Master thesis planning report

Static typing of Erlang programs using Partial Evaluation

Nachiappan Valliappan, nacval@student.chalmers.se

Supervisor: John Hughes

February 4, 2018

1 Introduction

1.1 Background

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. Developing a static type system suitable for Erlang is the goal of this thesis.

1.2 Typing Erlang using Partial Evaluation

Partial evaluation [JGS93] is an evaluation technique which accepts a part of the program's input and yields a residual program, which when executed with the remaining input, yields the same output as the original program. Often, the residual programs are relatively *simpler*, and present opportunities for type inference. Since Erlang programs generally contain many static values (such as module names), there are many opportunities for partial evaluation. Consider this Erlang program:

```
- 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 assign a type `remote_hello`, we must be able to assign a type to the `spawn/4` application in its body. For this, we need the type of `hello/0` defined in `Module`. But the value of `Module` is not known at compile time! Now, consider the case where the value is indeed known at compile time:

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

Partial evaluation of the program yields:

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

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 is a simple example of a case where partial evaluation can help type check programs in Erlang.

1.3 Aim

The main idea behind this thesis is to use Hindley-Milner type inference aided by partial evaluation to type Erlang programs. This involves designing and implementing a type inferencer and a partial evaluator for Erlang.

1.4 Limitations

Hindley-Milner type inference imposes certain restrictions over the language. Terms and values cannot belong to multiple types. Haskell, for example, approaches this by making the user provide explicit data definitions with unique constructors. Although Erlang does not have built in support for constructors, they can be mocked by using an atom as the first entry of a record as shown in [MW97]. Unfortunately, existing Erlang programs do not use atoms uniquely to identify records. Hence, a static type system must support overloaded constructors based on constraint solving.

Even then, we might be able to effectively 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. We do not aim to type check all valid Erlang programs in this thesis, and instead strive to ensure that a type checked program does not crash during run-time. In other words, we favour soundness over completeness.

2 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.

3 Methodology

A static type system needs to be implemented in the front-end of the compilation pipeline. The Erlang/OTP distribution contains a module called `compile` which provides an interface to the standard Erlang compiler. This interface can be used as the entry point for both the type inferencer and the partial evaluator. The compiler directive `-compile(parse_transform, Module)` can be used to access the Abstract Syntax Tree (AST), where `Module` exports a function `parse_transform(Forms, Options) -> Forms` which is called by the compiler during compilation, and it's result replaces the AST to be compiled. This allows us to make arbitrary transformations to the program to be compiled.

3.1 Type Inference for Erlang

The type checker - at least initially - would simply be a type inferencer which attempts to infer types from a given program. If the inferencer manages to infer a valid type, then the program is type correct. A type inferencer can be implemented as a parse transformation module which accesses the AST through `parse_transform` and attempts to infer the types as a side-effect. In the case where a type cannot be inferred, a type error can be reported silently by printing the error or by crashing the compilation using `erlang:error(Reason)`.

3.2 Partial Evaluation for Erlang

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. For Erlang, we could require that the programmer provide annotations to indicate potential for partial evaluation to the compiler. This would largely simplify BTA. The evaluation, on the other hand, can be achieved by using `erl_eval` -

an Erlang meta-interpreter. The residual program can be used as the subject of type inference.

4 Plan

