**TYPER: A Type Annotator of Erlang Code**  
(Extended Abstract)

Tobias Lindahl  
Konstantinos Sagonas

Department of Information Technology  
Uppsala University, Sweden  
{tobiasl,kostis}@it.uu.se

Abstract

We describe and document the techniques used in TYPER, a fully automatic type annotator for ERLANG programs based on constraint-based type inference of *success typings* (a notion closely related to principal typings). The inferred typings are fine-grained and the type system currently includes subtyping and subtype polymorphism but not parametric polymorphism. In particular, we describe and illustrate through examples a type inference algorithm tailored to ERLANG’s characteristics which is modular, reasonably fast, and appears to scale well in practice.

Categories and Subject Descriptors  
F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure;  
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Documentation

General Terms  
Languages, Theory

Keywords  
constraint-based type inference, success typings, subtyping, principal typings, Erlang

1. Introduction

ERLANG programs have been developed for quite some time now without containing any explicit information about types for their functions. Types, besides allowing some errors to be caught statically and early in the development cycle, are useful for software maintenance since they provide important documentation about function interfaces and explicitly state programmers’ intentions.

The effects that this lack of documentation has on software maintenance cannot be underestimated. In the best case, information about the intended types of function arguments and their results exists in the form of comments. However, experience shows that such documentation is often unreliable since comments tend to evolve in a different pace than the source and occasionally suffer for extreme code rot. On the other hand, type information which implicitly exists in the code itself is more reliable, but often hard to reconstruct in its entirety for a dynamically typed language such as ERLANG. Still, trying to do so is a worthwhile goal.

In an attempt to attack this goal, in this paper we describe and document the techniques used in TYPER, a fully automatic type annotator for ERLANG programs. Notable characteristics of TYPER are that it is completely automatic, never rejects any programs that are accepted by the BEAM compiler, is fast, scalable and reasonably precise, and performs reasonably even when only part of the code base is available.

The rest of the paper is structured as follows. In the next section we briefly review the basis of our work in order to put it into context. The next two sections form the main body of this paper describing TYPER’s design goals and basic usage (Section 3) and the type inference algorithm on which TYPER relies in Section 4 which forms the core of this paper. Consequences of inferring success typings for a language with side-effects such as ERLANG are discussed in Section 5. A taste of TYPER’s performance appears in Section 6 and the paper ends by reviewing some closely related work and with concluding remarks.

2. The Basis of our Work

2.1 The ERLANG language and ERLANG/OTP

ERLANG [2] is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution and fault-tolerance. ERLANG’s primary design goal was to ease the programming of soft real-time control systems commonly developed by the telecommunications industry.

ERLANG’s basic data types are atoms, numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. A notation for structured objects (records in the ERLANG lingo) is supported, but the underlying implementation of records is currently the same as that of tuples. To allow efficient implementation of telecommunication protocols, ERLANG nowadays also includes a binary data type (a vector of byte-sized data) and a notation to perform pattern matching on binaries. Functions are defined as ordered sets of guarded clauses, and clause selection is done by pattern matching. Explicit pattern matching against terms and clause guards, whenever present, provides information about a function’s intended input and return types.

The default compiler of ERLANG/OTP, nowadays based on the BEAM virtual machine, is a fast compiler which takes as input a single source file (*file.erl*) and produces a *file.beam* bytecode file as output. Relatively few checks for erroneous code are performed in the process: till recently, even obviously type-incorrect code (e.g. adding an atom to a number) was happily accepted as input. The situation is slowly changing, but the programs that are currently accepted by the BEAM compiler are far from being type correct and the compiler does not perform any sophisticated optimizations guided by a type analysis phase.
2.2 Dialyzer

The limited checking in that the BEAM compiler performs leaves a void in the toolkit that an ERLANG programmer can use to develop reliable code. Testing alone, no matter how thorough, cannot reveal all software bugs. In order to, at least partly, fill this void we have developed Dialyzer, a software tool that uses lightweight static analysis to detect discrepancies (i.e., software defects such as exception-raising code or hidden failures) in ERLANG code. Dialyzer has successfully been applied in real-world telecom projects with code bases ranging up to more than a million lines of code, where it has detected a significant number of discrepancies that have gone undetected during years of extensive testing. A description of Dialyzer and our experiences can be found in [5].

The ongoing development of Dialyzer has inspired new ideas and created new possibilities for exploiting type inference. Dialyzer can analyze large ERLANG code bases relatively fast, reconstruct type information available in them, identify obvious type clashes in them, and report them to the programmer. It should thus be possible to also derive similar type information not for defect detection but for automatic documentation purposes: to enhance a programmer’s understanding of what some piece of code really does and help them to maintain legacy code. To achieve this goal, we thus decided to develop a new tool, called TYPER, whose functionality and type inference technology we describe below. But we note that the type inference that TYPER is using is not the same as that has been used so far by Dialyzer.

3. TYPER: A High-level Overview

3.1 Properties

Before embarking on this project, we set the following goals:

- TYPER should take ERLANG code as-is and should never reject any programs, no matter how “obvious” type errors they may have. In short, TYPER should not act as a type checker.
- TYPER should be fully automatic, meaning that no type declarations or user annotations of interfaces need be supplied for TYPER’s use.
- TYPER should perform reasonably even when not all code or module interfaces are available. Of course, if TYPER gets access to all code, the precision of its type annotations usually improves.
- TYPER should never be wrong. Its type annotations should be conservative over-approximations. For example, the type annotations for a function’s arguments are possibly more general than the types that a function will terminate and thus return values for.
- TYPER should be fast and scalable.
- The type annotations produced by TYPER should be as precise as possible, but never more than that.

