mapsto \beta_n}\} \vdash e : \tau, L, C}{A \vdash (\texttt{fun}\ (\overline{v_n^{l_n}}) \to e^{l_e}\ \texttt{end})^l : \alpha, L \cup \bigcup \overline{l_n}, C \wedge (\alpha^l\ =\ (\overline{\beta_n^{l_n}}) \to \tau^{l_e}, L \cup \bigcup \overline{l_n})} \quad \texttt{[ABS]}$$

$$\frac{A \vdash e : \tau, L, C \quad \overline{e_n : \tau_n, L_n, C_n}}{A \vdash (e^{l_e}(\overline{e_n^{l_n}}))^l : \beta, \{l\}, C \wedge (\boldsymbol{\tau}^{\boldsymbol{l_e}} = (\overline{\boldsymbol{\alpha}_{\boldsymbol{n}}^{\boldsymbol{l_e.n}}}) \to \boldsymbol{\alpha}^{\boldsymbol{l_e.0}}, L \cup \{l\}) \wedge (\beta^l \subseteq \alpha^{l_e.0}, \{l\}) \wedge \bigwedge(\overline{C_n \wedge (\boldsymbol{\tau}_{\boldsymbol{n}}^{\boldsymbol{l_n}}\ \subseteq\ \boldsymbol{\alpha}_{\boldsymbol{n}}^{\boldsymbol{l_e.n}}, L_n \cup \{l\})})} \quad \texttt{[APP]}$$

$$\frac{A \vdash p : \tau, L_p, C_p \quad g : \tau_g, L_g, C_g}{A \vdash p\ \texttt{when}\ g^{l_g} : \tau, L_p, (\texttt{true}^0\ \subseteq\ {\tau_g}^{l_g}, L_g) \wedge C_p \wedge C_g} \quad \texttt{[CPAT]}$$

$$\frac{A \vdash e : \tau, L, C_e \quad \overline{A \cup \{v \mapsto \beta_v \mid v \in Var(cp_n)\} \vdash cp_n : \tau p_n, Lp_n, Cp_n \quad b_n : \tau b_n, Lb_n, Cb_n}}{A \vdash (\texttt{case}\ e^{l_e}\ \texttt{of}\ \overline{cp_n^{lp_n} \to b_n^{lb_n}}\ \texttt{end})^l : \alpha, \bigcup \overline{Lb_n}, C_e \wedge \bigvee \overline{(\alpha^l = \tau b_n^{lb_n}, Lb_n) \wedge (\boldsymbol{\tau}^{\boldsymbol{l_e}} = \boldsymbol{\tau p}_{\boldsymbol{n}}^{\boldsymbol{lp_n}}, L \cup Lp_n) \wedge Cp_n \wedge Cb_n}} \quad \texttt{[CASE]}$$

Figure 2. Type inference calculus with source positions propagation

to a function whose parameters have types $\overline{\beta_n^{l_n}}$ and whose return value is the type of e (τ^{l_e}). The set of labels defining this constraint is the union of the arguments' labels $\bigcup \overline{l_n}$, and the labels L needed to infer the type of e.

[APP]: The type of an application is a fresh type variable β. This rule defines three new constraints to restrict this type variable: (1) In the first constraint we ensure that the type τ of e is a function whose co-domain is α. (2) In the second constraint we force the type β of the application to be a subtype of α. Finally, (3) the third constraint is really a conjunction of constraints used to ensure that the types of all the arguments of the application ($\overline{\tau_n}$) are subtypes of the corresponding types of the arguments of τ ($\overline{\alpha_n}$). Note that we use sub-labels $l_{e.n}$ for the arguments and $l_{e.0}$ for the result of the function that defines the type τ. Regarding the constraints, the first one is defined by L needed to infer τ, and l associated with the application. In the second constraint, the label is l because it represents the type of the application. Finally, the constraints for the arguments are labeled with L_i, $1 \leqslant i \leqslant n$ (needed to infer type τ_i of the argument) and l defining the application itself.

[CPAT]: The resulting type τ is defined by L_p, which defines the type of p. A new constraint for the guard ensures that it can be evaluated to `true`. The constraints defined are C_p to infer the type of the pattern and C_g to infer the type of the guard.

[CASE]: The type α of a `case` expression is defined by the union of the labels $\overline{Lb_n}$ needed to infer the types of the results in all branches τb_n. We use the notation $Var(cp_n)$ to denote the set of variables appearing in the patterns cp_n. Two constraints are generated for each branch. The first constraint restricts the type of α forcing it to be equal to the type of the branch (τb_i, $1 \leqslant i \leqslant n$). This constraint is defined by Lb_i. The second constraint makes the type τ of e equal to the type of the pattern of this branch τp_i. The constraint is defined by L which is defining the type of e and Lp_i that defines the type of the pattern p_i.

Example 4. Consider the code in Example 3. The rules of Figure 2 would infer the following information:

$((\alpha^{l_1} = (\beta^{l_2}) \to \gamma^{l_3}, \{l_2, l_{13}, l_{17}, l_{21}\}) \wedge (\beta^{l_4} = 2^{l_5}, \{l_4, l_6\}) \wedge$
$(\texttt{true}^0 \subseteq \texttt{true}^{l_7}, \{l_7\}) \wedge (\gamma^{l_3} = \delta^{l_8}, \{l_{13}, l_{17}, l_{21}\})) \wedge$
$(((\beta^{l_9} = 1^{l_{10}}, \{l_9, l_{11}\}) \wedge (\texttt{true}^0 \subseteq \texttt{true}^{l_{12}}, \{l_{12}\}) \wedge (\delta^{l_8} = \texttt{a}^{l_{13}}, \{l_{13}\}))$
$\vee ((\beta^{l_9} = 2^{l_{14}}, \{l_9, l_{15}\}) \wedge (\texttt{true}^0 \subseteq \texttt{true}^{l_{16}}, \{l_{16}\}) \wedge (\delta^{l_8} = \texttt{b}^{l_{17}}, \{l_{17}\}))$
$\vee ((\beta^{l_9} = \varepsilon^{l_{18}}, \{l_9, l_{19}\}) \wedge (\texttt{true}^0 \subseteq \texttt{true}^{l_{20}}, \{l_{20}\}) \wedge (\delta^{l_8} = \varepsilon^{l_{21}}, \{l_{21}\})))$

Observe that the type constraint generated contains a disjunction. This happens because one of the `case` expressions in the function has three branches, thus, we have a constraint generated for each branch. Observe the use of type variables in the constraints. For instance, consider the second line with the constraint ($\beta^{l_4} = 2^{l_5}$, $\{l_4, l_6\}$). It was generated because variable `X` is assigned to `2` in the `case` branch; β represents `X`, and thus they both must have the same type (i.e., `2`).

