STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

TYPE INFERENCE FOR CORE ERLANG TO SUPPORT TEST DATA GENERATION

GÁBOR OLÁH, DÁNIEL HORPÁCSI, TAMÁS KOZSIK, AND MELINDA TÓTH

Abstract. Success typing is a well known technique to calculate the type of Erlang functions. Although success typing is commonly used for documentation and discrepancy analysis purposes, it results in an over-approximation of the real type. Therefore, when we want to generate arguments for a function call based on the success typing of a function, the function call may fail during execution. In this paper we introduce a new algorithm to calculate the type of Erlang functions to support accurate data generation.

1. Introduction

Erlang [3] is a dynamically typed, functional, concurrent programming language. *Core Erlang* [5] is a pure functional variant of the base language, where any construct of Erlang can be easily expressed in Core Erlang, whilst preserving most of the static semantic properties, like types.

In Erlang, types of variables and functions are not defined in the program. Although the compiler performs some strict type checks (i.e. no implicit type conversions can happen), if a function (or an operator) is invoked with an unexpected type of data, only a run-time exception is thrown. Such programming errors are a tedious task to reveal, therefore several tools have been made to help programmers find possible discrepancies in the program, i.e. situations where type mismatch can happen. However, the language allows polymorphic return types for branching expressions, which makes the type system very complex, and the types uneasy to comprehend. In order to overcome this issue,

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *Semantics*.

Key words and phrases. typing, Erlang, type-based data generation.

This work has been supported by the European Union Framework 7 under contract no. 288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems", `http://paraphrase-enlarged.elte.hu`.

success typing [8] has been introduced, reducing the aforementioned complexity by substituting an upper bound type for complex union types and making types readable; on the other hand, it loses static type information.

Success typing became the de-facto type inference algorithm for Erlang. It yields an over-approximation of types, which increases readability, but decreases accuracy. We cannot use success types, for example, for test data generation, since they are too general, so that the data we get based on success typing will likely be improperly typed. Our goal is to make a type inference system for Erlang that derives types accurate enough for test data generation. In particular, we transform these types to *QuickCheck* [4] data generators that can supply functions with random arguments, which we utilize to build an Erlang benchmarking system.

By introducing a new, more precise type system to Core Erlang, the information loss can be decreased for the price of long type expressions. As stated already, Erlang programs can be easily turned into semantically equivalent Core Erlang programs, where the functions preserve their types. The long, but more accurate types are more suitable for random argument generation. We define type inference rules and the full algorithm is described. A comparison of our achievements with earlier results is provided, as well as a proposal to extend the results for the full Erlang language.

1.1. **Erlang.** Erlang was developed by Ericsson to program concurrent, distributed telecom equipments. The main design principles of the language were easy prototyping, declarative and fault tolerant. Easy prototyping implied a dynamically typed language, i.e. all type checks happen at run-time. Fault tolerance, on the other hand, required that no hidden errors are introduced by type conversion, thus the type checks are strict. To keep the language small and easy to learn, there are no user defined types in the language besides the built-in ones.

Although providing a statically typed language would have seemed a must for fault tolerant, high availability telecom systems, the restrictions it would have implied were thought to be unnecessary. That is the reason, why the type system for Erlang is not first-class citizen of the language. This fact made creating a comprehensive and useful type system a challenging task. But the need for type checking is existing among Erlang programmers, so external tools have been created to fulfill this requirement.

Although Erlang is considered a functional language and it, indeed, is mostly functional, many of the type inference systems (e.g. Hindley-Milner) cannot be applied to Erlang. There are certain aspects that make typing Erlang programs a challenging task. First, its syntax is complex, compared to

other easily typable languages like ML. To overcome this problem a semantically equivalent, pure functional variant of the base language, *Core Erlang* [5] has been introduced. Every language construct of the base language can be easily expressed in Core Erlang. Type properties are preserved during the mapping, so in this paper we will deal with only Core Erlang programs.

