Master thesis project proposal

Static typing of Erlang programs using Partial Evaluation

Nachiappan Valliappan, nacval@student.chalmers.se

Supervisor: John Hughes

Relevant completed courses:

DAT280, Parallel Functional Programming
DAT350, Types for Proofs and Programs
TDA283, Compiler Construction
DAT151, Programming Language Technology

January 22, 2018

1 Introduction

Erlang is a concurrent functional programming language popular for its use in distributed applications. Being a dynamically typed language by design, it allows the successful compilation and execution of many programs which would typically be rejected by a type checker of a statically typed language. This idiosyncrasy of Erlang makes it difficult to retrofit existing type checking technology onto the language. In this proposal, we discuss some static typing ideas for Erlang, expand on the mentioned difficulties, and explain why *partial evaluation* could help. Based on these ideas, we propose to develop a static type system for a subset of Erlang programs.

1.1 Why Erlang?

Erlang is widely used in the industry for developing distributed and fault-tolerant applications. It has been used for a wide range of applications including telecom, social networking, and cloud services. Its simple approach to distributed systems has influenced the development of many distributed databases including Riak, CouchDB and Amazon SimpleDB. RabbitMQ, Ejabberd and WhatsApp are some other notable software applications written in Erlang.

Many traits of Erlang make it particularly well suited for writing distributed programs. It offers a functional *concurrency oriented programming* model based on message passing. It provides built-in message passing primitives, which remove the need to deal with the mechanics of message transportation. The functional paradigm makes the code concise, modular and easily composable. Data is *immutable* in Erlang, which implies that independent computations cannot interfere with each others data. This largely simplifies reasoning about the concurrent computations. The default implementation of the language consists of a run-time system (called ERTS) and a virtual machine (called BEAM) which offer a bunch of useful features for fault tolerance, *hot code reloading* and error handling.

1.2 The need for static typing

A prominent downside of Erlang is the lack of ability to catch type errors before runtime. This stems from the lack of a static type system. A static type system rules out such errors by disallowing compilation if a program fails type checking. A recent study shows us that static typing serves as an advantage in practice: "it does appear that strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing" [RPFD14].

Developing a static type system for Erlang has been - and still is - an active topic of research (more in section 4). This is because the ability to detect errors before runtime can be particularly valuable in a distributed and concurrent programming language. Unhandled type errors during runtime can cause programs to crash unexpectedly. In a distributed program, these errors might occur even on a remote machine, and can be hard to debug. For example, if a sender process S sends messages that a remote receiver process R does not recognize, R's mailbox may eventually become full and the process may run out of memory, or S may block forever expecting a reply.

A static type system can help catch such errors well in advance, and hence avoid the tedious task of debugging. The error in the case of message passing can be avoided by checking that the sender only sends messages of the type handled by the receiver. This is discussed in further detail in the next section.

2 A Static type system for Erlang

Given its language-level support for concurrency and flexibility for expressiveness, retrofitting a strong static type system to the entirety of Erlang can be very difficult. However, it appears that it is possible to type check the concurrent parts using a *type and effect system.* This has been hinted at multiple times in the past - such as in [MW97] and [Hug02] - but remains largely unexplored. This idea is discussed further in section 2.1. Section 2.2 deals with the challenge of typing the rest of the language, especially the ”flexible” parts. It explains the difficulties in applying a traditional static type system to these parts, and explains why partial evaluation could help increase coverage.

2.1 Typing concurrency

Erlang is based on an *actor* model of concurrency, where actors are independently executing entities equipped with communication abilities. In a very recent development, Fowler et al. define a type system for an actor model extension to lambda calculus [FLW16]. This work can be used as a basis for typing the concurrent parts of Erlang. In this section, we present some of these ideas.

An Erlang process represents an actor, and is uniquely identified by a process identifier (or PID). A process is created (or spawned) using a function. Any function can send messages using the PID of a corresponding process. Similarly, messages can be received by reading from the mailbox of the host process. In order to type check communication, send and receive operations must be assigned a type to ensure that only messages received by a process are sent to it.