In the following we assume the existence of a set *TVars* that contains all type variables generated for a given program by the type inference calculus.

4.2 Constraint Solver Instrumented for Slicing

After having computed all the type constraints associated with a given program, we need a constraint solver to identify subsets of constraints that, taken together, are incompatible. We want these sets of unsolvable constraints to be minimal, so that we can identify the exact parts of the program that are actually needed to cause the type error. In addition, the constraint solver must be able to handle the labels associated with the constraints in order to collect the final set of labels related to the type error.

Because we want our final slices to be minimal, we need to provide a formal notion of minimality for type error slices.

Definition 2 (Type error slice)**.** *Let P be a program. A* slice *of P is a set of labels identifying program points. Let S be a slice and let C be the type constraint associated with S. If C is unsolvable, then we say that S is a* type error slice*. A type error slice S with*

associated type constraint C is minimal *if for all slice $S' \subset S$ with associated type constraint C', C' is solvable.*

Our slices are associated with constraints generated by our calculus in Figure 2. We can ensure minimality by collecting *only* those labels related to the part of the code that produced the incompatible types, and also the labels needed to produce the association between these types. Let us explain this with an example.

Example 5. Consider the following program that, for clarity and simplicity, uses Erlang instead of Core Erlang. At the right of each comment, we explain the *success typing* of the variables, which is the set of values that avoid a type error.

```
1  main(X,Y) ->  % type(X) = type(Y) = any()
2    f1(X),      % type(X) = {1,2,3}
3    f2(X),      % type(X) = {1,2}
4    f3(X),      % type(X) = {1}
5    g1(Y),      % type(Y) = {1,2,4}
6    g2(Y),      % type(Y) = {2,4}
7    g3(Y),      % type(Y) = {4}
8    X = Y.      % type(X) = type(Y) = none()

9  f1(1) -> a;  f1(2) -> b;  f1(3) -> c.
10 f2(1) -> a;  f2(2) -> b.
11 f3(1) -> a.
12 g1(1) -> a;  g1(2) -> b;  g1(4) -> d.
13 g2(2) -> b;  g2(4) -> d.
14 g3(4) -> d.
```

This program is buggy, because it results in a runtime type error: At line 8, the value of `X` can never match the value of `Y`, and the program will crash at execution time. If we observe the first line, we see that at the beginning, `X` and `Y` had the same types ($any()$). But these types have been gradually restricted by successive calls to functions with these variables as arguments.

If we assume that there is only one error in the code, then, this is due to one of two possibilities: either the type of `X` or the type of `Y` was too restricted during the execution. If the value of `X` at line 8 is correct, then lines 6 and 7 are the culprit because they restrict `Y` too much. In contrast, if the value of `Y` is correct at line 8, then lines 2, 3 and 4 are the culprit because they restrict `X` too much.

If we only concentrate on the code of function `main/1` and ignore the other code, we have the following four minimal slices:

s_1 = [2,7,8] s_2 = [3,7,8] s_3 = [4,6,8] s_4 = [4,7,8]

Our constraint solver is defined in Algorithm 1. Given a type constraint, it produces a solution to it assigning a type to each type variable. In the case that it is unsolvable, it returns a type error by collecting the labels associated with the unsatisfiable constraints.

In Algorithm 1, $\bot(E)$ represents a set of type errors (i.e., no solution exists for the type constraint), where E is a set of unsolvable constraints represented with tuples that include: (i) an unsatisfiable simple type constraint, (ii) those labels of the program that make this constraint unsatisfiable, and (iii) the solution computed so far until the error was found. Essentially, the algorithm starts with a naive solution *Sol* and it recursively calls function `solve` to refine this solution until a type error has been found (i.e., it returns $\bot(E)$) or until the solution cannot be refined more (i.e., it returns *Sol*).

The data structure *Sol* is the key of the algorithm. It contains the information that is being updated in every iteration and that represents the solution to the constraints. And it also contains the information related to the type error when it exists. In particular, *Sol* is a mapping from type variables to a pair that contains their current concrete type, and the history of previous concrete types with the labels that were needed to define these types. Moreover, we include in *Sol* a special mapping from $\bot$ to the set E. It represents a set of type errors where E contains unsolvable constraints denoted with tuples as described before.

Initially, all labeled expressions in the program P are mapped to the type $any()$ except type variables involved in recursion that are mapped to type $none()$. This allows the constraint solver to address mutually recursive calls [10]. $\bot$ is initially mapped to $\varnothing$. Given a type variable α, $Sol(\alpha)$ returns from the mapping *Sol*, a pair with the type associated with α, and the history of types of α (represented in the algorithm with His_α). Let us explain the use of *Sol* with an example.

Example 6. Here again, we use Erlang instead of Core Erlang for clarity. The code has comments showing the information contained in *Sol*. Only expressions that are of interest have been labeled.

```
main(X,Y) ->   %  Sol(α) = Sol(β) = (any(), ∅)
  f^l1(X^l2),      % Sol(α) = ({1,2,3},[{1,2,3},L1])
  g^l3(X^l4),      % Sol(α) = ({1,2},[({1,2},L2),({1,2,3},L1)])
  h^l5(Y^l6).      % Sol(β) = ({3},[({3},L3)])
  case X^l7 of     % Sol(α) = ({1,2},[({1,2},L2),({1,2,3},L1)])
    1^l8  -> X     % Sol(α) = ({1},[({1},L4),({1,2},L2),({1,2,3},L1)])
    Y^l9  -> a     % Sol(⊥) = {(α = β, {l7,l9} ∪ L2 ∪ L3)}
  end.

f^l9(1^l10) -> a; f^l11(2^l12) -> b; f^l13(3^l14) -> c.
g^l15(1^l16) -> 1; g^l17(2^l18) -> 2.
h^l19(3^l20) -> 1.
```

where

$L_1 = \{l_1, l_2, l_9, l_{10}, l_{11}, l_{12}, l_{13}, l_{14}\}$ $L_3 = \{l_5, l_6, l_{19}, l_{20}\}$
$L_2 = \{l_3, l_4, l_{15}, l_{16}, l_{17}, l_{18}\}$ $L_4 = \{l_7, l_8\}$

First, we can assume that type variables α and β represent the type of `X` and `Y` respectively. Initially, both `X` and `Y` are assigned the type $any()$ and have associated an empty set of labels. After the call to `f(X)`, `X` is assigned the type $\{1,2,3\}$ because these are the only values that can lead to success after the call to `f(X)`. This type is stored in the `X`'s history of types together with the positions L_1 that forced this type. After the call to `g(X)`, the type of `X` is further restricted and thus the new type is $\{1,2\}$. This type is stored in the `X`'s history of types with associated positions L_2. Observe that the history also keeps the previously stored type. After the call to `h(Y)`, the type of `Y` is restricted and thus its new type is $\{3\}$. The use of X in the case expression does not implies any restriction of the type of `X`. Therefore, *Sol* remains unchanged. In the first branch of the case expression, the type of `X` is restricted to $\{1\}$ and it is stored in the history of types as before. Finally, when `Y` is used as a pattern in the case expression, the equality `X = Y` is forced. But this equality will never hold because their types are incompatible (i.e., `X` has type $\{1,2\}$ and `Y` has type $\{3\}$). Therefore, a type error is detected at this point. The information recorded for the type error is the unsolvable type constraint (i.e., $\alpha = \beta$), and the labels that produced this type error (in this case, those associated with the unsolvable constraint ($\{l_7, l_9\}$) plus those associated with the type of β (L_3) plus those associated with the type of α the first time in its history of types that it was incompatible with the type of β (L_2)).