Why is it difficult to type Erlang programs? Although there are only built-in types, and they are small in number, there are three major problems during typing.

Firstly, there can be calls to functions that are not written in Erlang.

(1) They can be implemented in the virtual machine. Built-in functions do not have an Erlang definition, they are implemented in C. This makes typing these programs with a single type system impossible. The usual solution is to hard-code the types of built-in functions to the type system.
(2) They can be implemented in other languages, and "attached" to the virtual machines with one of the possible ways, like port drives, or natively implemented functions. This latter might have a reference implementation in Erlang, but it is usually not the case.

Secondly, the body of the function may not available. The modules can be shipped without the source in a byte-compiled version. We cannot assume any type information on these functions, which makes typing uncertain.

Thirdly, Erlang allows polymorphic return types for branching expressions and for functions with more clauses. Strict type checking makes the return type unambiguous at runtime. But making static type analysis be aware of run-time properties of a program is a challenging task.

1.2. **Related work.** The operational semantics of Erlang does not allow constructor-based type inference, like Hindley-Milner, to be used. Fortunately, subtyping systems are more suitable to type Erlang programs. Adopting a subtyping system to Erlang was introduced in [9] by Marlow and Wadler. This work was based on [1] by Aiken and Wimmers. Their type system was mostly successful on real Erlang programs (on a portion of the Erlang standard library, the Open Telecom Platform), but it did not manage to type the "match all" (underscore) pattern properly.

There is another approach to type Erlang programs, to insert type annotations to programs. A soft-typing system was proposed by Nyström in [10]. It used data-flow analysis and was based on supplied type annotations. In industrial environment, modifying existing source code is usually not viable, so type systems requiring annotations are not adopted widely.

An interesting soft-type system was introduced by Aiken, Wimmers and Lakshman in [2]. They added conditional types to the soft-type system reflecting the data-flow information in the types. Very accurate types can be generated with conditional types, and theoretically only unreachable codes can add uncertainty to the type. But the size of type expressions is directly proportional to the code length, which makes the generated types unreadable to humans.

A very successful soft-typing system was developed by Lindahl and Sagonas [8], called *success typing*. It accumulates the pros and cons of previous systems, and tries to solve most of the problems. Its main objective is to generate types that are useful for programmers without any type annotations, while detecting as many type errors as possible. A success type guarantees that if it is not met, than there is a type error. Although it may seem too general, this is a scalable system to Erlang. A success type remains compact enough for human readability while accurate enough to ensure useful type checking. It was implemented in a standalone tool, TypEr [7], which is now included in the Open Telecom Platform.

2. Motivation

The underlying goal of our project [11] is to discover and rank sequential code fragments that are amenable to be transformed into instances of parallel algorithmic patterns. Typically, there are dozens of parallelizable expressions in a program, but most of them cannot offer a real improvement in run-time speed due to the communication overhead. Our method is intended at identifying and transforming the best candidates resulting in the best possible performance gain; to this end, we need to either check or forecast the parallel performance. Ranking of candidates is based on execution as well as on prediction: we benchmark some sequential code components and then estimate the parallel execution time by applying pattern-specific cost models.

Benchmarking, in our case, consists of multiple executions and timing of the sequential code fragment, which generates statistics and leads to some hints considering the time complexity of the operation to be parallelized. In order to be able to run the small program fragment picked out of the code, we encapsulate it into a self-containing Erlang module: after identifying its free variables, we turn it into a function abstraction, and then couple it together with all its dependencies. Since the execution time likely depends on the concrete arguments, we feed the synthesised function with a large number of randomly generated arguments in order to get a fairly accurate estimation of the average sequential execution time; this is the point where types play an important role.

Randomly generated arguments (or in other words, test data) are randomly chosen elements of the type of the function arguments. If we use *success typing* (the de-facto type inference algorithm for Erlang) to find out the argument types, we obtain an over-approximation of the actual type, which is too general: the function surely fails if we call it with a value outside this type, but it may also fail if we invoke it with values present in the type. Obviously, values of the latter kind are likely not supported by the function and lead to various run-time errors. For test data generation, we need more accurate types, for elements of which the function is certainly implemented, otherwise the timing statistics will be distorted. The accuracy obtained by the refined type inference algorithm presented in this paper is exploited in finding the type that we can use to generate proper function arguments.