3.2 Basic usage

Currently, TYPER comes with only a command line interface. The basic usage is:

```
> typer my_module.erl
```

which will read the file and create a new file called `my_module.ann`, located in the same directory as the original file, that contains the type signatures for all functions in `my_module.erl`. If the user wishes to see the annotations directly instead of reading the file, the output can be redirected to standard out by giving the `-stdout` option. If include paths and macro definitions are needed, these can be included by specifying appropriate `-I` and `-D` options.

```
tag_1(N) when is_atom(N) -> {atom', N};
tag_1(N) when is_float(N) -> {float', N};
tag_1(N) when is_integer(N) -> {int', N};
(tag_1(N) when is_integer(N) -> {int', N};
tag_2(N) -> `not_valid'.
```

(a) Function tagging a subset of atomic types

```
tag_2(N) when is_atom(N) -> {atom', N};
tag_2(N) when is_float(N) -> {float', N};
tag_2(N) when is_integer(N) -> {int', N};
tag_2(N) -> `not_valid'.
```

(b) Function that succeeds for all inputs

Figure 1. Illustrating the need for subtyping and the type `any()`.

In the presence of complicated build scripts it can be convenient to produce the annotations from `abstract code`. Abstract code is produced by the BEAM compiler by giving it the compiler option `-debug_info`, and it resides in the `.beam` file. The command for producing annotations from abstract code is:

```
> typer --byte my_module.beam
```

The most common command-line options of TYPER are listed in the appendix.

4. TYPER: The Implementation

Even though ERLANG is a dynamically typed language, most functions are typically intended to work on arguments of some input types and return a result of some other type. This information can be either explicitly expressed by the programmer by using pattern matching, data constructors and guard-tests, or it can only be implicitly available through the use of built-in functions. For example, if two variables are used as the arguments to the built-in addition operator, then they are both constrained to be numbers or the call would fail.

Such implicit type information can be reconstructed by a type inferencer, and indeed this is the basis of the approach taken by both Dialyzer and TYPER. On the other hand, there are some aspects of TYPER’s type system that require further explanation.

4.1 The need for subtyping

As mentioned, all valid (i.e., accepted by the BEAM compiler) ERLANG programs should be handled by TYPER and the annotations produced should be reasonable. Among other things, this means that the type system needs to be able to handle subtyping and include the universal type of all ERLANG terms, denoted `any()`. Why this is so, is easily shown with an example.

Consider the ERLANG function in Figure 1(a). Since the function accepts an atom, integer or a float in each of its clauses we need to represent the union type `(atom() | integer() | float())` (for convenience also abbreviated as `(atom() | number())`).

In Figure 1(b) the function has been modified to tag atoms or numbers with their proper tag and return the `not_valid` atom for any input argument whose type is not of an atom of number. Since there are no constraints for its input argument, this function will succeed when called with any term, i.e., its input argument has as type the universal type `any()`.

4.2 The basic types of the type system

The type system used by TYPER, shown in Figure 2, includes the basic types of the ERLANG language such as atoms, integers, floats, identifiers used for interprocess communication (pids, ports and references), and binaries (a finite sequence of bits). Structured types are tuples of any non-negative integer arity. This currently includes all user-defined records (tuples whose first argument is tagged by an atom). The other structured data type of ERLANG, namely lists, is given a special treatment since lists are very commonly used in
In general, the same idea is used for all other types. Whenever possible, the type system maintains detailed information about the specific type of some argument position or return value. For example, the analysis might infer that a function’s argument can be the integer -1 or the integer 42, denoted as (-1 | 42) and this is what the type union operation (T1 | T2) is used for. If such detailed information is either not available or reaches a certain threshold which could cause the analysis to explode, the information is collapsed to a more general type. For example, in this case we could collapse (-1 | 42) to the type integer(). A possible future extension to the widening of integers is to maintain ranges of the form \(-1..42\).

As can be seen in Figure 3, the most general type is the type \(\text{any}()\). The type \(\text{none}()\) denotes that the type analysis has determined that there is no type possible for this argument position or return value, which typically denotes the presence of a type error. For example, the return value of the following function has type \(\text{none}()\).

\[
\text{weird()} \rightarrow 'gazonk' + 42.
\]

Finally, some types are so commonly used, that we create special type aliases for them, shown in Table 1. These aliases are typically used for producing a prettier representation of TypER’s output.

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Type Alias for</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{any}())</td>
<td>({'\text{true}', 'false'})</td>
</tr>
<tr>
<td>(\text{bool}())</td>
<td>(\langle \text{integer()}, \text{float()} \rangle)</td>
</tr>
<tr>
<td>(\text{number}())</td>
<td>(\langle \text{pid}(), \text{port}(), \text{ref()} \rangle)</td>
</tr>
<tr>
<td>(\text{identifier}())</td>
<td>(\text{list}(T))</td>
</tr>
<tr>
<td>(T)</td>
<td>(\text{list}(T))</td>
</tr>
<tr>
<td>(T\ldots)</td>
<td>(\text{nonempty_list}(T))</td>
</tr>
<tr>
<td>(\text{function}())</td>
<td>(\langle \ldots \rightarrow \text{any}() \rangle)</td>
</tr>
<tr>
<td>(\text{string}())</td>
<td>(\text{list}(\text{char}()))</td>
</tr>
<tr>
<td>(\text{nonempty_string}())</td>
<td>(\text{nonempty_list}(\text{char}()))</td>
</tr>
</tbody>
</table>

Table 1. Common type aliases.