Algorithm 1 is based on function `solve` that is implemented with four rules. The first rule is used when a type error is detected, hence the information of the type error is returned $\bot(E)$. The second rule is used for simple type constraints, the third rule is used for conjunctions, and the fourth rule is used for disjunctions. The last three rules implement complex functionality which we explain:

- In the case of simple type constraints, three cases are possible: (i) when the constraint is solved we return the solution computed so far. Note that, to know that the constraint is satisfied, we use τ_i' instead of τ_i because τ_i' contains the solution found (in *Sol*) to τ_i in the case that τ_i is a type variable. (ii) When the constraint is solvable but to be solved it needs to restrict one type of the constraint, then we update (*Sol*) with the restricted type. In this case, τ_1 must be restricted to be a subset of τ_2. And,

Algorithm 1 Constraint Solver

Input: A type constraint C
Output: A solution for C or a type error if C cannot be solved
Preconditions: *Sol* is a mapping from all type variables to a tuple with the type *any*() (except recursion type variables that are mapped to type *none*()), and an empty history of types ($\varnothing$)
return `solve`(*Sol*)

where function `solve` is defined as:

$$\texttt{solve}(\bot(E), \_) = \bot(E)$$

$$\texttt{solve}(Sol, (\tau_1^{l_1} \subseteq \tau_2^{l_2}, L)) = \begin{cases} Sol & \text{when } (\tau_1' \sqcap \tau_2' = \tau \neq none() \wedge \tau_1 \notin TVars) \vee (\tau_1' \subseteq \tau_2') \\ Sol[\tau_1 \mapsto (\tau, ((\tau, L) : His_1))] & \text{when } (\tau_1' \sqcap \tau_2' = \tau \neq none() \wedge \tau_1 \in TVars) \\ \bot(\{(\tau_1 \subseteq \tau_2, \texttt{incomp}(\tau_1', His_2) \cup L_1 \cup L, Sol)\} \cup Sol(\bot)) & \text{otherwise} \end{cases}$$

$$\text{where } (\tau_i', His_i) = \begin{cases} (\tau_i, [(\tau_i, \{l_i\})]) & \text{when } \tau_i \notin TVars \\ Sol(\tau_i) & \text{when } \tau_i \in TVars \end{cases} \quad \wedge\ L_1 = \begin{cases} L_1' & \text{when } His_1 = ((\_, L_1') : \_) \\ \varnothing & \text{otherwise} \end{cases}$$

$$\texttt{solve}(Sol, \bigwedge \overline{C_n}) = \begin{cases} Sol & \text{when } \texttt{solve\_conj}(Sol, \bigwedge \overline{C_n}) = Sol \\ \texttt{solve}(Sol', \bigwedge \overline{C_n}) & \text{when } \texttt{solve\_conj}(Sol, \bigwedge \overline{C_n}) = Sol' \neq Sol \end{cases}$$

$$\texttt{solve}(Sol, \bigvee \overline{C_n}) = \begin{cases} Sol'[\bot \mapsto Sol(\bot) \cup Errors] & \text{when } Sols \neq \varnothing \\ \bot(Sol(\bot) \cup Errors) & \text{when } Sols = \varnothing \end{cases} \quad \text{where } \begin{cases} S = \{\texttt{solve}(Sol, C_i) \mid 1 \leqslant i \leqslant n\} \\ Errors = \bigcup_{\bot(E) \in S} E \\ Sols = \{s \mid s \in S, s \neq \bot(\_)\} \\ Sol' = \dot{+}(Sol, Sols) \end{cases}$$

$$\texttt{solve\_conj}(\bot(E), \_) = \bot(E)$$
$$\texttt{solve\_conj}(Sol, Conj) = \texttt{solve\_conj}(\texttt{solve}(Sol, C_1), C_2 \wedge \ldots \wedge C_n) \text{ where } \bigwedge \overline{C_n} = Conj$$
$$\texttt{solve\_conj}(Sol, C) = \texttt{solve}(Sol, C)$$

moreover, τ_1 must be a type variable. Then, we update *Sol* with the new type of τ_1. (iii) When the constraint is unsolvable, we return a type error. It is important to remark the way in which the algorithm computes the labels associated with the type error. Essentially, the labels collected are those related to the unsolvable type constraint (L), the labels needed to define the last type computed for τ_1 (L_1), and the first type computed for τ_2 that is incompatible with the type of τ_1. This is done with the auxiliary function `incomp` defined in Figure 3 (`incomp`(τ_1', His_2)).

- If the constraint is a conjunction, the algorithm solves all constraints participating in the conjunction with function `solve_conj`. Each time a constraint is solved, the resultant *Sol* is passed to be used in the next constraint. This process is repeated until *Sol* cannot change anymore (e.g., until a fix point is reached).
- Disjunctions represent the constraints imposed by the branches of a case expression. First, each branch is analyzed separately, and the union of all the solutions obtained for each branch is stored in S. Then, S is divided into two disjoint subsets with the errors found (*Errors*) and the solutions found (*Sols*). We have two possibilities: (i) If none of the branches can be solved ($Sols = \varnothing$), then, all the type errors (*Errors*) found in the branches are combined together with the errors ($Sol(\bot)$) already present in *Sol* and returned. (ii) If at least one solution exists, then all the solutions computed for each branch are also combined and returned. In this case, the combination is done with the auxiliary function $\dot{+}$ defined in Figure 3 that essentially performs two tasks. First, it assigns to each variable the least upper bound $\sqcup$ (or supremum) of the types in each branch. For instance, if a type variable α has type `1` in one branch, and has type `2` in other branch, then the final type of α is $\{\texttt{1}, \texttt{2}\}$. Second, it combines the histories of all branches. This task is done using two functions: Function `join` builds a single history which contains the combination of all histories of the branches. Function `insert_his` is in charge of orderly inserting these new entries in the previous history of the type variables.