2.1. **Examples.** To demonstrate the problem consider function `encode` in Fig. 1. This example is to demonstrate a key feature of success typing, i.e. using upper bound type if a type becomes too complex. In this case the type of the function argument would be the union of the five atoms. Success typing changes this type to the type `atom()`, which means *any kind of atom.*[1] Our aim is to never use upper bound types for union to enable accurate data generation.

```
encode(Label) ->
    case Label of
        function -> 0;
        module   -> 1;
        variable -> 2;
        expr     -> 3;
        value    -> 4
    end.
```

FIGURE 1. Example function for union type.

Another example where success typing uses upper bound types is to reach fixed point types in case of recursive types. A classic example of recursive types are trees; they can be expressed by tuples in Erlang. Consider the function `tree_to_list` in Fig. 2. Without collapsing the tuple to `any()` in success typing, the algorithm would never terminate, because a fixed point

[1]The theory does not specify the maximum number of elements in a union and the implementation in TypEr uses a threshold 5. For sake of simplicity, in our examples consider this threshold to be 4 for success typing.

would never be reached. Our aim is to give a *depth-k* representation type for recursive types.

```
tree_to_list(nil) ->
    [];
tree_to_list({Left, Data, Right}) ->
    tree_to_list(Left) ++ [Data|tree_to_list(Right)].
```

FIGURE 2. Example function for recursive types.

Although our types are not the exact types of the function, they contain less ambiguity about the possible usable types. Unfortunately, by keeping the union types and not including conditional types, our type system will not be able to provide more accurate types for functions where the intended use cannot be calculated by only examining the body of the functions. As an example, consider the function implementing the logical conjunction in Fig. 3. The problem here is that the *match all* pattern (_) is used for both function parameters. This can only be improved by using conditional types. We will not use them, to be compatible with QuickCheck.

```
and(true,  true)   -> true;
and(false, _)      -> false;
and(_ ,    false)  -> false.
```

FIGURE 3. Example function where no improvement is possible.

3. PRELIMINARIES

In this section we introduce the most important aspects of the Erlang type system. Later we summarize the most important Core Erlang structures.

3.1. **Erlang types.** Erlang has only built-in data types. A piece of data is usually referred to as a *term*. Here we list the possible data types our type system uses. Erlang and Core Erlang support the same data types, this makes type inference in Core Erlang be also valid in Erlang.

The usual notation for Erlang types is the type name and a pair of parentheses, like `list()`. In a compound type, the component types can be listed as parameters in the type expression, e.g. the tuple of integers and atoms is written as `tuple(integer(), atom())`. An even simpler – and more convenient – notation is to use the tuple data structure syntax in the type language, such as `{integer(), atom()}`.

Numbers. Erlang numbers can be either integers without upper or lower bound, or floating point numbers. There is a limited automatic conversion between them, usually the mathematical operations accept both representations.

Atoms. Erlang has a special data type for named constants. Atoms are massively used in Erlang programs. Two atoms are equal if their string representations are equal.

There are two atoms that have special meaning in some contexts, namely `true` and `false`. They represent the boolean values of logical operators. Still, they are ordinary atoms, since formally there exists no boolean type in Erlang.

Funs. Functions are first-class objects that can be created by other functions and passed as arguments. The number of arguments of a function is predetermined during its creation and it cannot be modified. Functions can have more clauses. Neither the argument types nor the return types of the different clauses have to be the same.

Tuples. One of the compound data types is the tuple. A tuple can contain a fixed number of terms from any type. Tuples are the programming equivalent of Cartesian products.

In Erlang, the *record* construct is a syntactic sugar for tuples, where the first element has to be an atom representing the name of the record. Core Erlang does not support records, but uses the tuple representation. Therefore, our type system does not contain records.

