Typing the Wild in Erlang

Nachiappan Valliappan
Chalmers University
Sweden
nachivpn@gmail.com

John Hughes
Chalmers University, Quviq AB
Sweden
rjmh@chalmers.se

Abstract

Developing a static type system suitable for Erlang has been of ongoing interest for almost two decades now. The challenge with retrofitting a static type system onto a dynamically typed language, such as Erlang, is the loss of flexibility in programming offered by the language. In light of this, many attempts to type Erlang trade sound type checking for the ability to retain flexibility. Hence, simple type errors which would be caught by the type checker of a statically typed language are easily missed in these developments. This has us wishing for a way to avoid such errors in Erlang programs.

In this paper, we develop a static type system for Erlang which strives to remain sound without being too restrictive. Our type system is based on Hindley-Milner type inference, however it—unlike contemporary implementations of Hindley-Milner—is flexible enough to allow overloading of data constructors, branches of different types etc. Further, to allow Erlang's dynamic type behaviour, we employ a program specialization technique called partial evaluation. Partial evaluation simplifies programs prior to type checking, and hence enables the type system to type such behaviour under certain restricted circumstances.

CCS Concepts • **Software and its engineering → Functional languages**; **Polymorphism**; *General programming languages*; Data types and structures;

Keywords Erlang, Type Inference, Partial Evaluation

ACM Reference Format:
Nachiappan Valliappan and John Hughes. 2018. Typing the Wild in Erlang. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3239332.3242766

Erlang '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242766

1 Introduction

Erlang is a dynamically typed concurrent functional programming language popular in distributed applications. Since Erlang is dynamically typed by design, the Erlang compiler does not type check Erlang programs during compilation. It allows the successful compilation of ill-typed programs which would be rejected by a Haskell or ML compiler. As a result, simple type errors which can be detected at compile time are not discovered until the program is executed.

Dialyzer is a static analysis tool which helps identify such type errors. It has been widely adopted by the Erlang community, and type specifications understood by the Dialyzer can be found in most Erlang/OTP libraries these days. The type system employed by Dialyzer is based on the idea of *success typing* [7], which means that it does not report false positives (type errors in a well-typed program). The type checker takes an optimistic approach and assumes that a program is well-typed unless it can prove otherwise. If it cannot prove a type error, it accepts the program—even if the program is unsafe—and may hence miss type errors. For example, the following ill-typed program, which crashes at run-time on the invocation of `find/0`, is accepted by Dialyzer.

```
-type maybe(A) :: none | {ok,A}.

lookup(K,[])         -> none;
lookup(K,[{K,V}|_]) -> {ok,V};
lookup(K,[_|KVs])    -> lookup(K,KVs).

find() ->  {ok,"s"} = lookup(0,[{0,1}]).
```

The purpose of type checking is to prevent programs from crashing at runtime. However, determining whether a program will crash at runtime is undecidable. Static type checkers take a conservative approach and reject more programs than necessary to guarantee the absence of runtime type errors. Dialyzer, on the other hand, does not provide this guarantee. As observed earlier, it accepts programs which may crash at runtime.

The success of static type checkers for many programming languages makes us wonder if it is possible to develop one for Erlang without being too restrictive. Languages such as Java and C++ implement static type checking by making the programmer specify explicit type annotations. But annotations can get tedious and often clutter the code. Instead, languages such as Haskell and ML implement *type inference* to free the programmer from having to specify excessive

annotations. Type inference implemented by Haskell and ML are both based on a type system popular for this purpose: the Hindley-Milner type system [2] [9].

Most successful adaptations of the Hindley-Milner type system to practical functional programming languages implement *Algebraic Data Types* (ADTs). ADTs are a feature which allow the programmer to define new types using user defined data constructors. Here's an example of an ADT which defines a tree data type in Haskell:

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)
```

`Nil` and `Node` are constructors of the tree data type. `Nil` is a nullary constructor, which constructs a term of type `Tree a`, and `Node` is a three argument constructor, which constructs a term of type `Tree a` when given a term of type `a` and two terms of type `Tree a` as arguments. ADTs give the programmer control over the types of a program, and constructors of an ADT act as tags for type inference.

Erlang, however, does not have ADTs. Data construction is typically done using *tagged tuples*. A tagged tuple is a tuple, where an atom is used as the first field to identify a specific construction. If the construction contains only an atom, then the atom is used in place of a tagged tuple. In the following example, `nil` and `node` are atoms used to identify constructions of the tree data structure (we'll use this example multiple times in this paper):