Example 7. Continuing our example, if we apply Algorithm 1 with the type constraint in Example 4, we get the following solution:

$$\begin{aligned} Sol = \{&\bot \mapsto \{(\beta^{l_9} = \texttt{1}^{l_{11}}, \{l_4, l_6, l_9, l_{11}\}, \{\beta \mapsto \texttt{2}, [(\texttt{2}, \{l_4, l_6\})]\})\}, \\ &\alpha \mapsto (\ (\beta) \to \gamma, [(\ (\beta) \to \gamma, \{l_2, l_{13}, l_{17}, l_{21}\}\ )]\ ), \\ &\beta \mapsto (\texttt{2}, [(\texttt{2}, \{l_4, l_6\})]), \\ &\gamma \mapsto (\texttt{2} \cup \texttt{b}, [(\texttt{b}, \{l_{17}\}), (\texttt{2}, \{l_{21}\})]), \\ &\delta \mapsto (\texttt{2} \cup \texttt{b}, [(\texttt{b}, \{l_{17}\}), (\texttt{2}, \{l_{21}\})]), \\ &\varepsilon \mapsto (\texttt{2}, [(\texttt{2}, \{l_9, l_{15}\})])\} \end{aligned}$$

This is not a type error, but an actual solution that assumes that the first branch of the innermost `case` expression is executed. A type error was detected in the first branch and stored in *Sol* associated with $\bot$. This is done by the algorithm as follows: (1) It starts with a call to `solve`$(Sol, Conj)$ being *Conj* the initial type constraint. Then, the contraints are solved individually. One of them is a disjunction. Therefore, each part of this disjunction (i.e., each branch of the `case` expression) is evaluated separately. (2) In the second and third parts of the disjunction, `solve` finds a solution assigning type `2` to type variable ε, and types `b` and `2` to type variable δ. (3) In the first part of the disjunction, `solve` evaluates a conjunction, but in this case, it finds an error in the third iteration (when evaluating $(\beta^{l_9} = \texttt{1}^{l_{11}})$ and after having evaluated $(\beta^{l_4} = \texttt{2}^{l_5})$). Observe in the history of types associated with the type error that β was bound to `2` and later to `1`, but these types are incompatible. (4) After solving the disjunction, the different types assigned to δ in the success branches (`b` and `2`) are joined by function $\dot{+}$. This type is exactly the same assigned to γ.

4.3 Producing Slices for Type Errors

After the second phase, we have a data structure *Sol* with information about type errors and also about the solution computed for type variables before[2] the detection of the type errors. This infor-

[2] Even though our formalization stops when the first type error is found, it is trivial to extend the algorithm in order to find all type errors and the complete solution. This extended version is the behavior of our implementation.

$$\texttt{incomp}(\tau, His) = \begin{cases} \varnothing & \text{when } His = [] \\ L & \text{when } His = His' + [(\tau', L)] \wedge \tau \sqcap \tau' = none() \\ \texttt{incomp}(\tau, His') & \text{when } His = His' + [(\tau', L)] \wedge \tau \sqcap \tau' \neq none() \end{cases}$$

$$\left\lfloor + \right\rfloor (Sol, Sols) = Sol[\alpha \mapsto (\tau_\sqcup, His_\sqcup)] \text{ for each } \alpha \text{ in } TVars(Sols),$$
$$\text{where } (\tau_\alpha, His_\alpha) = Sol(\alpha) \wedge \tau_\sqcup = \bigsqcup \{\tau_{Sol'} \mid Sol' \in Sols \wedge (\tau_{Sol'}, \_) = Sol'(\alpha)\} \wedge$$
$$His_\sqcup = \texttt{insert\_his}(\texttt{join}(\{His_{Sol'} \mid Sol' \in Sols \wedge (\_, His_{Sol'} + His_\alpha) = Sol'(\alpha)\}, []), His_\alpha)$$

$$\texttt{join}(His, His_{accum}) = \begin{cases} \texttt{join}(His', \texttt{join\_one}((\tau, L), His_{accum})) & \text{when } (\tau, L) : His' = His \\ His_{accum} & \text{otherwise} \end{cases}$$

$$\texttt{join\_one}((\tau, L), His) = \begin{cases} (\tau \sqcup \tau', L \cup L') : \texttt{join\_one}((\tau, L), His') & \text{when } (\tau', L') : His' = His \\ [] & \text{otherwise} \end{cases}$$

$$\texttt{insert\_his}(His_{inserted}, His) = \begin{cases} \texttt{insert\_his}(His'_{inserted}, \texttt{insert}((\tau, L), His)) & \text{when } (\tau, L) : His'_{inserted} = His_{inserted} \\ His & \text{otherwise} \end{cases}$$

$$\texttt{insert}((\tau, L), His) = \begin{cases} His & \text{when } (\tau_H, \_) : His' = His \wedge \tau = \tau_H \\ (\tau_H, L_H) : \texttt{insert}((\tau, L), His') & \text{when } (\tau_H, L_H) : His' = His \wedge \tau \subseteq \tau_H \\ (\tau, L) : His & \text{when } (\tau_H, \_) : His' = His \wedge \tau \supseteq \tau_H \\ [(\tau, L)] & \text{otherwise} \end{cases}$$

Figure 3. Auxiliary functions

$$\texttt{slicingErrors}(Sol) = \bigcup_{e \in E} \texttt{sliceError}(e) \text{ where } Sol = \bot(E) \vee Sol(\bot) = E$$

$$\texttt{sliceError}((\tau_1^{l_1} \subseteq \tau_2^{l_2}, L, Sol)) = \begin{cases} \{(\texttt{e1}(l, i, \tau'_1, \tau'_2), L)\} & \text{when } (e^{l_e}(\overline{e_n^{l'_n}}))^l \in P \text{ and } l_1 = l'_i \wedge l_2 = l_e.i \text{ with } 1 \leqslant i \leqslant n \\ \{(\texttt{e2}(l, n, \tau'_i), L)\} & \text{when } (e^{l_e}(\overline{e_n^{l'_n}}))^l \in P \text{ and } l_i = l_e \wedge \tau'_j = (\overline{\tau''_m}) \to \tau''_r \wedge m \neq n \\ & \qquad \text{with } (i, j) \in \{(1,2),(2,1)\} \\ \{(\texttt{e3}(l, p_k, \tau'_i, \tau'_j), L)\} & \text{when } (\texttt{case } e^{l_e} \texttt{ of } \overline{p_n^{l_{p_n}} \to b_n} \texttt{ end})^l \in P \text{ and } l_i = l_e \wedge l_j = l_k \\ & \qquad \text{with } 1 \leqslant k \leqslant n \wedge (i, j) \in \{(1,2),(2,1)\} \\ \ldots & \ldots \\ & \text{where } \tau'_i = \begin{cases} \tau_i & \text{when } \tau_i \notin TVars \\ Sol(\tau_i) & \text{when } \tau_i \in TVars \end{cases} \end{cases}$$

Figure 4. Algorithm for producing warnings from the unsatisfiable constraints

mation can be exploited to detect the different causes of the type errors, and also to produce their associated slices. This process can be done automatically.