Lists. The other compound data type in Erlang is the list. They can hold any number of terms; each term can be of any type. In Erlang, lists can be constructed in different ways (e.g. by listing the elements, using lists generators etc.), but in Core Erlang we only use the *cons* representation: a list is build from a head element and a tail. Strangely, the tail need not be a list – this case is referred to as *improper list*.

Strings are also syntactic sugar in Erlang, since they are lists of integers.

Identifier types. There are three types of identifiers in Erlang. Each identifies a resource in an Erlang system. *Pid*s (process identifiers) are used to refer to processes. *Port*s refer to port drivers, i.e. an embedded resource used to communicate to other OS processes. *Ref*s are used as a unique reference point in an Erlang system.

These are usually created with built-in functions, and cannot be implicitly converted to anything.

Universal type. Our type system has a universal type, `any()`. It is a pseudotype, and is not presented by any terms. We use it to refer to types that cannot be determined due to lack of information.

Bottom type. On the other end of the type system, we use a bottom type, `none()` to represent type clashes.

Union types. Unions are used as a pseudo-types to represent the return type of branching expressions. If the type `any()` is added to the union, it becomes `any()`. Adding the bottom type `none()` to a union has no effect.

3.2. **Core Erlang.** Core Erlang is a pure functional variant of Erlang. The full Core Erlang reference can be found in [6]. In this paper we focus on the base constructs of Core Erlang, listed in Fig. 4, but the algorithm can be extended straightforwardly to the whole Core Erlang language. A Core Erlang program consists of one or more modules. In each module, a set of functions can be defined. The input for our type system will be the set of functions in one module, thus we can omit e.g. module-prefixed function names from the syntax.[2]

$$
\begin{array}{rcl}
e & ::= & x \mid c(e_1, \ldots, e_n) \mid e_1(e_2, \ldots, e_n) \mid f \mid \\
 & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e \mid \\
 & & \texttt{case}\ e\ \texttt{of}\ (p_1 \to b_1); \ldots; (p_n \to b_n)\ \texttt{end} \\
f & ::= & \texttt{fun}(x_1, \ldots, x_n) \to e \\
p & ::= & p'\ \texttt{when}\ g \\
p' & ::= & x \mid c(p'_1, \ldots, p'_n) \\
g & ::= & g_1\ \texttt{and}\ g_2 \mid g_1\ \texttt{or}\ g_2 \mid x_1 = x_2 \mid \texttt{true} \mid \texttt{is\_atom}(x) \mid \ldots
\end{array}
$$

FIGURE 4. Core Erlang expressions.

In Fig. 4, x denotes variables. The symbol c is a data constructor function, creating a term from its argument(s). If only one argument is given, it creates simple data types, while from multiple arguments it creates compound ones. In function application, e_0 must evaluate into a function. Note that in (Core) Erlang there is no partial application, thus all applications must be saturated. The number of arguments (n) can be zero – this is how atoms are modelled. The expression f represents fun-expressions (unnamed functions).

The `let` expression is a standard *let*, well-known from other functional languages. The `letrec` expression represents the named functions defined in a module, and hence can only appear as a top-level expression in a module. In this paper we do not type `letrec` expressions.

[2]The Erlang compiler is capable of transcoding Erlang sources to Core Erlang via an undocumented modifier `to_core`.

The `case` structure is the only branching expression in our simplified system. Other branching expressions, like `if` can be easily expressed with `case`. Every branch of a `case` contains a pattern (p'), and a guard (g). Patterns can only contain variables and constructors of other patterns. The guard part can contain logical operations, equality checks and a limited number of built-in functions.