Given that the PID is used to send messages, if the type of a value received by a process can be captured in the type of its PID, the type system can ensure that only messages of the expected type are sent. This is formalized in the following rules. To say e is of type τ in the typing context Γ, we write: $\Gamma \vdash e : \tau$.

$$\Gamma \vdash e : \tau \textbf{ receives } \alpha \tag{1}$$

$$\Gamma, f : () \to e \,\textbf{receives}\, \alpha \vdash spawn(f) : pid\,\alpha \tag{2}$$

$$(!) :: (pid\,\alpha,\ \alpha) \to () \tag{3}$$

where, an expression e with value of type τ and a receive statement for messages of the type α is of the type τ **receives** α. `spawn/1` (which denotes a function `spawn` with 1 argument) is a function which, when called with the name of a function as an argument, spawns a process and returns the PID. When a function of type $() \to e\,\textbf{receives}\,\alpha$ is spawned, it returns a PID of type $pid\,\alpha$. The send operation `!` is assigned a type that ensures only messages of type α are sent to a process identified by a PID of type $pid\,\alpha$

2.2 Typing the hard parts using Partial Evaluation

Haskell is a language with a static type system and strong type guarantees. This is achieved by placing some restrictions over the expressiveness of the language. For instance, a function is only allowed to return a value of one type. Such restrictions are absent in Erlang. This makes it difficult to use a Haskell-like static type system for Erlang. For example, consider the `list_to_tuple` function in Erlang which accepts a list of length n and returns an n-tuple. It is not possible to assign a simple type to this function because the result of the function has a different type for every value of n. The type of `list_to_tuple([a])` is different from `list_to_tuple([a,b])`, as {`a`} and {`a,b`} are tuples of different types.

The return type of the function depends on the value of the argument, and since the value cannot be determined before runtime, it becomes tricky to assign a type to this function at compile time. Functions of this kind are used throughout Erlang programs because they offer a great deal of flexibility to the programmer. Rejecting programs containing such functions would restrict the type checked language to a very small fragment of Erlang. Hence, that is not an option.

Moreover, some built-in functions of this nature are of crucial importance in Erlang, e.g. `spawn/4`. Consider its use in this example:

```
- module(sample).
- export([remote_hello/2]).

remote_hello(Node, Module) ->
    spawn(Node, Module, hello, []).
```

The function `remote_hello` uses `spawn/4` in its body. To type `remote_hello`, we must be able to type the `spawn/4` application in its body. For this, as in typing rule 2, we need the type of the function passed as an argument. That is, we need the type of `hello/0` defined in `Module`. But the value of `Module` is not known at compile time!

We must be able to somehow "determine" the value of `Module` to examine whether it defines a function `hello/0`. In some cases, it is possible to do so at compile time. Consider such a program:

```
remote_sample_hello(Node) ->
    M = sample,
    remote_hello(Node, M).
```

The variable `M` is assigned the value `sample`, and then passed as the argument `Module` to `remote_hello`. Given this, we can observe that the value of `Module` is available (since the value of `M` is available), and our problem simplifies to fetching the type of `hello/0` from the module `sample`.

It is possible to do more complex forms of such "determination" by *partially evaluating* a program. A partial evaluation of the above function would yield a residual function:

```
remote_sample_hello(Node) ->
    M = sample,
    spawn(Node, sample, hello, []).
```

where the call to `remote_hello` has been replaced by it's specialized body. The unknown value `Module` has been replaced with the known value `sample`. Since our definition of the module `sample` does not define a function `hello/0`, this program can be rejected by the type checker. This ensures that the program does not crash during runtime with a "bad argument" exception.

Partial evaluation [JGS93] is an evaluation technique in which the static computations in the program can be pre-computed before the actual execution of the program to yield a residual program. We propose that the residual program is relatively *simpler*, and presents opportunities for type inference. Since Erlang programs appear to contain many static values (such as module names), there are many opportunities for partial evaluation. Given the ability to simplify static computations, we can allow the "flexible" functions (such as `list_to_tuple/1`, `spawn/4` etc.) on arguments which can be statically determined. This would allow us to retain some amount of flexibility in the language while ensuring rigorous type correctness. This is the main idea behind this proposal.