When function `solve` detects a type error (i.e., it returns $\bot(E)$), we can identify the cause (i.e., the slice) associated with this type error thanks to the information collected. However, as mentioned, the same information also allows Dialyzer to detect and explain why some code is dead or unreachable. Such code is often confusing, it pollutes the source, and can cause maintenance problems.

Example 8. Consider the following Erlang code:

```
main(X) ->                  f(one) -> a;
  Y = f(X),                 f(_) -> b.
  case Y of
    a -> 1;
    b -> 2;
    _ -> 42
  end.
```

This program will never crash at execution time. However, for the current definition of function `f`, the third branch of the `case` expression will never be executed. This is due to the fact that the first two branches completely cover all possible values of `Y`. The presence of such code, although not a programming error, often indicates programmer confusion or misunderstanding of APIs, in case `f` is a library function or a function from another module.

Thanks to the information computed by function `solve` we can identify problems produced by a bad use of types as the one in the previous example and produce a slice to explain them. Therefore, we have two different methods; one to identify type errors and one to identify type inconsistencies. The former uses the information produced about errors ($\bot(E)$), and the later uses the information collected about solutions to type variables (*Sol*). Both methods are formalized in Figures 4 and 5. For the sake of clarity and because both methods are conceptually distinct, we present them separately.

In Figure 4, function `slicingErrors` takes a solution produced with function `solve` and it extracts all the information related to type errors (E). All type errors are then processed separately with function `sliceError`. Therefore, function `sliceError` implements the main functionality of detecting the cause (and producing a slice for it) that produced a given type error. There are many different causes of a type error. In particular, the current version of Dialyzer is able to detect 26 different inconsistencies produced by type errors. Each inconsistency should be reported to the user together with a slice explaining the cause. This is the objective of function `sliceError`. For simplicity, we have formalized here only three different type errors with their associated slices (that are the second output produced by `sliceError`):

- Fun application with arguments $e(\overline{e_n})$ will never return since it differs in the *i*st argument from the success typing argu-

$$\texttt{slicingSol}(Sol) = \{\texttt{sliceSol}(Sol') \mid (Sol = \bot(E) \wedge (\_,\_,Sol') \in E) \vee (Sol \neq \bot(\_) \wedge Sol' = Sol)\}$$

$$\texttt{sliceSol}(Sol) = \begin{cases} \{(\texttt{e4}(l,\tau_\alpha), L)\} & \text{when } ([\,](e_1, e_2^{l_2}))^l \in P \text{ and } \alpha^{l_2} \in TVars \text{ and } [\,](\_,\_) \not\subseteq \tau_\alpha \text{ and } [\,] \not\subseteq \tau_\alpha \\ & \quad \text{and } L = \texttt{incomp}(\{[\,](\_,\_), [\,]\}, His(\alpha)) \\ \{(\texttt{e5}(l, p_i, \tau_\alpha \backslash \tau_{Lbl}, \tau_\beta), & \text{when } (\texttt{case } e^{l_e} \texttt{ of } \overline{p_n^{lp_n} \to b_n} \texttt{ end})^l \in P \text{ and } \alpha^{l_e} \in TVars \\ \quad L_e \cup L_i \cup L_{Lbl})\} & \quad \text{and } \exists i \in \{1..n\} \text{ and } \exists Lbl \subseteq \{1..i-1\} \text{ such that } \beta^{lp_i} \in TVars \text{ and} \\ & \quad\quad \tau_\beta \subseteq \bigsqcup\{\tau_\gamma \mid j \in Lbl \wedge \gamma^{lp_j} \in TVars \wedge \tau_\gamma \not\subseteq \bigsqcup\{\tau_\delta \mid k \in \{1..j-1\} \wedge \delta^{lp_k} \in TVars\}\} \\ & \quad \text{where } \tau_{Lbl} = \bigsqcup\{\tau_\gamma \mid j \in Lbl \wedge \gamma^{lp_j} \in TVars\} \text{ and } (\tau_\alpha, L_e) \in His_\alpha \text{ and } (\tau_\beta, L_i) \in His_\beta \\ & \quad\quad \text{and } L_{Lbl} = \bigcup\{L_j \mid j \in Lbl \wedge \gamma^{lp_j} \in TVars \wedge (\_, L_j) \in His(\gamma)\} \\ \{(\texttt{e6}(l, p_i, \tau_\alpha, \tau_\beta), & \text{when } (\texttt{case } e^{l_e} \texttt{ of } \overline{p_n^{lp_n} \to b_n} \texttt{ end})^l \in P \text{ and } \alpha^{l_e} \in TVars \\ \quad L_e \cup L_i \cup L_{Lbl})\} & \quad \text{and } \exists i \in \{1..n\} \text{ and } \exists Lbl \subseteq \{1..i-1\} \text{ such that } \beta^{lp_i} \in TVars \text{ and } \tau_\alpha \subseteq \tau_{Lbl} \text{ and} \\ & \quad\quad \tau_\beta \subseteq \bigsqcup\{\tau_\gamma \mid j \in Lbl \wedge \gamma^{lp_j} \in TVars \wedge \tau_\gamma \not\subseteq \bigsqcup\{\tau_\delta \mid k \in \{1..j-1\} \wedge \delta^{lp_k} \in TVars\}\} \\ & \quad \text{where } \tau_{Lbl} = \bigsqcup\{\tau_\gamma \mid j \in Lbl \wedge \gamma^{lp_j} \in TVars\} \text{ and } (\tau_\alpha, L_e) \in His_\alpha \text{ and } (\tau_\beta, L_i) \in His_\beta \\ & \quad\quad \text{and } L_{Lbl} = \bigcup\{L_j \mid j \in Lbl \wedge \gamma^{lp_j} \in TVars \wedge (\_, L_j) \in His(\gamma)\} \\ & \text{where } (\tau_\chi, His_\chi) = Sol(\chi) \text{ and } P \text{ is the program} \end{cases}$$

Figure 5. Algorithm for producing warnings from a solution

ments: τ_2'. Represented as `e1`, this error message identifies a function call where one of the arguments has a type that is not a subtype of the corresponding parameter of the function. This is represented in the first case of function `sliceError` by identifying a function call $e^{l_e}(\overline{e_n^{l'_n}})^l$ in the program P—for the sake of simplicity we assume that P is a global variable—where the label of τ_1 corresponds to the label of an argument e_i, and the label of τ_2 corresponds to the correct sub-label of the applied expression e in the same application. The slice associated with this type error is L. Note that we add some information to the error message. This information is the only information needed by a decompiler to identify the parts of the source code that correspond to the slice and to show the corresponding Dialyzer's error. We do not formalize here how to do this because it is just a generic and simple process of parsing and decompilation.