```
findNode(_,nil) ->
    false;
findNode(N,{node,N,Lt,Rt}) ->
    true;
findNode(N,{node,_,Lt,Rt}) ->
    findNode(N, Lt) orelse findNode(N,Rt).
```

One may consider such atoms as constructors and the remaining fields of a tagged tuple as arguments to the constructor. By treating this programming pattern as data constructor applications, it is possible to apply a constructor based type system.

The real challenge, however, is that none of the Hindley-Milner implementations allow constructors to be overloaded, and applying this restriction to Erlang can be very limiting. For example, consider the construction `{ok,Value}`. It is used across programs in many different contexts. Forcing the `ok` constructor to belong to one specific ADT definition can not only be cumbersome, but also practically impossible as it would require a lot of Erlang code to be re-written. For this reason, most notable efforts to typing Erlang [6][7][8] reject the Hindley-Milner type system.

Instead, they opt for type inference based on *subtyping*. But experience shows us that subtyping doesn't mix well with type inference. Type signatures inferred in a subtyping system can often be large and hard to understand—which defeats the whole purpose of a type checker. In practice, Dialyzer (which uses subtyping) also appears to be much slower than a Haskell/ML type checker.

Restricting constructors to have a unique type has been considered essential in a Hindley-Milner type system. However, in this paper, we show that it is possible to type overloaded data constructors and yet retain the type inference properties of a Hindley-Milner type system. We present an inference based type system for Erlang with a form of adhoc polymorphism for typing overloaded constructors. This leads to much faster type checking and comprehensible type signatures and errors. The type system and its implementation are discussed in section 3.

A static type system based on type inference alone can be quite restrictive. Erlang's flexible programming utilities such as `list_to_tuple/1`, `element/2`, `is_function/1`, etc., cannot be assigned a type easily in a Hindley-Milner type system. The difficulty arises from their *dynamic type behaviour*, i.e., the type of the function depends on the value of their arguments (which may not be known until runtime). To solve this problem, we employ an evaluation technique called *partial evaluation* [5]. Partial evaluation *reduces* an expression when one or more of its components can be computed at compile time. The reduced expression is often simpler and the dynamic type behaviour may be removed. Our type system uses partial evaluation to simplify certain expressions before typing them, and hence allows restricted forms of dynamic type behaviour in programs. This is discussed in further detail in section 4.

To develop an intuition for the types inferred by the type checker, we illustrate the type checker on various examples in section 2. The type checker has also been applied to a few small libraries, including a couple of OTP libraries. The initial results look promising, and are discussed further in section 5.

Applying Hindley-Milner to Erlang requires us to make certain simplifying assumptions (such as homogeneous lists) and place restrictions which are typically absent in Erlang programming. The use of ADTs for typing means that the programmer must supply additional type definitions for compound data types. Another important limitation is that our type system does not type interprocess communication. These limitations are discussed in section 6.

Our type system defines a flavour of typed Erlang programming which looks much like programming in Haskell. Our effort shows that it is possible to apply a type checker based on sound typing principles to Erlang without making a radical change to the programming style. Two key reasons for this are the typing of overloaded constructors in Hindley-Milner, and the application of partial evaluation. Overloaded constructors free the programmer from worrying about defining context specific constructor names, and partial evaluation helps retain Erlang's flexibility (when expressions can be reduced at compile time).

2 Erlang Type Inference, by Example

Type inference computes a type for a function, given the function's body as an input. In this section, we run the type checker on various examples and discuss the inferred types. These examples illustrate the features of the type checker, which are discussed in later sections. This section serves as an informal introduction to type signatures and type inference.

2.1 Lists

The following Erlang function appends two lists and returns the resulting list.

```
append([H|T], Tail) ->
    [H|append(T, Tail)];
append([], Tail) ->
    Tail.
```

The type checker infers the type:

$$append/2 :: ([A], [A]) \rightarrow [A]$$

Here, A is a polymorphic type variable which indicates that list elements can be of any type. $[A]$ is the type of a list where all elements are of type A (we implement homogeneous lists in our type system, and hence all elements of a list must be of the same type). The inferred type signature for append states that the function accepts two arguments of type $[A]$ and returns a list of type $[A]$.

The inferred type of append is *polymorphic* over the type variable A, meaning that it can be used on any two lists of the same type. When it is applied to two lists of a specific type, the type variable A is instantiated with the type of the elements in the lists. For example, when it's applied to lists of type $[boolean()]$, A is instantiated with $boolean()$, and the type of append is specialized to $([boolean()], [boolean()]) \rightarrow [boolean()]$.

To understand how the type checker infers this type, note that the second function clause returns the second argument of the function. Hence, the return type of the function must be the same as the type of the second argument. Moreover, the first clause appends the head of the first argument list to the result of append, and so the first argument must be of the same type as the result. Using this information, the type checker infers that the arguments and the return value must all be of the same type.

2.2 Numeric Types

In Erlang, there are two types of numbers: integers and floats. Some operations (such as div) are allowed to operate only on integers, whereas other operations are overloaded over both integers and floats (such as + and *). Our type system allows overloading by implementing a simple *type class* system for Erlang (inspired by Haskell's type classes). Type classes allow us to assign polymorphic types to operators, restricted by a constraint. For example, the + operator (which is overloaded over integers and floats) is assigned the type:

$$(+) :: Num\ A \Rightarrow (A, A) \rightarrow A$$

where $Num\ A$ is a constraint on the type A that asserts that A must be a numeric type (that is, an integer or a float). Note that both the operands are expected to be of the same numeric type—a simplifying assumption in our type system. When A is instantiated with a concrete type in an application of +, the type checker checks whether the type constraint can be solved. If so, the application is accepted, otherwise it is rejected with a type error.

For instance, in the expression `40.0 + 2.0`, A is instantiated with $float()$, and the type constraint is specialized to $Num\ float()$. The type checker knows that $Num\ float()$ is solvable, and hence accepts the expression as well-typed. If A is instantiated with an non-numeric type, such as $boolean()$, the type checker reports a type error as it cannot solve the constraint $Num\ boolean()$.

Let's look at the type assigned to an expression which uses +. Consider this function:

```
sum ([]) -> 0;
sum ([X|Xs]) -> X + sum(Xs).
```

It computes the sum of a given list. In its second clause, the + operator is applied to an element of the argument list and the result of sum. Since the type of + requires the arguments and the result to have the same numeric type, then the elements of the list and also the computed sum must be of the same numeric type. Using this information, the type checker infers the type:

$$sum/1 :: Num\ A \Rightarrow ([A]) \rightarrow A$$

This type correctly states that sum maybe used for either integers or floats.

An operator restricted to arguments of specific numeric type, such as div which is restricted to integers, is assigned a type as follows:

$$div :: (integer(), integer()) \rightarrow integer()$$

On the other hand, the division operator /, which can be applied to operands of any numeric types, is assigned a type using type constraints:

$$(/) :: (Num\ A,\ Num\ B) \Rightarrow (A, B) \rightarrow float()$$

Note that the operands need not be of the same type—they may be an integer and a float, for instance. As an example, consider the following average function:

```
average(Xs) -> sum(Xs) / length(Xs).
```

It uses the / operator to divide the sum of a list (a numeric type) by its length (an integer) to return a float. The type checker infers the type:

$$average/1 :: Num\ A \Rightarrow ([A]) \rightarrow float()$$

2.3 Algebraic Data Types

ADTs are used to define new types using user defined constructors, which may optionally accept a number of arguments. An ADT definition must declare its constructors and the types of their arguments. In the following example, `tree(A)` is an ADT parametrized over the type `A`.