4. Type derivation rules

In this section we give the type derivation rules for the expressions of Fig. 4. The general format of a typing judgement is the following: $\Gamma, \Phi \vdash e : \tau, \Gamma', \Phi'$, where Γ denotes a variable context containing mappings of variables to types, Φ denotes a function context containing mappings from functions to types. The function context is global. It contains types for all functions with names, i.e built-in functions and top level functions of the module. Both the variable and function contexts are sets. For brevity we use the notation of $\Gamma \cup f : \tau$ for adding element to contexts and indicating that the mapping is element of the context. Thus the judgement is read as "with Γ and Φ, the expression e is typed with τ resulting contexts Γ', Φ'". The derivation rules[3] can be found in Fig. 5.

If variable x is not in the variable context, then Rule (Var) extends the context, and returns the type $any()$. If the variable is already in the context, then we can use Rule (Var') and return the type from the context. We do not remove variables from the context.

To be able to type constructors (Rule (Constr)), first we have to type all the contained terms. For each term, we need to use the type context of the previous term.

In an application, e_1 can be either a call to a named function (Rule (Appl)), or an experssion evaluating to a fun expression (Rule (Appl')). In case of a call, the type of the function should be in the type context ($\tau_1 = (\psi_2, \ldots, \psi_n) \to \psi$). If e_1 is a fun expression, then we calculate the type by applying Rule (Fun). The return type of the application expression is the return value of the function applied. The variable context Γ is updated by the matching operator $\mathcal{M}$, which matches the variables in each e_i to the corresponding type from the arguments'.

To type a fun expression (Rule (Fun)) we type the component expressions reusing contexts of the preceding expressions. The returning type is a function type with the exact number of arguments. Γ_n contains the types for the

[3]The derivation rules are used only to type one function and supposes that all available type information for other functions is present in the function context.

$$\text{(Var)} \qquad \frac{}{\Gamma, \Phi \vdash x : any(), \Gamma \cup x : any(), \Phi} \qquad x \notin dom(\Gamma)$$

$$\text{(Var')} \qquad \frac{}{\Gamma = \Gamma' \cup x : \tau, \Phi \vdash x : \tau, \Gamma, \Phi} \qquad x \in dom(\Gamma)$$

$$\text{(Constr)} \qquad \frac{\Gamma_{i-1}, \Phi \vdash e_i : \tau_i, \Gamma_i, \Phi \ (\forall i \in [1..n])}{\Gamma_0, \Phi \vdash c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n), \Gamma_n, \Phi}$$

$$\text{(Appl)} \qquad \frac{\begin{array}{c}\Gamma_0, \Phi = \Phi' \cup e_1 : (\psi_2, \ldots, \psi_n) \to \psi \vdash e_1 : \tau_1, \Gamma_1, \Phi \\ \Gamma_{i-1}, \Phi \vdash e_i : \tau_i, \Gamma_i, \Phi \ (\forall i \in [2..n])\end{array}}{\begin{array}{c}\Gamma_0, \Phi \vdash e_1(e_2, \ldots, e_n) : \psi, \Gamma, \Phi, \text{ where} \\ \Gamma = \mathcal{M}(\{e_1, \ldots, e_n\}, \{\psi_1, \ldots, \psi_n\}, \Gamma_n)\end{array}}$$

$$\text{(Appl')} \qquad \frac{\begin{array}{c}\Gamma_0, \Phi \vdash e_1 : (\psi_2, \ldots, \psi_n) \to \psi, \Gamma_1, \Phi \\ \Gamma_{i-1}, \Phi \vdash e_i : \tau_i, \Gamma_i, \Phi \ (\forall i \in [2..n])\end{array}}{\begin{array}{c}\Gamma_0, \Phi \vdash e_1(e_2, \ldots, e_n) : \psi, \Gamma, \Phi, \text{ where} \\ \Gamma = \mathcal{M}(\{e_1, \ldots, e_n\}, \{\psi_1, \ldots, \psi_n\}, \Gamma_n)\end{array}}$$