- Fun application will fail since $e{::}\tau_i'$ is not a function of arity n: Represented as `e2`, this error message identifies a buggy application. This happens when an expression of the incorrect type, or with the wrong number of arguments have been applied to other expressions. Note that we can identify the application in the source code thanks to the labels of the types. In this case, l_e is the label of the applied expression.
- The pattern p_k can never match the type τ_i': Represented as `e3`, this error message identifies a `case` expression where the expression being evaluated cannot match one of the branches (thus, this branch will never be executed). In this case, function `sliceError` searches for a `case` expression where the label of τ_1 is equal to the label of the matching expression e, and the label of τ_2 corresponds to some pattern p_k.

In Figure 5, function `slicingSol` takes a solution produced with function `solve` and, contrary to Figure 4, it ignores the information related to type errors, and only concentrates in the solutions computed so far, even in the case that an error ($\bot(E)$) was reported. The solutions are then processed by function `sliceSol` to identify type errors and produce slices. For simplicity, here again, we have formalized only three different type errors with their slices:

- Cons will produce an improper list since its 2nd argument is τ_α: Represented as `e4`, this error message identifies a list constructor where the inferred type of the second argument cannot be a list (i.e., a list $[X|Xs]$ where Xs is not a list). This is known because a type variable α associated to the second argument cannot be evaluated to a list (neither $[\,]$ nor $[\,](\_,\_)$). The associated slice L contains all the labels that defined the first type in the history of types that could not be a list.
- The pattern p_i can never match the type $\tau_\delta^{l_e} \backslash T_{Lbl}$: Represented as `e5`, this error message is raised when a branch of a case expression can never be executed because the types of the pattern p_i of this branch have been already covered by previous patterns. Then, the idea is to find in the `case`-expression a pattern (p_i) whose type is a subtype of the types of the previous patterns. The slice contains the labels defining the type of the pattern, the labels defining the matching expression e, and the set of labels corresponding to all the patterns p_j with $j \in Lbl$.
- The pattern p_i can never match since previous clauses completely covered the type $\tau_\delta^{l_e}$: This case is completely analogous to the previous one, with the only difference that a branch cannot be executed because the type of the `case`-expression has been already covered by the types of previous branches.

Example 9. Continuing our running example, we can extract the slices associated with the type errors and inconsistencies. If we use functions `slicingErrors` and `slicingSol` with the solution computed in Example 7, we get the following slices:

$$\texttt{slicingErrors}(Sol) = \{\texttt{e3}(l_8, \texttt{1}, \texttt{2}, 1), \{l_4, l_6, l_9, l_{11}\}\}$$
$$\texttt{slicingSol}(Sol) = \{\texttt{e6}(l_8, \texttt{Y}, \texttt{2}, \texttt{2}), \{l_4, l_6, l_9, l_{15}, l_{19}\}\}$$

The first slice, computed with the functions in Figure 4, produces a type error. In particular, the first branch of the internal `case` expression will crash because variable `X` can never match the value `1` (it is always `2`). The second slice has been computed with the functions in Figure 5. This slice corresponds to a type inconsistency: the third branch of the `case` expression will never be reached because the pattern in the second branch (`2`) has covered the whole type of `X` (`2`), and thus the second branch will be always selected.

5. Termination and Minimality

In this section we formulate two important properties of our technique: (1) our method to produce slices for type errors is a finite process (termination), and (2) the slices produced are minimal (minimality).

First, we define some auxiliary lemmata.

Lemma 1. *Let C be a type constraint not formed with disjunctions of type constraints (i.e., only formed with simple type constraints or conjunctions of constraints). Given a call* $\texttt{solve}(Sol_1, C_1)$*, for each solution Sol_2 produced with a recursive call* $\texttt{solve}(Sol', C_2)$*, performed by Algorithm 1 with C as input, and given the types and*

histories associated with an arbitrary type variable α, $Sol_1(\alpha) = (\tau_1, His_1)$ and $Sol_2(\alpha) = (\tau_2, His_2)$, then $\tau_2 \subseteq \tau_1$.

Proof. In Algorithm 1, if no disjunction exists in the input constraint C, the fourth rule of function `solve` cannot be executed. Therefore, the only case when a binding in *Sol* is updated happens in the second case of the second rule of function `solve`; where *Sol* is updated as $Sol[\tau_1 \mapsto (\tau, ((\tau, L) : His_1))]$, previously being $Sol(\tau_1) = (\tau_1', His_1)$. Moreover, the condition needed to enter in this case ensures that $\tau_1' \sqcap \tau_2' = \tau$. Therefore, trivially, $\tau \subseteq \tau_1'$. □

Lemma 2. *Given a call* $\texttt{solve}(Sol_1, \bigvee \overline{C_n})$*, for each solution* Sol_2 *produced with a recursive call* $\texttt{solve}(Sol_1, C_i), 1 \leqslant i \leqslant n$*, performed by Algorithm 1, we have that for all* $\alpha \in Tvars$*, if* $Sol_1(\alpha) = (\tau_1, His_1)$ *and* $Sol_2(\alpha) = (\tau_2, His_2)$ *then* $\tau_2 \subseteq \tau_1$.

Proof. First, note that the recursive call can only be done in the fourth rule of function `solve`. Let us prove the lemma by induction over the structural depth of the constraint $\bigvee \overline{C_n}$.
(base case) In this case, $\forall C_i, 1 \leqslant i \leqslant n$, C_i is a simple constraint. Hence, the lemma holds by Lemma 1.
(induction hypothesis) We can assume as the induction hypothesis that the constraint has a structural depth of d and the lemma holds.
(inductive case) In this case the constraint has a structural depth of $d+1$. Therefore, $\forall C_i, 1 \leqslant i \leqslant n$, C_i has a structural depth $x \leqslant d$ and the lemma holds for this constraint according to the induction hypothesis. Hence, the lemma holds for all calls to `solve` done during the computation of Sol_2. Moreover, the lemma also holds for the call $\texttt{solve}(Sol_1, \bigvee \overline{C_n})$ because all C_i must fall in one of these cases:

- C_i is a simple type constraint or a conjunction of type constraints. In this case the Lemma holds according to Lemma 1.
- C_i is a disjunction of type constraints. In this case Sol_2 is computed with function $\lfloor + \rfloor$. However, function $\lfloor + \rfloor$ assigns to each type variable α in Sol_1 the least upper bound of the set of solutions computed by each recursive call done to `solve` with C_i, and we know by the induction hypothesis that the lemma holds for all of them. Hence, we know that $\forall \alpha \in Tvars$, $Sol_1(\alpha) = (\tau_1, His_1) \wedge Sol_2(\alpha) = (\tau_2, His_2)$ then $\tau_2 \subseteq \tau_1$. □

Lemma 3. *Let P be a program, and let C be the type constraint associated with P. Given a call* $\texttt{solve}(Sol_1, C_1)$*, for each solution* Sol_2 *produced with a recursive call* $\texttt{solve}(Sol', C_2)$*, performed by Algorithm 1 with C as input, and given two bindings* $\alpha \mapsto (\tau_1, His_1) \in Sol_1$ *and* $\alpha \mapsto (\tau_2, His_2) \in Sol_2$*, then* $\tau_2 \subseteq \tau_1$.