```
-type tree(A) :: nil
    | {node, A, tree(A), tree(A)}.
```

This is the Erlang version of the Haskell tree ADT we saw earlier. The nullary constructor `nil` can be defined as an atom (as shown), or as a single element tuple (such as `{nil}`). The three argument constructor `node` is defined as a four element tuple, where the first element is the atom `node` and the remaining elements are the types of its arguments.

A term of an ADT can be constructed by providing the arguments to a constructor, also called a *constructor application*. When a type checker encounters an atom or a tuple whose first element is an atom, it considers it as a constructor application—provided the atom has been defined as a constructor in some ADT.

Now, let's understand type inference for ADTs. Recollect the `findNode/2` example from section 1. The `findNode` function pattern matches on the constructors of the tree data type to search for a given node value and returns a boolean indicating success or failure. Since the second clause of the `findNode` function matches the given value directly with a value in the node of a tree, the type checker infers that the values of the nodes in the tree must be of the same type as the given value. As a result, the inferred type of this function is:

$$findNode/2 :: (A, tree(A)) \rightarrow boolean()$$

Note that the ADT definition must be provided for this type to be inferred. In the absence of an ADT definition for the above example, `nil` and `node` are simply treated as atoms. As a consequence, the type checker would reject the `findNode` function as it expects the arguments to have the same type on all clauses—a property which fails to hold for the second argument in this case.

2.4 Overloaded Data Constructors

Overloaded constructors make type inference tricky. Consider the following example where the constructor `nil` could construct a list or a tree.

```
-type list(A) ::
    nil | {cons, A, list(A)}.
-type tree(A) ::
    nil | {node, A, tree(A), tree(A)}.

empty () -> nil.

flattenTree(nil) ->
    [];
flattenTree({node,N,Lt,Rt}) ->
    flattenTree(Lt) ++ [N|flattenTree(Rt)].
```

In the case of `flattenTree`, it is easy to see that it operates on trees, and not on lists, because the second function clause pattern matches on `node`—which only appears in the tree data type. Hence, `flattenTree` is assigned the type:

$$flattenTree/1 :: (tree(A)) \rightarrow [A]$$

But what should the inferred type of `empty` be? Should the return type be a list or a tree? Since the type checker lacks the reason to make a choice, it infers a type allowing `empty` to be used with either type:

$$empty/0 :: (D \sim \{tree(A),\ list(B)\}) \Rightarrow () \rightarrow D$$

This type denotes that `empty` is a nullary function which returns a value of type D, under the constraint that D is a tree or a list. When it's called to return a list, the type constraint gets specialized to $(D \sim \{list(B)\})$, and hence D gets instantiated with $list(B)$. Similarly, in the expression `flattenTree(empty())`, since `flattenTree` expects a tree argument, the type constraint gets specialized to $(D \sim \{tree(A)\}$, and hence D is instantiated with the type $tree(A)$ to yield the type of the expression as $[A]$.

2.5 Messaging

At the heart of Erlang's concurrency model lies message passing between processes. Our type system does not check whether the types of the messages sent to a process match the types of the messages it expects. However, messaging primitives such as `!` (send), `receive`, `spawn/1`, etc., are used extensively in Erlang, and they must be assigned a type in order to type check Erlang programs. This section illustrates the types assigned to such primitives and the inferred types of expressions which use them.

`spawn/1`, which is used to spawn nullary functions, is assigned the type:

$$spawn/1 :: (() \rightarrow A) \rightarrow pid()$$

where the return type $pid()$ is the type of a process identifier (or pid). Our type system does not differentiate between pids of different processes, and all pids are assigned the type $pid()$.

The `!` operator, which sends a message to a process, is assigned the type:

$$(!) :: Padd\ A \Rightarrow (A, B) \rightarrow B$$

where $Padd\ A$ is type constraint over the first argument of type A (the destination), and the return type B is also the type of the second argument (the message) . $Padd$ (for Process address) is a type constraint which restricts the first argument to a pid, an atom (a registered name) or a tuple of two atoms (registered name and node).

The `receive` expression, on the other hand, is similar to a `case` expression, but is used to pattern match over messages in the inbox of a process. The type checker expects

all the patterns of the `receive` expression to be of the same type. This may initially appear to be a limitation as it is quite common to pattern match over different types of messages. However, this can be easily overcome by adding an ADT definition which combines the types of the messages. Consider the following example:

```
-type request()  :: {ping, pid()}
    | {get_sum,pid(),integer(),integer()}.