### 4.3 Success typings and their relation to principal typings

We define the basis of the inferred type signatures in terms of success typings, i.e., types that in principle could be accepted by a function and would return a result. Success typings share a lot of properties in common with principal typings [4]. Chief among them is that their type inference is compositional (i.e., each program fragment’s analysis result does not depend on its lexical context) and can be done on a modular (i.e., component-by-component) basis which in turn allows for separate type inference and compilation.

For example, when inferring success typings, the type signature for the function in Figure 1(a) is:

\[
\text{tag	extunderscore 1/1} :: (\text{atom}() \mid \text{number}()) \rightarrow \{\text{atom}', \text{atom}()\} \\
\mid \{\text{float}', \text{float}()\} \\
\mid \{\text{int}', \text{integer}()\}
\]

which in words means that its input argument must be a subtype of \(\langle \text{number}(), \text{atom}() \rangle\) in order for the function to return a pair (i.e., two tuple) whose first element is the atom ‘atom’ and its second element is an atom, or its first element is the atom ‘float’ and the second is a float, or its first element is the atom ‘int’ and the

---

ERLANG code. Their type comes in various forms (see Figure 2) depending on how much information can be inferred about their structure (i.e., whether they are empty or not, nil-terminated or not, etc). Also note that lists are the only recursive data type of the type system in the sense that \(\text{list}(T)\) is really a shorthand for the recursive type \(\langle [] | T['\text{list}(T)'] \rangle\).

The type system also includes \(\text{fun}s\), i.e., functions with either a known or unknown number of input arguments. If the number of arguments, \(n \geq 0\), is known then the input arguments are denoted as \(T_1, \ldots, T_n\) where \(T_1, \ldots, T_n\) denote their respective types. If the number of arguments is unknown but it is known that the function return type is \(T\), then the function is denoted by \(\langle \ldots \rightarrow T \rangle\). Note that \(T\) can also be \(\text{any}()\).

Figure 3. The lattice of type values.
We use a type inference algorithm where types of all variables are applied their most general type signature. We have good knowledge about its input argument. The input argument is collapsed to the second is an integer. Note that the type system we employ loses the information that the input argument and the second element of the returned pair is the same constant.

Similarly, for the function in Figure 1(b), nothing can be said about its input argument. The input argument is collapsed to the type any() and its type signature is:

\[
\text{tag}_2/1 : (\text{any}()) \rightarrow \{\text{'atom'}, \text{atom}()\} \\
\mid \{\text{'float'}, \text{float}()\} \\
\mid \{\text{'int'}, \text{integer}()\} \\
\mid \text{'not\_valid'}
\]

We will henceforth denote the type any() with an underscore (its shorthand from Table 1) in deeply nested type arguments.

### 4.4 Inferring type annotations

We use a type inference algorithm where types of all variables are represented using constraints.

#### 4.4.1 Definitions

The type of an expression, \(e\), is denoted with \(\tau_e\). The constraints are expressed with the subtyping relation (denoted with \(\subseteq\)) and with equality constraints (denoted with =), which is really a bidirectional subtyping relation (\(\tau_{e_1} \subseteq \tau_{e_2} \land \tau_{e_2} \subseteq \tau_{e_1}\)).

A conjunction of constraints is denoted \(C_1 \land \ldots \land C_n\) and a disjunction is denoted \(C_1 \lor \ldots \lor C_n\). Recall that a conjunction of constraints is a set of constraints where each constraint must hold for the constraint to be satisfiable, and a disjunction of constraints is a set of constraints where at least one of the constraints must hold. Typically conjuncts are generated for straight-line code (i.e., code containing matching statements or function calls), and disjunctions are generated by choices in the code (e.g., when a function has multiple clauses as in the code of Figure 1(a)) or when some of its clauses contain branching statements: case, if or receive as in the code of Figure 4(a)). The disjunctions and conjunctions can of course be deeply nested, generated for example from nested case statements.

#### 4.4.2 Constraint generation

We will not give a formal definition of the constraint generation, but to give an intuition we will describe some of the basic expressions.

A call to a function with known signature constraints the call site arguments to be subtypes of the arguments of the signature and the destination is a subtype of the signature range. For example, the built-in function \(\text{length}/1\), which given a proper (i.e., nil-terminated) list as argument computes its length, has the signature:

\[
\text{length}/1 : (\text{list}()) \rightarrow \text{integer}()
\]

Thus, the expression

\[
N = \text{length}(L)
\]

yields the type constraints

\[
\tau_N \subseteq \text{integer}() \land \tau_N \subseteq \text{list}()
\]