3 Implementing the type system

The proposed type system involves two major components: a type checker and a partial evaluator for Erlang. Designing the type system involves developing on the typing ideas discussed above to cover more parts of the language. The type checker (and inferencer) can be implemented by performing a static analysis of the Abstract Syntax Tree (AST) via tools such as `syntax_tools` and `parse_transform`.

Given that some programs are notoriously difficult to infer types from, it may be required for some programs to be annotated with type annotations to aid the type checker. Erlang already has a type meta-language - used by static analysis tools - specified via the `-type` and `-spec` attributes. This meta-language appears suitable for our type annotation requirements.

A partial evaluator for Erlang needs to be designed and implemented. In [JGS93], Gomard and Jones implement a partial evaluator for the untyped lambda calculus using a two pass approach: by annotating an expression with static and dynamic information (called Binding Time Analysis or BTA), and then evaluating parts of the expression which have been annotated as static. Building upon this, for Erlang, BTA can be achieved by analyzing the AST for static values. `erl_eval` is an Erlang meta-interpreter which can be used for the performing the static evaluations. `parse_transform` also provides some useful utilities for evaluating Erlang terms at compile time. The residual program can then be used to aid type inference.

4 Related Work

The development of static type checking and analysis has been attempted using various approaches. Among the earliest notable efforts were that of Marlow and Wadler [MW97] to develop a type system based on *subtyping.* Their system types a subset of Erlang by solving typing constraints of the form $U \subseteq V$, which denotes that the type U is a subtype of type V. Subtyping allows for more flexible programs, and, most importantly for Erlang, allows terms to belong to multiple types. Although their work has increased type awareness among Erlang programmers, it was not adopted as their system was slow, complex, and did not cover concurrency.

The standard release of Erlang/OTP distribution includes a static analysis tool called Dialyzer [LS04]. The Dialyzer analyzes Erlang programs and identifies "obvious" type errors using type inference. If it cannot prove a program to be wrong, it will not flag an error. It favors completeness over soundness, causing the type checker to accept programs which may crash during run-time due to type errors. Ours is a more conservative approach: we favour soundness over completeness. Our proposal is based on Hindley-Milner (HM) type inference, which solves type constraints of the kind $U = V$.

Although popular in the world of strictly typed languages such as Haskell and OCaml, HM has not been a popular choice for typing Erlang. This stems from the strictness of a HM type system, which makes it unsuitable for all of Erlang. In [MW97], the authors note that HM requires terms to belong only to one type, and this makes it unsuitable for Erlang as its programmers do not adhere to this restriction. However, much has changed since their work. Erlang programmers have become more type aware and restrict themselves to writing type checkable code as a good programming practice. Moreover, HM appears to be a suitable choice in spite of its restrictions since our goal is a more modest one: to type check a subset of Erlang. Equipped with partial evaluation, such a type system would also allow the flexibility of Erlang in many cases.

5 Reflections

As discussed in the previous section, Hindley-Milner type inference imposes certain restrictions over the language. Haskell approaches this by making the user provide explicit type definitions with unique constructors which pack terms of multiple types into one type. OCaml uses *structural typing* to infer the type of a term from it's structure. Both of these approaches appear suitable for Erlang. Constructors can be mocked by the first atom of a tuple as shown in [MW97]. Structural typing could also be applicable as Erlang has simple data structures. Picking the more suitable choice requires further investigation.

Irrespective of the choice, trade-offs will have to made. The resulting type system would be effectively able to type only a subset of Erlang. Partial evaluation can help extend coverage of this type system to more parts of the language, but is limited to simplifying programs with static computations. In the absence of such "useful" static computations, it may not be possible to type the program.

The larger the type checked subset, the better the result. The goal is to see how much of this can be achieved in practice. As an ideal result, we expect the subset to be large enough to allow many programs in existing Erlang libraries. However, in the worst case, the subset would at least be a well-typed concurrent version of Erlang which allows the flexibility of Erlang in some limited cases involving static values.

References

[FLW16] Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. *arXiv preprint arXiv:1611.06276*, 2016.

[Hug02] John Hughes. Typing erlang, 2002.

[JGS93] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Peter Sestoft, 1993.

[LS04] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Asian Symposium on Programming Languages and Systems*, pages 91–106. Springer, 2004.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for erlang. *ACM SIGPLAN Notices*, 32(8):136–149, 1997.

[RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.