server() ->
    receive
        {ping, Ping_PID} ->
            Ping_PID ! {pong, self()};
        {get_sum, Pong_PID, X, Y}  ->
            Pong_PID ! {sum, X + Y}
    end,
    server().
```

The `receive` expression is well-typed because `ping` and `get_sum` are defined as constructors of the same type in the `request()` ADT, hence making the patterns to be of the same type. Note that there is no such requirement for the clause bodies of the `receive` expression. The type of the first clause body is $\{atom, pid()\}$ and that of the second clause body is $Num\ A \Rightarrow \{atom, A\}$—clearly different types.

The clause bodies of `case`, `if` and `receive` expressions are not expected to be of the same type unless their return value is used. In the above example, the value returned by the `receive` expression is discarded, and hence the bodies need not be of the same type. When the return value is used, all bodies are expected to be of the same type—in order to assign a single type to the returned value.

3 Implementing Typing Inference

The original Hindley-Milner type system (as in [2], for instance) is far too simple for a real programming language such as Erlang. For example, it does not support overloading, and as we've seen earlier, overloading is required to type Erlang's functions and data constructors. Programming languages such as Haskell and ML, which base their type system on Hindley-Milner, use a variation of it by adding several extensions. Haskell's type system allows overloading of functions by implementing a form of adhoc polymorphism called type classes. However, none of these languages allow data constructors to be overloaded, and this is an absolute requirement for typing Erlang.

The type system we present for Erlang is also based on Hindley-Milner, but it supports overloading of both functions and data constructors. For overloading functions, we implement a type class system similar to Haskell's. Whereas, for overloading constructors, we implement a constraint system closely related to type classes.

In this section, we are concerned with the details of implementing the type system. We first introduce the basic Hindley-Milner type system (section 3.1 - 3.2), then gradually add more features towards overloading functions and constructors (sections 3.3 - 3.6), and finally present a solution for typing records in Erlang (section 3.7).

3.1 Overview of Hindley-Milner

In this section, we introduce key concepts of the Hindley-Milner type system such as type variables, *unification* and *generalization*. Readers familiar with Hindley-Milner may skip this section.

Type variables are central to the Hindley-Milner type system. A type variable represents an unknown type. It can be instantiated with a base type (such as *integer*() or *boolean*()) or left as it is itself until more information is available, i.e, a type variable is also a valid type. The types in Hindley-Milner can be described using the grammar:

$$\begin{aligned}\langle type\rangle ::= &\ \langle base\rangle \\ \mid &\ \langle tvar\rangle \\ \mid &\ (\langle type\rangle,..,\langle type\rangle) \rightarrow \langle type\rangle\end{aligned}$$

where $\langle base\rangle$ represents a base type, $\langle tvar\rangle$ represents a type variable, and $(\langle type\rangle, .., \langle type\rangle) \rightarrow \langle type\rangle$ represents a function type.

Instantiation of type variables is done using a *substitution*. A substitution is as a mapping of type variables to types. The mapped variables are called the domain of the substitution and the types it maps to are called the co-domain of the substitution. A substitution σ is *applied* to a type t—denoted as $\sigma(t)$—by replacing all free occurrences of the type variables in t belonging to the domain of σ by their corresponding values in the co-domain of σ. For example, when the substitution $\{X \mapsto boolean()\}$ is applied to the the type $(X) \rightarrow X$, it yields the type $(boolean()) \rightarrow boolean()$. Note that the notion of applying a substitution is the standard one which replaces only free variables and accounts for name capture.

During type inference, a type is considered to be *partially known* if it contains type variables. Unification is a process which takes two partially known types that are expected to be equal and instantiates the type variables in them to ensure that it is indeed the case. For example, unification of the types $(X) \rightarrow X$ and $(boolean()) \rightarrow Y$ yields the substitution $\{X \mapsto boolean(), Y \mapsto boolean()\}$, which when applied to the types equalizes them. Formally, unification is the process of computing a substitution σ that equalizes two types t_1 and t_2 when applied to them, as in, $\sigma(t_1) = \sigma(t_2)$. Unification is used by type inference to ensure that two types are of the same type. For example, it is used to ensure that both sides of a match expression are of the same type, or to ensure that all patterns of the case expression are of the same type, etc.

The unifying substitution is also called the unifier. Note that there may be several (or no) unifiers for any two given types. Type inference uses the *most general unifier* (mgu) to compute the most general type. A substitution σ is said to be the mgu of two types if for every other unifier σ' of the

types, there exists γ such that $\sigma' = \gamma \circ \sigma$, where $\gamma \circ \sigma$ is the *composition* of the two substitutions and is defined as $\gamma(\sigma(t))$. That is, σ is the mgu if all other unifiers can be expressed in terms of it.