**Erlang** has a number of built-in functions (BIFs). These include both basic functions such as arithmetic operations, but also commonly used functions such as certain list-manipulating functions (e.g., \(\text{length}/1\), \(\text{append}/2\), etc.). These BIFs are typically implemented in C for efficiency, so their type signatures cannot be derived using type inference. Instead, their type signatures are hard-coded in **Type**. For some BIFs we can do better than simply apply their most general type signature. We have good knowledge about how for example the arithmetic operations behave. Consider addition, \(+/2\). The most general typing of this function is

\[
+/2 : (\text{number}(), \text{number}()) \rightarrow \text{number}()
\]

but we also know that adding two integers will yield a new integer, but if at least one operand is a float the result will also be a float. Furthermore, since the type system accommodates enumerated types for integers, such as the integer 1, we would also expect the type inference to find that adding, for example, the integers 1 and 2 yields the integer 3. The constraints for the expression

\[
X = Y + Z
\]

are

\[
\tau_X \subseteq \text{number}() \land \tau_Y \subseteq \text{number}() \land \tau_Y \subseteq \text{number}() \\
\land \tau_X \leq \tau_{Y+Z} \land \tau_Y \leq \tau_{X-Z} \land \tau_Z \leq \tau_{X-Y}
\]

where the first part assures that the variables are all numbers and the second part makes use of the specific knowledge of addition.

The general form of a case expression is

\[
\text{case } E \text{ of } \\
P_1 \text{ when } G_1 \rightarrow B_1; \\
\vdots \\
P_N \text{ when } G_N \rightarrow B_N \\
\text{end}
\]

where \(E\) is an expression to be matched against the patterns, \(P_i\) are patterns, \(G_i\) are guard expressions, and \(B_i\) are clause bodies. This case expression yields the following constraints:

\[
\tau_E \land \bigg( \bigg( \bigwedge_i \tau_E \subseteq \tau_{P_i} \land \tau_{G_i} \land \tau_{B_i} \land \tau_{\text{out}} \subseteq \tau_{B_i} \bigg) \bigg)
\]

where \(\tau_{E}\) are the constraints generated by the expression \(x\) and \(\tau_{\text{out}}\) is the type of the whole case expression. Intuitively, the type of the case expression is the least upper bound of the types of the clause bodies. Also, in order for the clause body to have a type, the expression \(E\) must be a subtype of the corresponding clause pattern and the clause guard must be satisfiable. The constraint generation for case expressions is easily generalized to fit other expressions containing multiple clauses (e.g., receive statements and functions with multiple clauses).

#### 4.4.3 Solving the constraints

When solving the constraints we could transform the constraint into disjunctive normal form, i.e., a disjunction of conjunctions, \(C_1 \lor \ldots \lor C_n\) where each \(C_i\) denotes a conjunction of constraints, and then solve each conjunction. However, doing so runs the risk of a possible explosion. To avoid this explosion we have chosen not to do this transformation, effectively making the analysis path-insensitive, thereby gaining time at the cost of precision.

Solving conjunctive constraints is more or less straightforward. A type is the greatest lower bound of its subtype constraints. To solve a disjunction, all its parts are solved and then the solution is the least upper bound of the solutions to each disjunctive part.

Consider the function in Figure 4(a). The case expression generates the constraints:

\[
(\tau_x \subseteq \text{number}() \land \tau_{\text{out}} \subseteq \text{true}) \lor (\tau_{\text{out}} \subseteq \text{false})
\]

The solutions to each of the two conjunctions are trivial. Note that \(\tau_x\) is unrestricted in the second conjunct, effectively making its type any() in this context. To find the solution for the whole disjunction we take the least upper bound (or supremum) of the types from each solution and find that

\[
\tau_{\text{out}} \subseteq \sup(\text{true}, \text{false}) = \text{bool}() \\
\tau_x \subseteq \sup(\text{any}, \text{any}) = \text{any}()
\]

Therefore, the derived type signature for the function is:

\[
is\_this\_the\_answer\_1 : (\text{any}()) \rightarrow \text{bool}()
\]
4.4.4 Unsatisﬁable constraints

If two constraints in a conjunction are found to be contradictory, the whole conjunction is unsatisﬁable (i.e., it has no solution) and all types are considered to be the bottom type none(). If the constraints represent a clause in the program, we have found that this clause can never return a value, which of course means that this clause cannot inﬂuence its intended type signature. In other words, only clauses that succeed determine the type annotation derived for a function. In the extreme case where all clauses of a function yield unsatisﬁable constraints, this means that the function cannot return at all.

Consider the function in Figure 4(b). The constraints are:

\[
\begin{align*}
\tau_{fib} & \subseteq \text{atom}() \land \left( (\tau_x \subseteq \text{atom}() \land \tau_x \subseteq \text{integer}) \lor (\tau_{out} \subseteq \text{true}) \right) \\
& \lor (\tau_{out} \subseteq \text{false})
\end{align*}
\]

On the one disjunct, we have the contradiction:

\[
\tau_x \subseteq \text{atom}() \land \tau_x \subseteq 42
\]

which means that the type signature that is derived for this function is obtained from the other disjunct and is:

\[
is_this_the_answer_2 :: (\text{atom}()) \rightarrow \text{false}
\]

4.4.5 Recursive constraints

Recursive constraints are solved by iteration where the partial results are inserted in the constraints until a ﬁx-point is reached. In Figure 5(a) an ERLANG implementation of the Fibonacci function is shown. For this function, the corresponding type constraints are shown in Figure 5(b).