Proof. In Algorithm 1, there are only two cases when a binding in *Sol* is updated. Therefore, it is enough to prove that, in both cases, the type associated with the new binding remains unchanged or is a subset of the previous type:

- In the second item of the second rule of function `solve`, *Sol* is updated. This case happens when the constraint is a simple constraint (no disjunction exists), and thus it is proved by Lemma 1.
- *Sol* is also updated in the fourth rule of function `solve` when at least a constraint of a disjunction has a solution ($Sols \neq \varnothing$). In this case the type variable is updated with function $\lfloor + \rfloor$. However, function $\lfloor + \rfloor$ assigns to each type variable α in *Sol* the least upper bound of the set $\{\tau_{Sol'} \mid Sol' \in Sols \wedge (\tau_{Sol'}, \_) = Sol'(\alpha)\}$, i.e. the set containing the types of α for each solution Sol' in *Sols*. By Lemma 2, we know that $\forall Sol' \in Sols\ .\ \forall(\alpha \mapsto (\tau_1, His_1)) \in Sol, (\alpha \mapsto (\tau_2, His_2)) \in Sol'\ .\ \tau_2 \subseteq \tau_1$. Therefore, $\tau_1 \subseteq \tau_2$ for any two bindings $\alpha \mapsto (\tau_1, His_1) \in Sol_1$ and $\alpha \mapsto (\tau_2, His_2) \in Sol_2$. □

Lemma 4. *Let P be a program, and let C be the type constraint associated with P. The number of errors assigned to* $\perp$ *when computing a solution using Algorithm 1 with C as input is finite.*

Proof. First, C is finite by construction according to the type inference calculus of Figure 2. Second, $\perp$ is only updated when a disjunction is being proceeded by function `solve` (rule four of function `solve`). Concretely, in the case where at least one constraint of the disjunction has a solution ($Sols \neq \varnothing$). The union of all the errors produced by unsolvable constraints is calculated, and added to the current value of $\perp$. Therefore, the number of errors is limited by the number of constraints that form the disjunction. □

Theorem 1 (Termination)**.** *Let P be a program, and let C be the type constraint associated with P. The type inference system in Figure 1 terminates with C as input.*

Proof. We can ensure termination providing an invariant of the algorithm: the size of all types assigned to type variables is always decreased or it remains unchanged. Moreover, this size cannot decrease infinitely because type $none()$ will eventually be assigned to type variables. This ensures termination because type $none()$ cannot be further decreased and the algorithm returns a solution whenever it is reached or whenever types cannot be further reduced. Let us consider an arbitrary constraint C and analyze all possible cases that can happen with a call $\texttt{solve}(C)$:

- C is a simple constraint: In this case, trivially, no recursive call exists and thus the call $\texttt{solve}(C)$ terminates.
- C is a conjunction: In this case, $\texttt{solve\_conj}(Sol, \bigwedge \overline{C_n})$ is called, and we have two possibilities: (i) if the output produced is equal to *Sol* the algorithm trivially terminates with output *Sol*. (ii) If the output produced is different to *Sol*, then a new recursive call is produced. However, by Lemmata 3 and 4 we know that *Sol* cannot be changed infinitely, because
  - the type associated with all type variables in *Sol* is always reduced or it remains the same (Lemma 3). Therefore, in the worst case type variables always decrease their types until the type $none()$ is assigned to them.
  - The number of errors stored in *Sol* cannot increase infinitely (Lemma 4).

  Hence, *Sol* cannot change infinitely and thus a fix point will be reached. This ensures that the case (i) will be eventually executed and the algorithm will terminate.
- C is a disjunction: First, we know that C is finite by construction. In particular, by the definition of rule [CASE] in the type inference calculus in Figure 2, we know that the disjunction has a finite number of constraints (one for each branch of a finite case expression). In this case, the only recursive calls are $\texttt{solve}(Sol, C_i)$ with $1 \leqslant i \leqslant n$. In the rest of cases, the algorithm trivially produces either a solution or an error, thus terminating. Therefore, we can ensure that $\texttt{solve}(C)$ terminates because each individual call $\texttt{solve}(Sol, C_i)$ terminates because it is either a simple constraint, a conjunction or a (finite) disjunction, and thus no infinite recursion is possible. □

Theorem 2 (Minimality)**.** *Let P be a program, C the type constraint associated with P, and* (C_e, L_e, Sol_e) *a type error produced by the type inference system in Figure 1 with C as input. Then,* L_e *is a minimal slice for the unsolvable constraint* C_e.

Proof. (sketch) First, C_e is of the form $\tau_1 \subseteq \tau_2$. But we know that $\tau_1 \nsubseteq \tau_2$ because it was reported as a type error by Algorithm 1. Observe that the constraint can only be violated if $Sol_e(\tau_2) = (\tau, His_2)$ with $\tau \neq none()$ and $His_2 \neq [\,]$. Moreover, $His_1 \neq [\,]$ because otherwise the condition $\tau_1' \sqcap \tau_2' = \tau \neq none()$ would hold

and an error would not be reported (one of the two first cases of the rule would be executed instead). We prove the theorem showing that L_e is (only) formed by three subsets of labels:

L_1 only contains the labels needed to assign a concrete type to τ_1.
L_2 only contains the labels needed to assign a concrete type to τ_2.
L_3 only contains the labels needed to define C_e.

Clearly, the three sets are needed to produce the type error. If we prove that $L_e = L_1 \cup L_2 \cup L_3$ then, trivially, L_e is a minimal slice for the unsolvable constraint C_e.

In Algorithm 1, all errors are produced by function `solve` in the rule for single constraints. In the third case of this rule, it builds an error of the form $(\tau_1 \subseteq \tau_2, \texttt{incomp}(\tau_1', His_2) \cup L_1 \cup L, Sol)$, where $\texttt{incomp}(\tau_1', His_2) \cup L_1 \cup L$ is the set of labels associated with the slice of the unsatisfiable constraint $\tau_1 \subseteq \tau_2$. Let us show that these sets of labels correspond to L_2, L_1 and L_3 in the theorem, respectively.