The main goal of the thesis is a rather broad one and is difficult to objectify. The primary difficulty in quantifying the result lies in being able to assess the accomplishment of the goal. However, the expected end products are clear: a usable type checker and a thesis report.

An ideal type checker for Erlang should be able to type check many existing Erlang programs while not compromising on soundness. Given the flexibility of Erlang semantics, it may be the case that it is not possible to type check some Erlang programs without code modification. Here, "modification" can be quantified as number of lines added (or changed) to aid the type checker. If $X\%$ is the amount of change required to type check existing Erlang code, a concrete goal would be to strive for a specific upper bound for the value of X. This can in turn be evaluated against a few selected Erlang/OTP libraries.

This section attempts to provide a holistic breakdown of the type checker's goals, potential assessment criteria, and a tentative time allotment for each sub-goal.

4.1 Sub-goals

The main goal of implementing the type checker can be divided into the following sub-goals for type checking:

1. Simple expressions, functions
2. Branching expressions
3. Recursive functions
4. Simple data types
5. Recursive data types, Lists
6. Overloaded constructors
7. Partial evaluation
8. Extension: Exceptions
9. Extension: Messaging

Sub goals 1 - 3 are the absolute minimum requirement to type check any valid Erlang program. 3 requires some special attention as Erlang does not have

special syntax or annotations for functions to be marked as recursive.

With the introduction of the Dialyzer and type annotations, data type definitions were introduced to Erlang. These type definitions - especially the syntax - can be leveraged to support user defined data types in our type system. This is the primary purpose of sub-goal 4. Sub-goal 5 takes this further to provide full support for recursive ADTs.

As discussed earlier, a type system for Erlang needs to support overloaded constructors as requiring the first atom of a record to be unique would be an arbitrary constraint which is not adhered to in existing Erlang programs. This is covered by sub-goal 6.

Sub-goal 7 is an important and significant part of the project. Partial evaluation is a tool for simplifying - and in many cases, make possible - type checking, and hence a feature in itself which can take significant time to implement. Hence, we also consider the development of a partial evaluator as a sub-goal.

Sub-goals 8 and 9 require the modelling and addition of *effects* to the type system. These can be considered extensions to the type checker which results from the previous stages.

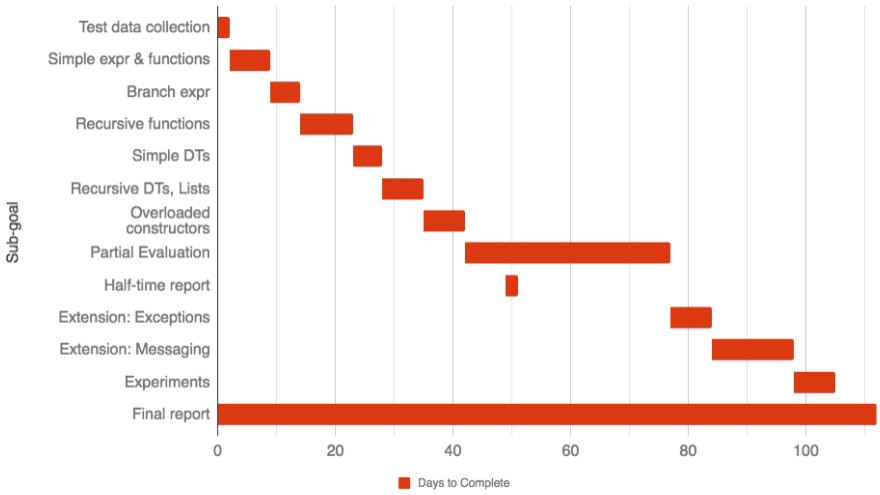
4.2 Experiments & Assessment

In order to assess the coverage of the type checker, we would need to run it against existing Erlang programs. For this, we need to collect interesting Erlang programs to run the type checker on. A small period of time has been assigned for this task in the next section. One option would be to aim to type check some Erlang/OTP libraries.

The type checker must be able to type check the selected Erlang programs with less than 10% modification to the code. This value derives from an estimate of being able to add a type signature to every function definition, in the worst case.

4.3 Time plan

On the basis of the breakdown in section 4.1, the type checker will be implemented in an incremental fashion to cover more and more parts of the language. In our estimate, this requires approximately 16 weeks. The following Gantt chart lays out a tentative time plan, where each time slot is allocated towards achieving a specific sub-goal.



The thesis report will be written during the entire process recording difficulties, observations and successes. One working day from every week will be dedicated towards writing the report and discussing the progress with the supervisor.

5 Reflections

As discussed earlier, the type system is aimed only at accepting a subset of Erlang programs. The "subset" here refers to a subset of all semantically valid Erlang programs. The subset is not defined by supporting only parts of the language, but rather from programs being sound and "type checkable". The larger the type checked subset, the better the result. The goal is to see how much of this can be achieved in practice.

References

[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.