When solving these constraints, in the ﬁrst iteration we ﬁnd that \(\tau_{fib}\) has closed forms in the ﬁrst two disjunctive constraints:

\[
\begin{align*}
\tau_{fib} = (0) & \rightarrow 1 \\
\lor \tau_{fib} = (1) & \rightarrow 1
\end{align*}
\]

\[
\Rightarrow \tau_{fib} = (0 \mid 1) \rightarrow 1
\]

Note that intuitively this partial result describes the leaf cases that at some point must hold in order for the function to return. The third disjunction has no solution in the ﬁrst iteration since we do not have a type signature for \(\tau_{fib}\) yet. In the second iteration the constraints become:

\[
\begin{align*}
\tau_{x-1} & \subseteq (0 \mid 1) \\
\land \tau_{x-2} & \subseteq (0 \mid 1) \\
\land \tau_{x} & \subseteq (1 \mid 2 \mid 3) \subseteq \text{integer}() \\
\land \tau_{fib} = (\text{integer}()) & \rightarrow 2
\end{align*}
\]

Figure 4. Functions yielding disjunctive constraints.

\begin{center}
\begin{tabular}{ll}
\text{fib}(0) & \rightarrow 1; \\
\text{fib}(1) & \rightarrow 1; \\
\text{fib}(X) & \rightarrow \text{fib}(X - 1) + \text{fib}(X - 2). \\
\end{tabular}
\end{center}

\begin{center}(a) ERLANG function.
\end{center}

\begin{center}
\begin{align*}
\tau_{fib} &= (0) \rightarrow 1 \\
\lor \tau_{fib} &= (1) \rightarrow 1 \\
\lor \tau_{fib} &= (\tau_x \rightarrow \tau_{fib}(\tau_{x-1}) + \tau_{fib}(\tau_{x-2}))
\end{align*}
\end{center}

\begin{center}(b) Generated constraints
\end{center}

Figure 5. Vanilla Fibonacci numbers.

\begin{center}
\begin{tabular}{ll}
\text{fib}_2(0) & \rightarrow 1; \\
\text{fib}_2(1) & \rightarrow 1; \\
\text{fib}_2(X) & \rightarrow \text{fib}_2(\text{trunc}(X - 1)) + \text{fib}_2(\text{trunc}(X - 2)). \\
\end{tabular}
\end{center}

\begin{center}(a) All case clauses are valid.
\end{center}

\begin{center}(b) Function where one case clause cannot match.
\end{center}

\begin{center}
\begin{align*}
fib_2(Zero) & \rightarrow 1; \\
fib_2(One) & \rightarrow 1; \\
fib_2(X) & \rightarrow \\
\end{align*}
\end{center}

\begin{center}(a) ERLANG function.
\end{center}

\begin{center}
\begin{align*}
\tau_{fib} = (0) & \rightarrow 1 \\
\land \tau_{fib} = (1) & \rightarrow 1 \\
\land \tau_{fib} = (\text{integer}()) & \rightarrow 2
\end{align*}
\end{center}

\begin{center}(b) Generated constraints
\end{center}

Figure 6. Fibonacci function with a twist.

Note that we have widened the union \((1 \mid 2 \mid 3)\) to \text{integer}() for brevity.\footnote{In the actual analysis, the \texttt{union limit}, determining when to apply widening, is bigger, and also the iteration would pass through the types byte() and char().}

The current type for \text{fib} now becomes:

\[
\tau_{fib} = (\text{integer}()) \rightarrow (1 \mid 2)
\]

Inserting this once again yields:

\[
\begin{align*}
\tau_{x-1} & \subseteq \text{integer}() \\
\land \tau_{x-2} & \subseteq \text{integer}() \\
\land \tau_{x} & \subseteq \text{integer}() \\
\land \tau_{fib} = (\text{integer}()) \rightarrow (2 \mid 3 \mid 4) \\
\subseteq (\text{integer}()) \rightarrow \text{integer}()
\end{align*}
\]

and we arrive at a fix-point. The inferred type signature is:

\[
\text{fib}_2/1 :: (\text{integer}()) \rightarrow \text{integer}()
\]

This relatively heavyweight approach to inferring that the input argument of the Fibonacci function is of type \text{integer}() might seem a bit extreme to readers familiar with a vanilla Hindley-Milner type inﬁrmer, but its use is necessary in the presence of subtyping and in the context of a dynamically typed language such as ERLANG where the built-in arithmetic operators are overloaded. Principal typings for input arguments cannot be derived by making assumptions which are valid in a Hindley-Milner type system but not valid in ERLANG. To see why, consider the somewhat unorthodox version of the Fibonacci function shown in Figure 6. (The \text{trunc}/1 \footnote{\text{trunc}/1 is equivalent to what some other program languages like C call floor().} function\footnote{\text{trunc}/1 is equivalent to what some other program languages like C call floor().})

Clearly, the Fibonacci-with-a-twist can be called either with a non-negative integer or float and still return an answer. Writing and solving the type constraints for this function is left as an exercise for the reader, but the inferred type signature is:

\[
\text{fib}_2/2 :: (\text{number}()) \rightarrow \text{integer}()
\]

Note that this function is indeed allowed in ERLANG but it would be rejected by the type system of e.g., Standard ML.