$$\text{(Fun)} \qquad \frac{\begin{array}{c}\Gamma_{i-1}, \Phi \vdash x_i : \tau_i, \Gamma_i, \Phi \ (\forall i \in [1..n]) \\ \Gamma_n, \Phi \vdash e : \tau, \Gamma', \Phi, \text{ where } \Gamma \cup \{x_i : \phi_i \mid i \in [1..n]\} = \Gamma'\end{array}}{\Gamma_0, \Phi \vdash fun(x_1, \ldots, x_n) \to e : (\phi_1, \ldots, \phi_n) \to \tau, \Gamma, \Phi}$$

$$\text{(Let)} \qquad \frac{\Gamma, \Phi \vdash e_1 : \tau_1, \Gamma_1, \Phi \qquad \Gamma_1 \cup \chi^{\Gamma}_{\tau_1}(x : \tau_1), \Phi \cup \chi^{\Phi}_{\tau_1}(x : \tau_1) \vdash e_2 : \tau_2, \Gamma_2, \Phi'}{\Gamma, \Phi \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau_2, \Gamma_2, \Phi'}$$

$$\text{(Case)} \qquad \frac{\begin{array}{c}\Gamma, \Phi \vdash e : \tau, \Gamma_e, \Phi \\ (\Gamma_e, \Phi \vdash p_i : \tau_{p_i}, \Gamma_{p_i}, \Phi \qquad \Gamma_{p_i}, \Phi \vdash b_i : \tau_{b_i}, \Gamma_{b_i}, \Phi)(\forall i \in [1..n])\end{array}}{\begin{array}{c}\Gamma, \Phi \vdash \texttt{case}\ \ e\ \ \texttt{of}\ \ (p_1 \to b_1); \ldots; (p_n \to b_n)\ \ \texttt{end} : \bigcup_{i=1}^{n} \tau_{b_i}, \\ \Gamma' = \mathcal{M}(e, \bigcup_{i=1}^{n} \tau_{p_i}, \bigsqcup_{i=1}^{n} \Gamma_{b_i}), \Phi\end{array}}$$

FIGURE 5. Type derivation rules for Core Erlang.

variables of the arguments.[4] The returned variable context does not contain the variable mappings of the arguments, because funs open new scope for the argument variables.

Typing a let-expression (Rule (Let)) is not standard. To type e_2, we update either the variable or the function context. We define the operator χ as follows

[4]In Core Erlang, the arguments of funs are always variables.

$$\chi^{\Gamma}_{\tau}(x:\tau) = \begin{cases} x:\tau & \text{if } \tau \neq fun() \\ \emptyset & \text{otherwise,} \end{cases}$$

$$\chi^{\Phi}_{\tau}(x:\tau) = \begin{cases} x:\tau & \text{if } \tau = fun() \\ \emptyset & \text{otherwise.} \end{cases}$$

The χ is used to update the appropriate context, i.e. if τ is a function type ($fun()$), then we update the function context, otherwise the variable context is updated. Note that χ^{Γ} and χ^{Φ} are mutually exclusively return a type mapping. The returned type is the type of e_2 with its context.

$$\text{(Pat)} \qquad \frac{\Gamma, \Phi \vdash p' : \tau_{p'}, \Gamma_{p'}, \Phi \qquad \Gamma_{p'}, \Phi \vdash g : bool(), \Gamma_g, \Phi}{\Gamma, \Phi \vdash p = p' \text{ when } g : \tau_{p'}, \Gamma_g, \Phi}$$

$$\text{(GuardEq)} \qquad \frac{\Gamma, \Phi \vdash x_1 : \tau_1, \Gamma_1, \Phi \qquad \Gamma_1, \Phi \vdash x_2 : \tau_2, \Gamma_2, \Phi}{\Gamma, \Phi \vdash x_1 = x_2 : bool(), \mathcal{M}(x_2, \tau_1, \mathcal{M}(x_1, \tau_2, \Gamma_2)), \Phi}$$

$$\text{(GuardFun)} \qquad \frac{x \in dom(\Gamma)}{\Gamma, \Phi \vdash \texttt{is\_}\textit{type}(x) : bool(), \mathcal{M}(x, type(), \Gamma), \Phi}$$

FIGURE 6. Derivation rules for pattern matching.