- The first set of labels $\texttt{incomp}(\tau_1', His_2)$ corresponds to the set L_2. First, τ_1' is the current concrete type of τ_1. Its value could be extracted from *Sol* or be directly τ_1 depending on whether τ_1 is a type variable or not. Second, the value His_2 also depends on τ_2. If τ_2 is a variable, then His_2 is the history of τ_2 in the current solution *Sol*. Otherwise, His_2 is a naive history with only one element that defines $\{l_2\}$ as the type of τ_2. Finally, set L_2 is the result of a call to function `incomp`, that looks for the first type τ' in the history His_2 that is incompatible with type τ_1'. When it is found (it must be found because $His_2 \neq [\,]$), the set of labels L that define τ' is the result. Concretely, all the labels that define the first type of τ_2 that is incompatible with τ_1'. Therefore, only the labels needed to define the type of τ_2 are included.
- The second set of labels L_1 corresponds to the set L_1 of the proof. It is extracted form the first element of the history His_1 of τ_1 (recall that $His_1 \neq [\,]$). If the set L_1 is extracted from His_1, then its value depends on the type τ_1. If it is a type variable, His_1 is extracted directly from the current solution *Sol*. Then its first element contains the labels needed to define the current concrete type τ_1' of τ_1. However, if τ_1 is not a type variable, a naive history is build including only l_1 in the set of labels that defines the type of τ_1. In this case, it is trivial that this set contains the labels defining the type.
- The third set of labels L corresponds to the set L_3 of the proof. It is extracted directly from the constraint, so it only contains the labels needed to define the constraint $\tau_1 \subseteq \tau_2$, i.e., C_e. □

6. Implementation

All the work we described has been implemented and combined with the analysis performed in Dialyzer v2.5 (in R15B). The main idea during the implementation was to be conservative with the previous Dialyzer behavior in such a way that we want the user to be able to use Dialyzer as always producing the usual information. Additionally, we also want to allow the user to see the slices associated with the type errors when she is interested in them. This objective has been achieved by introducing a new flag. In the extension, the user can produce the slices associated with the type errors detected by using a new command line argument `--slice`.

In order to integrate the new technique, we have modified the implementation of five different Dialyzer's modules. However, the bulk of the changes are in module `dialyzer_typesig.erl`, which is the module generating and solving the type constraints. The explanation component consists of about 2,000 lines of Erlang code. In addition, we had to implement additional functionality related to the treatment of Core Erlang. For instance, the labeling of Core Erlang programs has been changed as described in Section 3 so that each relevant expression is now identifiable in all phases and can be isolated as part of a type error slice. All the source code of our experimental prototype implementation is publicly available at: `http://www.it.uu.se/research/group/hipe/dialyzer/`.

The following example shows the output of this prototype.

Example 10. Consider the following part of an Erlang program, in a file `ex3.erl`, which is an Erlang version of Example 3:

```
 5  main(X) ->
 6    case X of
 7      2 ->
 8        case X of
 9          1 -> a;
10          2 -> b;
11          Y -> Y
12        end
13    end.
```

The new output of Dialyzer is:

```
> dialyzer --slice ex3.erl
ex3.erl:9: The pattern 1 can never match the type 2
 discrepancy sources:
   ex3.erl:6   case X of <= Expressions: X
   ex3.erl:7     2 -> <= Expressions: 2
   ex3.erl:8       case X of <= Expressions: X
   ex3.erl:9         1 -> a; <= Expressions: 1
ex3.erl:11: The variable Y can never match since previous clauses
completely covered the type 2
 discrepancy sources:
   ex3.erl:6   case X of <= Expressions: X
   ex3.erl:7     2 -> <= Expressions: 2
   ex3.erl:8       case X of <= Expressions: X
   ex3.erl:10        2 -> b; <= Expressions: 2
   ex3.erl:11        Y -> Y <= Expressions: Y
```

The original output by Dialyzer omits the "discrepancy sources" information. The new information is the slices shown after these lines. Each slice indicates the exact expressions that caused the error (e.g., in the second error explanation, the expressions reported are `2`, `X` and `Y`). This information helps understand the warning message provided by Dialyzer, especially if the program is big and the expressions are located in different functions or modules.

7. Concluding Remarks

We described a new technique to explain the cause of a definite success typing error through the use of program slices. Given an Erlang program, the technique analyzes the type information of all its expressions trying to assign a type to all of them. Whenever an expression cannot be assigned a type, the technique detects a type error. During the detection of this error, our technique propagates information along the program about the exact position in the source code of those expressions that produced the type error. With this information, it is possible to determine what exact parts of the source code should be changed to correct the type error.

Besides type errors, the technique can also use the information collected to explain the detection of dead or unreachable code.

One important characteristic of our technique is that it produces minimal slices, which are composed of all and only those expressions that contribute to a success typing error. Our implementation has been integrated into a prototype extension of Dialyzer, the Erlang's static discrepancy analyzer. For future work, we plan to extend our technique to also handle *refined* success typings [9], and integrate it into an Erlang editor so that the slices produced by Dialyzer can be directly highlighted in the original source code.

References

[1] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001.

[2] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical Report 426, Indiana University, 1995.

[3] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[4] N. El Boustani and J. Hage. Improving type error messages for generic Java. *Higher-Order and Symbolic Computation*, 24(1–2):3–39, 2011.

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

[6] E. Fragkaki. Explanation of success typing violations in Erlang programs. Master's thesis, National Technical University of Athens, 2010.

[7] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[8] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Proceedings of IFL'06*, volume 4449 of *LNCS*, 2007.

[9] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *APLAS*, volume 3302 of *LNCS*, pages 91–106. Springer, 2004.

[10] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *PPDP'06*, pages 167–178, New York, NY, USA, 2006.

[11] V. Rahli, J. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical report, Herriot Watt University, 2010.

[12] K. Sagonas, J. Silva, and S. Tamarit. Precise explanation of success typing errors (extended version). Technical report, DSIC, Universitat Politècnica de València, November 2012.

[13] T. Schilling. Constraint-free type error slicing. In *Proceedings of TFP'11*, pages 1–16, 2012.

[14] J. Silva. A vocabulary of program-slicing based techniques. *ACM Computing Surveys*, 44(3), 2012.

[15] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proceedings of Haskell'04*, pages 80–91, 2004.

[16] F. Tip. A survey of program slicing techniques. *Journal on Program. Lang.*, 3(3):121?–189, 1995.

[17] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55, 2001.

[18] M. Wand. Finding the source of type errors. In *POPL*, pages 38–43, New York, NY, USA, 1986.

[19] J. Yang. Explaining type errors by finding the source of a type conflict. In *Trends in Functional Programming*, pages 58–66, 2000.

[20] J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *IFL*, pages 71–86, 2000.