4.5 Beneﬁting from the module system

It is common that ERLANG functions are organized in modules with a speciﬁed interface, declared with an \texttt{-export()} statement. Non-exported, called \texttt{internal} or \texttt{module-local}, functions are protected against arbitrary calls from other modules. This means that for module-local functions we can employ a closed world assumption about their intended uses and exploit information about their calling contexts to derive better typings for them.
The type inference which TYPRE employs is modular. Functions are analyzed by taking their (mutual) dependencies into account. Once a function has been analyzed, the resulting type signature is put into a lookup table. Whenever a call to this function is encountered, we can use the current value in the lookup table. By ordering the analysis based on a reverse topological sort of the strongly connected components (SCCs) of the function call graph, we can ensure that all information about the functions that a SCC depends on already has been gathered.

But, what if the called function is not available? In the requirements of Section 3.1 we mentioned that TYPRE must do something reasonable if not all the code is available. The solution is both simple and safe: a call to an unknown function does not constrain any of the arguments nor the return type. This would be problematic if we were trying to type check the program, but since we are merely trying to gather all available type information, we can fall back on the type signature \((\text{any}() \ldots, \text{any}()) \rightarrow \text{any}()\) for any unknown functions.

If previously missing information about functions becomes available, e.g., by including more modules in the analysis, we will possibly get new constraints for the call site. Since this can only make the information more precise, it is easy to see that the previously derived type must be an over-approximation, which is of course sub-optimal but safe.

Erlang comes with an extensively used standard library. When TYPRE is used for the first time, the current version of the stdlib is analyzed and the information is put into a persistent lookup table that is thereafter used as the starting point of subsequent analyses.

**4.7 Putting it all together**

The basic type annotation algorithm that TYPRE employs is as follows:

1. Construct the call graph for the functions and sort it topologically based on the dependencies between its strongly connected components (SCCs).
2. Analyze the SCCs in a bottom-up fashion using the constraint-based type inference to find their most general success typings under the current constraints.
3. Analyze the SCCs in a top-down order using the data-flow analysis to propagate information from the call sites to module-local functions.
4. Add new constraints for the type inference, based on the propagated information from step 3.
5. If a fix-point has been reached, annotate the program with the derived type signatures, otherwise repeat from step 2.

**4.8 A slightly bigger example**

Consider the module list util shown in Figure 9(a), containing some well-known functions for list manipulation. We will describe how we infer type annotations for functions of this module.

In Figure 9(b) the function call graph for this module is shown. Apparently, each strongly connected component consists of a single function, and we also see that function reverse/2 has to be analyzed first since it does not depend on any other functions but itself. Performing the type inference we get the following success typings (in the order with which the functions are analyzed):

\[
\begin{align*}
\text{reverse/2} & : (\text{list}(), \text{any}()) \rightarrow \text{any}() \\
\text{reverse/1} & : (\text{list}()) \rightarrow \text{any}() \\
\text{map/3} & : (\text{any}(), \text{list}(), \text{list}()) \rightarrow \text{any}() \\
\text{map/2} & : (\text{any}(), \text{list}()) \rightarrow \text{any}()
\end{align*}
\]
These signatures are correct, but they are arguably not showing the intention of the programmer. By applying the data-flow analysis for module-local functions outlined in Section 4.5, we find that the second argument in calls to \texttt{reverse/2} can only be list(), and that yields the typings shown below:

\begin{align*}
\texttt{reverse/2} & : (\text{list()}, \text{list()}) \rightarrow \text{list()} \\
\texttt{reverse/1} & : (\text{list()}) \rightarrow \text{list()} \\
\texttt{map/3} & : (\text{any()}, \text{list()}, \text{list()}) \rightarrow \text{list()} \\
\texttt{map/2} & : (\text{any()}, \text{list()}) \rightarrow \text{list()}
\end{align*}

For this example, this is as far as we can get. Although intuition tells us that \texttt{map/2} should always be called with a higher order function in its first argument, this is not reflected in the type signature of the function. However, if we closely look at its code, we can convince ourselves that a call in another module

\begin{align*}
\text{L} = \text{list.util:map}(3.14, [])
\end{align*}

would happily return the empty list. In the presence of arbitrary subtyping, this is not a type error.

### 4.9 Using type inference for documenting function interfaces

Such over-approximations in the interfaces of functions like that of \texttt{map/2} often signal an alarm. When the authors of the module examine the automatically derived type signature, they might consider changing \texttt{map/2} to the function in Figure 9(c), using the new \texttt{is_function/2} guard BIF which is available starting with Erlang/OTP R11. The programmer thus can, with minimal cost, thereby ensure that the first argument is indeed a higher order function of arity 1. Doing so, would change the inferred type signatures to:

\begin{align*}
\texttt{map/3} & : (\text{any()}) \rightarrow \text{any()}, \text{list()}, \text{list()}) \rightarrow \text{list()} \\
\texttt{map/2} & : (\text{any()}) \rightarrow \text{any()}, \text{list()}) \rightarrow \text{list()}
\end{align*}

This change would of course make the \texttt{map/2} call with 3.14 as the first argument now fail, but this is arguably either an erroneous call or possibly an extremely obfuscated way of checking that the second argument of \texttt{map/2} is the empty list.

In other words, the annotations produced by \textsc{TypTer} not only can be used for automatic documentation and program understanding, but also signal places where the code does not necessarily reflect the programmers’ intentions. The following example\footnote{Posting on the \texttt{erlang-questions} mailing list by Ulf Wiger.} shows how reasoning about automatically derived, slightly counter-intuitive, type annotations for commonly used functions can often reveal subtle errors and unintended behavior of functions which have survived over a long period of time.