To type a case-expression (Rule (Case)) first we type the head of the expression (e). For each branch is typed seperately. First the pattern is typed by applying the rules in Fig. 6. Each pattern consists of a pattern expression (p') and a guard (g). Typing pattern expressions is straightforward. The equation between variables (Rule (GuardEq)) returns boolean, but the type of the variables are matched respectively to the type of the other variable. In guards only a limited amount of built-in functions can be used. They express type contraints to a variable. The name of the function contains the type, which will be used to match the variable in the type context (Rule (GuardFun)). The return type of the pattern is the type of pattern expression ($\tau_{p'}$). The body of the case clause is typed using the variable context from the pattern. The return type of the whole case expression is union of types of the bodies. The return variable context is calculated by matching the head expression with the union of the types of the patterns in the element-wise union ($\sqcup$) of contexts from clause types (Γ_{b_i}).

Matching operator. In the application and case rules, we need to update the variables of an expression with the corresponding types. We define (Fig. 7) a partial function $\mathcal{M}$ that maps an expression, a type and a variable context

to a new variable context. If the expression does not contain any variables, then the context is returned. (The Var function returns the variables from an expression.) If the expression is itself a variable, then the context is updated with the infimum type of the previous type of the variable and the new type. If the type is a compound type (either a constructor or a fun type), then the components are recursively matched.

The matching function is partial, since if the structure of the expression and the type is not identical, then the function is not defined. There is no fail case, since the is calculated from the expression, hence the structure has to be the same. The function is well defined: its computation always terminates because it is called with a smaller expression in each recursive step.

(Match op.)

$$\mathcal{M} : Expression \times Type \times VarContext \to VarContext$$

$$\mathcal{M}(e,\tau,\Gamma) = \begin{cases} \Gamma & \text{if } Var(e) = \emptyset \\ \Gamma' \cup x : \inf(\tau,\tau') & \text{if } e = x \wedge \Gamma = \Gamma' \cup x : \tau' \\ \Gamma_n & \text{if } \begin{array}{l} e = c(e_1,\ldots,e_n) \wedge \tau = c(\tau_1,\ldots\tau_n) \wedge \\ \Gamma_i = \mathcal{M}(e_i,\tau_i,\Gamma_{i-1})\ (\forall i \in [1..n]) \end{array} \\ \Gamma_{n+1} & \text{if } \begin{array}{l} e = fun(e_1,\ldots,e_n) \to e_{n+1} \wedge \\ \tau = (\tau_1,\ldots,\tau_n) \to \tau_n \wedge \\ \Gamma_i = \mathcal{M}(e_i,\tau_i,\Gamma_{i-1})\ (\forall i \in [1..n+1]) \end{array} \end{cases}$$

FIGURE 7. Definition of the match operator $\mathcal{M}$.

5. The type inference algorithm

Our type system is not a general type system for Erlang programs. It is tailored to fit for the needs of test data generation from the calculated type. The input of our algorithm is a single module of Core Erlang terms. The algorithms follows the following steps.

(1) Take all function definitions from the module and populate the function context with general function types $((any(),\ldots,any()) \to any()$.
(2) Initialize a graph that will hold the function call graph of the module. Later it will be used to optimize the performance of the algorithm.
(3) Calculate the type of each fun expression of the module. In every application expression, populate the call graph with the information.
(4) Update the function context with the newly calculated values by overwriting the previous values.

(5) Calculate a directed acyclic graph from the call graph built during the first run. Serialize this graph from the least element. Use this serialization for the further steps to determine the order of the function to type.
(6) For a predefined number (in our implementation it is 15), recalculate the type of fun expressions by reusing the type context from the previous run, and updating it after each run. The order of the functions is defined in the previous point.

The result can be found in the type context for each function in the module. The type context is used to store the types of all built-in functions.

The algorithm is based on the syntax of Core Erlang. It evaluates a single module for a predefined number of times. The type inference rules do not contain recursion, hence the algorithm is always finite for any input.