In \texttt{stdlib}, the implementation of \texttt{lists:suffix/2} is as follows:

\begin{align*}
\texttt{suffix(Suffix, Suffix)} & \rightarrow \text{true} \\
\texttt{suffix(Suffix, [\_Tail])} & \rightarrow \\
\texttt{suffix(Suffix, Tail)} & \\
\texttt{suffix(_, [])} & \rightarrow \text{false}
\end{align*}

for which \textsc{TypTer} derives the following type signature:

\begin{align*}
\texttt{suffix/2} & : \text{(any(), any())} \rightarrow \text{bool()}
\end{align*}

while its documented interface, describing its intended use, is:

\begin{align*}
\texttt{suffix/2} & : \text{(list(), list())} \rightarrow \text{bool()}
\end{align*}

When faced with such a discrepancy between the documented and inferred type signature of a such a commonly used and relatively intuitive function, one is easily tempted to assign blame to the over-approximation that a type inferencer is usually forced to employ. In this simple case, the culprit is clear: the first clause does not specify any type constraints for its arguments, so both arguments have \texttt{any()} as their derived type. On the other hand, at least on the surface, the programmer’s intention is that the second argument is of a \texttt{list()} type. The problem is that the interface of \texttt{lists:suffix/2} is under-specified and allows for calls of e.g. the form:

\begin{align*}
\texttt{lists:suffix(a, [1,2,3|a])}
\end{align*}

to happily return ‘true’, which arguably is not part of the intended behavior of the function. In this case, the problem can be fixed by simply changing the first clause to either be:

\begin{align*}
\texttt{suffix(Suffix, Suffix)} \text{ when is_list(Suffix)} & \rightarrow \text{true} \\
\end{align*}

which allows the derivation of the following type signature:\footnote{The guard \texttt{is_list/1} ensures that the argument is either nil or a cons-cell. It does not traverse lists to ensure nil-termination.}

\begin{align*}
\texttt{suffix/2} & : \text{(possibly_improper_list(), possibly_improper_list())} \rightarrow \text{bool()}
\end{align*}

or be as follows:

\begin{align*}
\texttt{suffix(Suff, Suffix)} \text{ when length(Suff) >= 0} & \rightarrow \text{true} \\
\end{align*}

which then derives the intended type signature:

\begin{align*}
\texttt{suffix/2} & : \text{(list(), list())} \rightarrow \text{bool()}
\end{align*}

An arguably better fix is to actually rewrite the \texttt{lists:suffix/2} code using \texttt{lists:reverse/2} and \texttt{lists:prefix/2}.

### 5. Consequences of Inferring Success Typings

As discussed in Section 4.3 we define success typings as describing the types of arguments for which a function will ever possibly return a value (having an associated return type). This view stems from the usual definition of a function in mathematics: a function

\begin{align*}
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\end{align*}

denotes a function \texttt{ys} which assigns to each \texttt{X} an \texttt{Y}. This definition is easily extended to cover functions which produce multiple values (\texttt{map/2}, \texttt{fold/2}, \texttt{foldl/2}, etc.)

\begin{align*}
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}}
\end{align*}

where \texttt{ys} denotes a function \texttt{ys} which assigns to each \texttt{X} an \texttt{Y}. This definition is easily extended to cover functions which produce multiple values (\texttt{map/2}, \texttt{fold/2}, \texttt{foldl/2}, etc.)

\begin{align*}
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}} \\
\text{\texttt{ys}} & : \text{\texttt{X}} \rightarrow \text{\texttt{Y}}
\end{align*}
accepts some parameters as input and returns a result. However, in the context of a programming language with side-effects like ERLANG, this has some consequences.

5.1 Handling exceptions

Consider the function in Figure 10(a). The first clause contains an explicit guard test to find out if the argument is an atom, and then prints an error message and exits without returning to its caller. Thus, this clause does not contribute to the success typing for the function and the inferred type signature for this function is:

\[
\text{foo1/1 :: (} \text{number()} \text{) → number()}\]

Note that the programmer’s intention to do something with atoms, namely print an explicit error message, is not reflected in the inferred type signature. In other words, success typings do not take side effects into account and treat input argument types in non-returning functions similarly to any other type that the function does not explicitly handle (e.g., to \(X\) being a list in this example).

A similar situation happens when instead of an explicit \texttt{exit}, a function clause does not return due to a type error. If the function \texttt{foo2/1} in Figure 10(b) is called with an atom, it will definitely raise a runtime exception. The success typing once again disregards the failing clause, and the inferred type signature is:

\[
\text{foo2/1 :: (} \text{number()} \text{) → number()}\]

5.2 Non-returning functions and servers

A related problem is that concurrent ERLANG applications often use the concept of servers. These are possibly non-returning functions that respond to received messages (effectively via a side-effect), and then perform a tail call to themselves, effectively implementing an infinite loop that cannot return. In effect, these functions must be annotated as having no return, simply because they have none. But then the problem arises: What is the difference, expressed in success typings, between a function call that does not return and a function call that fails?

An example of a simple server can be found in Figure 11(a). Note that the input argument has to be a pid, but since our type inference cannot differentiate between failing and non-returning clauses, the success typing for this function is:

\[
\text{loop1/1 :: (} \text{none}() \text{) → } \text{ok}\]

meaning that the function will not return for any type of input argument, which is of course correct.

One solution to this problem is to modify the server as shown in Figure 11(b). Now the server has a returning clause and its inferred success typing is:

\[
\text{loop2/1 :: (} \text{pid}() \text{) → ok}\]

Since servers are common in ERLANG and having to rewrite the server code violates the goal from Section 3.1 that TypER should be able to take any ERLANG program, we are currently investigating possible ways to distinguish between failing and non-returning function clauses.

6. A Taste of Performance

As mentioned in Section 3.1, one of the desired properties for TypER is that its analysis should be fast and scalable. Table 2 shows times for deriving type annotations for various applications of ERLANG/OTP R10B-5 when running on an 2GHz AMD64-based machine with 1GB memory running Linux. As can be seen, the type inference, although not blindly fast, is perfectly usable.

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>asnl</td>
<td>40,045</td>
<td>10m45s</td>
</tr>
<tr>
<td>gs</td>
<td>17,371</td>
<td>57s</td>
</tr>
<tr>
<td>inets</td>
<td>22,344</td>
<td>1m18s</td>
</tr>
<tr>
<td>kernel</td>
<td>39,031</td>
<td>1m53s</td>
</tr>
<tr>
<td>mnesia</td>
<td>24,042</td>
<td>1m59s</td>
</tr>
<tr>
<td>hipe</td>
<td>90,526</td>
<td>6m35s</td>
</tr>
<tr>
<td>snmp</td>
<td>42,317</td>
<td>1m43s</td>
</tr>
<tr>
<td>asrl</td>
<td>9,276</td>
<td>22s</td>
</tr>
<tr>
<td>ssl</td>
<td>19,703</td>
<td>34s</td>
</tr>
<tr>
<td>stdlib</td>
<td>56,148</td>
<td>6m59s</td>
</tr>
<tr>
<td>tools</td>
<td>14,531</td>
<td>3m42s</td>
</tr>
</tbody>
</table>

Table 2. Times for annotating some ERLANG/OTP applications.

7. Related Work

Till now, there have been various attempts to make the type awareness greater in the ERLANG community. Two formally documented type systems have been developed: one based on subtyping by Marlow and Wadler [6] and one based on soft typing by Nyström [7]. So far, neither has been widely used in ERLANG development, partly due to those efforts never becoming mature tools or integrated in the ERLANG/OTP environment. Besides this reason, in the former case, we believe that the reasons include that the type system tries to impose a certain style of programming to ERLANG, closer to the one in statically typed languages, for example by demanding explicit handling of failing cases and manually adding type annotations. It would certainly require a significant amount of effort or
legacy code to adopt and a fair amount of discipline for newer code to adhere to this style of programming. In contrast, TyPER is completely automatic and does not impose any change to existing programs or coding practices to be useful, but instead only implicitly encourages a more type-friendly development of ERLANG code.

An existing tool for annotating Erlang code with documentation information is edoc. It produces API documentation from manual type annotations, taking ideas from javadoc as initial inspiration, adopting them to the context of ERLANG, and often extending them. In contrast to TyPER, edoc does not try to infer types and it provides no mechanisms to verify that the types in the documentation correspond to types used in the code. As a result, there is an obvious risk that the code evolves and the documentation does not, rendering the documentation obsolete, leaving manual inspection as the only tool to find this out.

In the context of Scheme, which is also a dynamically typed functional language, type inference has been explored in the DrScheme/MrSpidey programming environments where the programmer has possibility to interactively inspect the types at specific program points. In [3] the authors also describe a way of dealing with imprecision in the analysis that is similar to our suggestions in Section 4.9 for tuning the code for better analysis precision.

In both dynamically and statically typed languages there has been a lot of work on automatic type inference. The work that is based on unification is of little relation to ours, since unification-based type inference is typically used in Hindley-Milner type systems without subtyping. The work using constraint-based type inference (see [1] for an early such work) is more closely related. However, the work which most closely relates is that on inferring principal typings by Jim [4] (but see also [8]). Like success typings, principal typings allow for compositional type inference. More specifically, a principal typing is not only a principal type but also an associated environment. This resembles our success typings in that we say that the annotated type signature only holds if the arguments in an application are subtypes of the arguments in the signature. In some sense an environment is exported from the function to the call site. If the types in the exported environment contradict the types at the call site, the call will fail, but the typing of the called function still holds.

8. Concluding Remarks

Sometimes the arguments between advocates of static typing and those of dynamic typing verge on religious wars. One side claims the virtue of catching type errors at compile time, the other claims freedom of expression and flexibility. One side claims careful type-based design of interfaces, the other claims rapid prototyping. Luckily, in this paper we do not have to choose side in this discussion. We can simply state that the creator made ERLANG dynamically typed, and this is not likely to drastically change in the near future. On the other hand, we hold that creating software tools like TyPER which raise the level of type awareness in the ERLANG community is probably a good idea, especially if such tools can help developers to maintain their code more easily or to understand code written by somebody else, without sacrificing any of the characteristics and flexibility of the language.

Acknowledgments

This research has been supported in part by VINNOVA through the ASTEC (Advanced Software Technology) competence center as part of a project in cooperation with Ericsson and T-Mobile.

References


A. More information on TyPER

In Figure 12 the command line options of TyPER are listed. Note that, where applicable, options are analogous to the options that the ERLANG compiler command erlc takes.