Usually type inference algorithms (like Hindley-Milner and success typing) use a fixed point calculation as a termination condition. In the case of Hindley-Milner, reaching a fixed point in types corresponds to the finite behaviour of the program being inferred. In the case of success typing, the fixed point is reached by using an upped bound type if a type expression is too complex (e.g. unions or recursive types).

Our algorithm is finite and we do not want to reach the fixed point of types, because it would require applying an upper bound type like in success typing. Instead our types are always finite due to the lack of recursion.

5.1. **Evaluation.** The type inference algorithm presented in the previous section is way more conservative and therefore more suitable for our purpose; namely, it results in an under-estimation of the actual type of the function, for any element of which our function is definitely defined. For instance, in the example shown in Fig. 1, success typing infers the type `atom()` for the argument of the function, whereas for every atom except the five values present in the branching (`function`, `module`, `variable`, `expr` and `value`) it crashes with a run-time exception called `case_clause`. Apparently, there is a very little chance that random generation of arbitrary atom values picks one of these five words. However, the proposed type inference algorithm precisely derives the type with the five possible values (in this case, the result is not an under-estimation, but the actual type), and all the randomly generated values will be supported by the function. Similar consequences can be drawn for the example shown in Fig. 2, where the recursive type is over-generalised by the success typing algorithm, leading to completely random, most probably incompatible test data. On the other hand, with the more conservative algorithm the type becomes under-approximated and, although we won't be

able to cover all the values, we certainly generate data that the function is supportive for.

6. Future work

Our type system works only on a single module of Core Erlang code. An obvious extension can be to analyse multiple modules. Although it seems straightforward, the function call graph optimization makes it a bit challenging task.

Our generated types still lack precision in certain cases like in the one presented in Fig. 3. To solve that problem, our type system can be extended with conditional types. As mentioned earlier, our type system is compatible with the one of success typing in order to be able to use already existing random data generator tools. Introducing conditional types would require to implement/extend tools to generate test data.

Using our method was not yet tested on large-scale, industrial code-base. Doing so may reveal other extension possibilities.

The use of this type system can be useful for static code analyser tools, to provide more precise results in various areas e.g. data-flow analysis.

7. Conclusion

In this paper we have shown a new type inference algorithm for Core Erlang which is the pure functional variant of Erlang. Although there is a commonly used type system for Erlang, called success typing, it is not adequate for our purposes. The main goal of our work is to generate accurate test data for benchmarking sequential execution time of parallel pattern candidates.

During the benchmarking process random test data is generated based on the type of the function to measure. When the type is not accurate enough, we can generate test data that results in the function to fail during execution. Therefore we have developed a type inference system that never over-approximates the type of a function.

We have formalised the type derivation rules, and have shown an algorithm to calculate the types of the functions defined in an Erlang module. We have also presented some examples to demonstrate the strength of our algorithm.

References

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA, 1993. ACM.

[2] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL '94, pages 163–173, New York, NY, USA, 1994. ACM.

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

[4] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

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

[6] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0.3 language specification. `http://www.it.uu.se/research/group/hipe/corerl/doc/core_erlang-1.0.3.pdf`, 2004.

[7] Tobias Lindahl and Konstantinos Sagonas. TypEr: A Type Annotator of Erlang Code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, pages 17–25, New York, NY, USA, 2005. ACM.

[8] Tobias Lindahl and Konstantinos Sagonas. Practical Type Inference Based on Success Typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM.

[9] Simon Marlow and Philip Wadler. A Practical Subtyping System for Erlang. *SIGPLAN Not.*, 32(8):136–149, August 1997.

[10] Sven-Olof Nyström. A Soft-typing System for Erlang. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, ERLANG '03, pages 56–71, New York, NY, USA, 2003. ACM.

[11] ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. `http://www.paraphrase-ict.eu/`, 2011.

Eötvös Loránd University & ELTE-Soft Nonprofit Ltd
E-mail address: {`olikas, daniel-h, kto, tothmelinda`}`@elte.hu`