Another important feature of the Hindley-Milner type system which is used to allow generic programming is polymorphism. Intuitively, a function is polymorphic if it can be used in different contexts with different types. To achieve this, the polymorphic function is assigned a generic *type schema*. A type schema acts as a representation of all the valid types that the function can be assigned. A type schema can be described using the grammar:

$\langle schema \rangle$ ::= $\langle type \rangle$
| $\forall \langle tvar \rangle . \langle schema \rangle$

To generate a type schema, type inference employs a technique called generalization. Generalization converts the inferred type of a function to a type schema. An example of generalization is the conversion of $(T) \rightarrow T$ to $\forall T.(T) \rightarrow T$. Generalization is achieved by binding all the type variables in a type that do not occur freely in the *environment* (which assigns types to free variables of a term).

When the function is applied in a certain context, the type schema is instantiated to yield the context specific type of the function. A type schema is instantiated by replacing all the bound variables with fresh type variables. For example, instantiation of $\forall T.(T) \rightarrow T$ yields $(P) \rightarrow P$ for some fresh type variable P.

3.2 Beyond Hindley-Milner

Type inference for Erlang requires techniques well beyond simple Hindley-Milner. For instance, a recursive function in Hindley-Milner is defined using an explicit fix point combinator. Some modern implementations of Hindley-Milner, such as OCaml [10], for instance, require the programmer to annotate recursive functions explicitly. But Erlang has no such construct as this problem is irrelevant for a dynamically typed language. Hence, we need an approach to type inference that treats recursive and non-recursive functions alike. The standard solution to this problem is to assign a fresh type variable to the function in the environment it is being type checked in, and then unify the inferred type with the assigned type. This way, the function has a type when its type is being inferred and it is also enforced to be the same as the inferred type.

The case of mutually recursive functions is a little more complex. OCaml requires that programmers define mutually recursive functions using the same recursive let. Haskell, on the other hand, has no such requirement. Programmers can write mutually recursive bindings freely without any annotations or grouping. Haskell achieves this by doing a kind of dependency analysis to group all mutually recursive functions and then performing type inference on them in the order of their dependency. Our implementation is based on Haskell's technique. For further details, we refer the interested reader to the implementation of Haskell's type inference [4].

3.3 Type Classes

Another requirement beyond Hindley-Milner to type Erlang is the overloading of operators and functions. To achieve this, we implement a simple type class system for Erlang. In this section, we present a brief of overview of type classes and how they are implemented. For more details, we refer the interested reader to Haskell's implementation of type classes [4] (which is the basis for our implementation).

Type classes are essentially a way to group types. A type class has a name and a group of types which are referred to as its *instances*. For example, Num is a type class, and $integer()$ and $float()$ are its instances. In Haskell, the programmer can define new type classes and extend existing ones. However, in our type system, type classes are a purely built-in feature. The list of all valid type classes and their instances (also called the type class premise) is a pre-defined constant. For the reader familiar with Haskell's type classes, also note that there is no class hierarchy in our system.

A type class constraint (which we've seen earlier in our examples) contains a type class and a type variable, and it specifies that the type which replaces the type variable must be an instance of the type class. For example, the constraint $Num\ A \Rightarrow \ldots$ specifies that a type which replaces A must be an instance of Num.

A type class constraint over the type of a function is coupled along with the type in its type schema [1]. To add type constraints to a type schema, we modify the type schema grammar from Hindley-Milner to:

$\langle schema \rangle$ ::= $\langle type \rangle$
| $\forall \langle tvar \rangle . [\langle constraint \rangle] . \langle schema \rangle$

$\langle constraint \rangle$ ::= $\langle class \rangle . \langle tvar \rangle$

When a type schema is instantiated, the type variables in the constraints are also replaced with fresh type variables. For example, instantiating the type schema $\forall T.Num\ T \Rightarrow T \rightarrow T$, yields a type $U \rightarrow U$ and a type constraint $Num\ U$ (for some fresh variable U).

All type class constraints which arise from a function's body during type inference are collected as *class predicates* to be solved later. A class predicate $\{class, c, i\}$ is an assertion that the type i is an instance of the class c. For example, the predicate $\{class, "Num", integer\}$ asserts that the type $integer()$ is an instance of the class Num. The difference between a predicate and a class constraint is that the type i in a predicate need not be a type variable. Moreover, as we will see later, class predicates are not the only predicates.

[1] In earlier sections, the inferred type signatures for functions which contain type class constraints are actually type schemas. Typically, type schemas are also considered as types (by excluding the quantifiers) as a simplification.

To understand the collection of type class constraints as predicates, consider the `average/1` function which we saw earlier:

```
average(Xs) -> sum(Xs) / length(Xs).
```

The instantiation of the `/` operator's type schema generates two type class constraints (one on each operand): $Num\ A$ and $Num\ B$, where A is the expected type of the first operand and B is the expected type of the second operand. These type constraints are collected as the following predicates:

$$\{class, "Num", A\}, \{class, "Num", B\}$$

Now, suppose that the inferred type of `sum(Xs)` is $Num\ T \Rightarrow T$ for some type variable T. Since the inferred type of an expression is unified with the expected type, type inference unifies A with T, and as a result instantiates A with T. Similarly, since the inferred type of `length(Xs)` is $integer()$, B is instantiated with $integer()$. These instantiations specialize the predicates to:

$$\{class, "Num", T\}, \{class, "Num", integer()\}$$

The collected predicates are then solved by simply eliminating all valid predicates which follow directly from the premise. Hence, $\{class, "Num", integer()\}$ is removed, leaving the predicate:

$$\{class, "Num", T\}$$

The remaining predicates are then generalized along with the type to yield the type schema of the subject function. In this case, $\{class, "Num", T\}$ is generalized along with the inferred type of the function $([T]) \rightarrow float()$ to yield the type schema $\forall T.Num\ T \Rightarrow ([T]) \rightarrow float()$—which we saw as the inferred type of `average` earlier.

3.4 ADTs

To implement ADTs in the type system, we need a way to add user defined types to the type system, a mechanism to assign these types to user defined constructors, and an inference algorithm to infer the types of data constructor applications in expressions. We address these needs in this section.

User defined types, such as $tree(A)$, are added to the type system using a *type constructor*. Just like a data constructor accepts some data arguments to construct a data value, a type constructor accepts some type arguments to construct a type. It can be defined as an extension to the grammar of types from the Hindley-Milner system as:

$\langle type \rangle$ ::= ...
| $\langle constructor \rangle$ [$\langle type \rangle$]

where $\langle constructor \rangle$ represents the name of the type constructor, and [$\langle type \rangle$] represents the list of type arguments. For example, in the type $tree(A)$, $tree$ is the type constructor and the type variable A is its argument.

A data constructor constructs a term of a user defined data type when given some arguments. In this sense, a data constructor is exactly like a function. Hence, it is assigned a function type, where the argument types are the argument types of the constructor and the return type is the type defined by its corresponding ADT. In the tree ADT, `nil` is assigned the type $nil/0 :: () \rightarrow tree(A)$, and node is assigned the type $node/3 :: (A, tree(A), tree(A)) \rightarrow tree(A)$.

To implement type inference for data constructor applications, we must first understand how type inference is implemented for function applications.

In a function application $f(x_1, ..., x_n)$, the types of the arguments given to f must match the arguments expected by it, and the inferred type of the application must be the return type of the f. To implement this, we first lookup the type of the function f in the type inference environment, and then we infer the types of the arguments. Let the inferred type of the function be T and the inferred type of the arguments be $(A_1, ..., A_n)$. T is then unified with the type $(A_1, ..., A_n) \rightarrow V$ (where V is a fresh variable) to yield a unifier σ. And finally, the type of the application is the type V specialized using the result of the unification, i.e., $\sigma(V)$.

A data constructor application is similar to function application. It merely has a different syntax $\{c, x_1, ..x_n\}$, where c is the constructor and $x_1, ..x_n$ are its arguments. If c is a unique constructor of a data type, then c is assigned a single type in the environment, and the treatment of the constructor application is no different from function application. However, if c is overloaded, then it has more than one type in the environment and the lookup for the type of c results in list of types. Which one should be used for unification? The treatment of the latter case requires more sophisticated techniques, which is the focus of the next section.

3.5 Overloading Data Constructors

When the lookup of an overloaded constructor returns a (non-empty) list of types $[T_1, ..., T_n]$, the type of the application $(A_1, ..., A_n) \rightarrow V$ (discussed in the previous section) may unify with more than one of these types. Since we cannot make a decision on exactly one type of $[T_1, ..., T_n]$ at the time, we *defer* this unification by generating a new kind of predicate called the deferred unification constraint (duc) predicate:

$$\{duc, (A_1, ..., A_n) \rightarrow V, [T_1, ..., T_n]\}$$

which asserts that the type $(A_1, ..., A_n) \rightarrow V$ must eventually unify with exactly one type from the list $[T_1, ..., T_n]$ (called the candidate types).

Like class predicates, duc predicates generated during type inference of a function are collected and then solved before generalization. Solving them later—as in the case of class predicates—allows us to use specializing information which is generated during type inference. Once a duc predicate is specialized, it may be possible to reduce the list of candidate types. If the candidate type can be reduced to exactly one, then the duc predicate is solvable and the unification is performed using the remaining candidate type. Otherwise,

the duc predicate is generalized along with the type of the function as a type constraint.

Now, consider type inference for the `findNode` example (from section 1). The first clause has an overloaded constructor `nil` as an argument, which generates the predicate:

$$\{duc, () \rightarrow V, [() \rightarrow tree(A), () \rightarrow list(B)]\}$$

where $() \rightarrow V$ is the inferred type of the constructor application. Notice how this type unifies with both the candidate types (with unifiers $\{V \mapsto tree(A)\}$ and $\{V \mapsto list(B)\}$, and hence at this stage the predicate is not solvable. However, the occurrence of the `node` constructor in the second clause, instantiates V to $tree(C)$ (the return type of `node`, for some type variable C), hence specializing the predicate to

$$\{duc, () \rightarrow tree(C), [() \rightarrow tree(A), () \rightarrow list(B)]\}$$

Evidently, only one of the candidates types is now unifiable, and hence we may reduce the predicate to:

$$\{duc, () \rightarrow tree(C), [() \rightarrow tree(A)]\}$$

This predicate is now solvable, and the unification can be performed to yield the substitution $\{C \mapsto A\}$, which is then applied to the inferred type $(C, tree(C)) \rightarrow boolean()$ to yield the type $(A, tree(A)) \rightarrow boolean()$.

In the case where a duc predicate is not solvable, it is generalized along with the type of the function as a type constraint called the duc type constraint. The duc type constraint is defined by extending the constraint grammar as:

$\langle constraint \rangle$ ::= ...
| $\langle type \rangle \sim \{\langle type \rangle, ..., \langle type \rangle\}$

A constraint $T \sim \{T_1, T_2, ..T_n\}$ specifies that the type T must eventually unify with exactly one of the types $T_1, T_2, ..T_n$. These constraints are, like type class constraints, added to the type schema of a function. To do this we extend the type schema grammar as:

$\langle schema \rangle$ ::= ...
| $\langle constraint \rangle \langle schema \rangle$

For example, in the case of `empty()` (from section 2.4), due to lack of specializing information the generated duc predicate is generalized along with the type of the function to yield the type schema with a type constraint $(D \sim \{tree(A), list(B)\})$.

However, unlike the case of class predicates, simply retaining the unsolved duc predicates as type constraints can lead to loss of type information in the case where no specializing information is available. To understand this problem, consider this example:

```
-type sr(R) :: {'EXIT', pid(), R}.
-type cl(R) :: {'EXIT', pid(), R}.

getReason({'EXIT', _, Reason}) -> Reason.
```

The type `{'EXIT', pid(), R}` represents an exit signal sent by a process before its exit with its pid and reason for exit. The `getReason` function here extracts the reason from such a signal. Given the defined ADTs, one expects to see the inferred type as:

$$getReason/1 :: C \sim \{cl(B), sr(B)\} \Rightarrow (C) \rightarrow B$$

which specifies that the argument is of type C, where C unifies with $sr(B)$ or $cl(B)$, and the return type is B. But, without any simplification of duc predicates, the type checker infers the type:

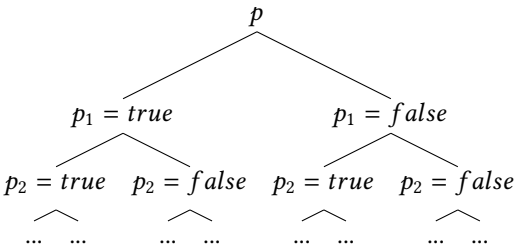
$$getReason/1 :: (A, B) \rightarrow C \sim \{(pid(), B) \rightarrow cl(B), (pid(), B) \rightarrow sr(B)\} \Rightarrow (C) \rightarrow B$$

The reason for this is that the generated predicates that cannot be solved have simply been generalized as type constraints. Although it's possible for us to see from the generated type constraints that A always unifies with $pid()$ and B always unifies with B, this information has not been exploited by the type checker to simplify the type constraint.

The problem is not just one about simplification. There are cases in which not exploiting the information in the type constraints can lead to missing type errors (a concrete example of such a case can be found in section A.2). In the next section, we discuss a solution to this problem by applying a proof procedure technique from classical propositional logic.

3.6 Applying Dilemma Rule

In classical propositional logic, a proposition is either true or false (but not both). An attempt to prove (or check) a propositional formula can hence *branch* over the truth of a proposition in it. For example, to prove a formula p which contains propositions $p_1, p_2 ... p_n$, we assume that p_i is either true or false, and attempt to prove the formula for each assignment. Doing this leads to the proof of the formula branching into two separate proofs, which are called as branches of the proof. If we do this for all p_i, starting from 1 to n, then the entire proof tree would like this:



where, at depth i of the tree, the proof branches over proposition p_i, and each assignment in the node of tree represents an assumption over the value of a proposition.

Stålmarck's proof procedure [11] is a method to prove propositional formulas by applying various transformation rules. One such rule of interest to us is called the Dilemma rule. It states the following:

1. If one branch of the proof leads to a contradiction, then the result is the outcome is the other branch

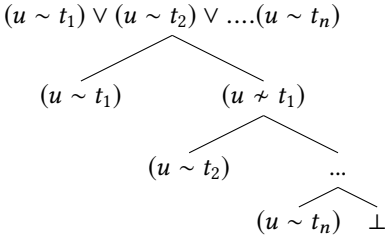
2. If neither branch leads to contradiction, then the result is an intersection of truth assignments in both branches

Informally, it simply states that if a proof of a formula branches over the truth of a proposition in it, then the intersection of information gained from both branches must be true.

Now, recall the definition of a duc predicate: a duc predicate $\{duc, u, [t_1, t_2..., t_n]\}$ asserts that the type u eventually unifies with any of the types from the list of types $[t_1, t_2...t_n]$. In formal logic, the duc predicate— which is essentially a *nullary predicate* or a proposition—can be expressed as a propositional formula:

$$(u \sim t_1) \vee (u \sim t_2) \vee ....(u \sim t_n)$$

where $u \sim t_i$ specifies that u unifies with t_i. Now, if we try to prove this formula by branching, the proof tree would look as follows:



where $(u \nsim t_i)$ denotes that u does not unify with t_i. The node $(u \sim t_i)$ is a leaf in the proof tree (i.e., there is no sub-tree under the node) because if $(u \sim t_i)$ holds, then it is already a valid solution for the entire formula. The rightmost leaf is a contradiction (denoted by $\perp$) because there is no valid solution to the formula under the assumption that $(u \nsim t_i)$ for all t_i.

If a unification $u \sim t_i$ in the leaf of a proof tree succeeds, then we have a substitution σ_i. However, if it fails, then we have reached a contradiction. Now, applying Dilemma rule to this proof tree gives us the following rules:

1. If all branches lead to a contradiction, then none of the candidate types are unifiable, and we have a type error
2. If not all branches lead to contradiction, then the result is the *intersection of the substitutions* which arise from the successful unifications

Let's look at an example of applying these rules. Consider the (problematic) inferred type from the previous section again:

$$getReason/1 :: (A, B) \rightarrow C \sim \{(pid(), B) \rightarrow cl(B), (pid(), B) \rightarrow sr(B)\} \Rightarrow (C) \rightarrow B$$

The duc predicate here can be expressed as:

$$\{duc, (A, B) \rightarrow C, [(pid(), B) \rightarrow cl(B), (pid(), B) \rightarrow sr(B)]\}$$

which corresponds to the propositional formula:

$$(A, B) \rightarrow C \sim (pid(), B) \rightarrow cl(B) \vee (A, B) \rightarrow C \sim (pid(), B) \rightarrow sr(B)$$

The successful unifications in the proof tree of this formula yield the following substitutions:

$$\{A \mapsto pid(), C \mapsto cl(B)\}$$
$$\{A \mapsto pid(), C \mapsto sr(B)\}$$

By applying the Dilemma rule here, we get that the intersection of both the substitutions $\{A \mapsto pid()\}$ must always hold. This substitution is then applied to the inferred type to yield the type:

$$getReason/1 :: (pid(), B) \rightarrow C \sim \{(pid(), B) \rightarrow cl(B), (pid(), B) \rightarrow sr(B)\} \Rightarrow (C) \rightarrow B$$

which can be further simplified to the following (as the arguments are always the same):

$$getReason/1 :: C \sim \{cl(B), sr(B)\} \Rightarrow (C) \rightarrow B$$

The resulting type is the expected type, and this process is the essence of applying the Dilemma rule to extract type information from a duc predicate.

Here, we have illustrated the extraction of type information from a single duc predicate and the simplification of the inferred type constraint. In the presence of multiple duc predicates, say $d_1, d_2, ..d_n$, the propositional formula at the root of the proof tree is a conjunction of all the propositional formulas of the corresponding duc predicates: $d_1 \wedge d_2 \wedge ..d_n$. To solve this formula, we first solve d_1 to yield a substitution γ_1. This substitution is then applied to d_2 and then solved to yield a substitution γ_2 and so on until d_n . The final resulting substitution is $\gamma_n \circ ... \gamma_2 \circ \gamma_1$. The main idea here is to compose all the substitutions as all the propositions must be true for the formula to be true.

3.7 Records

Records are treated as a special case of ADTs. The type system generates an ADT definition for every record definition in a module and its usage is handled in a special way. For the following record:

```
-record(person,{
    name  :: [char()],
    age   :: integer(),
    id  }).
```

the type system generates the following ADT:

```
-type person(A) ::
    {person,[char()],integer(),A}
```

The type of the generated ADT and its (only) constructor are both given the same name as the record. The types of the arguments to the constructor are the types of the fields in the

record. If the type of the field is specified, as in the definition of `name` and `age`, the specified type is used as the type of the corresponding argument in the constructor. Otherwise, the argument is assigned a type variable and the ADT type is parametrized over this type variable. Since the type of `id` has not been specified in the above record, the third argument of the constructor is assigned a type variable A and the type *person* is parametrized as $person(A)$.

When a record is created, the type of a given value (*available type*) is unified with the type of the constructor argument corresponding to the field (*expected type*) . For example, in the expression

```
#person{name="Nachi",age=25,id="c1"}
```

the available type of value of `id` is $[char()]$ and the expected type is A. When A is unified with $[char()]$, the resulting substitution instantiates A with $[char()]$, hence specializing the type $person(A)$ to $person([char()])$—which is the type of the above expression.

During creation, if the value of a field is not specified, then the type of the default value is used as the available type. However, if a default value is also not specified, then the available type is *undefined*. This is because, the value must be the atom `undefined`, and in our type system, the atom `undefined` is assigned the type *undefined*. For example, the expression

```
#person{name="Nachi",age=25}
```

has the type $person(undefined)$. In light of this treatment, if the type of a field has been declared and a default value has not been specified, then its value must be provided when the record is created. Otherwise, the expected type will fail to unify with the available type *undefined*, and a type error will be reported.

Note that the default value is used by the type checker to determine the available type, and not the expected type. This means, to assign a concrete type to a field, the type of the field must be declared in the record definition irrespective of whether a default value has been provided.

A record access returns a specific field, and hence the return type of a record access must be the type of the field. To achieve this, we return the type of the argument corresponding to the field in the constructor of the record type. For example, the type of the record access `Rec#person.age` is the type of the second argument to the constructor, i.e., $integer()$.

A record update returns a new record by changing the value of one or more fields in the original record. If the type of a field has been declared in the record definition, then the updated value must be of the same type as the declared type. Otherwise, the updated value maybe of a different type. For example, consider the following record update:

```
updateId(Rec,ID) ->
    Rec#person{id=ID}
```

Here `ID` may be of a different type from that of `Rec#person.id`. Since we want to allow the change in type of the field `id`, this function is assigned the type

$$updateId/2 :: (person(A), B) \rightarrow person(B)$$

This is achieved by inferring the type of `Rec` and replacing the type of the updated field with the inferred type of the new value. In this case, the inferred type of `Rec` is $person(A)$, the inferred type of `ID` is B, and the result type of the updated expression is $person(B)$ (where A has been replaced by B).

4 Partial Evaluation

Type inference alone isn't enough to type Erlang. For example, consider the `list_to_tuple/1` function. What should its return type be? For a list of size n given as an argument, it returns a tuple of size n—which has a different type for each value of n.

However, if the values of the arguments to a function which exhibits dynamic type behaviour are available at compile time (called *static* values), then it maybe possible to transform (or *reduce*) the function application to an expression where the dynamic behaviour has been removed. For example, consider the application `list_to_tuple([1,2,3])`. It is evident that it can be reduced to `{1,2,3}` at compile time. The reduced expression does not exhibit any dynamic type behaviour, and can be easily assigned a type in our type system.

Partial evaluation reduces an expression by pre-computing the static parts. The reduced expression is often simpler, and the dynamic type behaviour may be removed—hence creating an opportunity for type inference.

Unlike the previous example, partial evaluation is not limited to static values alone. It can also reduce expressions by using the construction of a value. For example, the expression on the left in is reduced to the one on the right:

Original	After PE
`T = {F(X),G(X)},`	`T1 = F(X),`
`element(1,T).`	`T2 = G(X), T1.`

Although the value of `T` is not known in the original expression, its structure is known to be a 2-element tuple by construction. This information is exploited by the partial evaluator to reduce `element(1,T)` to `T1` (the result of `F(X)`). Notice that the dynamic behaviour has been removed in the reduced expression. As a result, the type system correctly assigns it the type:

$$struct\_pe/2 :: ((A) \rightarrow B, (A) \rightarrow C, A) \rightarrow B$$

There are also other advantages to partial evaluation besides evaluation of function applications. In some cases, the partial evaluator cannot evaluate an application, because all the arguments may not be available statically. In this case, the static arguments are reduced, and the others are left intact in the function call. Even this behaviour helps the type checker

in many cases. For example, if a call to `is_function/2` is reduced to the expression `is_function(F,3)`, where the value of `F` is unknown, then the type checker unifies the inferred type of `F` with the fresh type $(A, B, C) \rightarrow D$, which enables it to assert that `F` must be a function type with arity 3 and some return type. As a result, the expression `is_function(F,3)` is type checked successfully even though partial evaluation does not reduce it further.

5 Results

We have implemented a prototype of this type system as a parse transform in Erlang, which takes the abstract syntax tree (AST) of a subject Erlang program as input and does type checking as a side-effect. If type checking succeeds, the parse transform writes a module interface file with the types of top level functions and returns the AST (possibly modified by partial evaluation), otherwise it throws a type error using `erlang:error/2`—causing the compilation to crash. The link to the implementation can be found in section A.1. In this section, we show some preliminary results of running the type checker on several hundreds of lines of code.

5.1 Evaluation

We evaluate our type checker by running it against many example functions (some shown in earlier sections) and some small Erlang libraries. The selection includes a couple of OTP libraries, and a library which implements a fault tolerant distributed resource pool. The current version of the type checker does not support typing of modules, and the libraries have been chosen such that they contain only one module. Note that, however, for the purposes of this evaluation, remote function calls to certain built-in functions are allowed because they have been pre-assigned a type in the environment.

The following table shows the number of lines of code (LOC) in the library, the number of lines modified (to make the type checker accept it) and the total compilation time of the module (which includes type checking, writing module interface and compilation).

Library	LOC	LOC +/-	Time
OTP/orddict	179	2	0.450s
OTP/ordsets	150	0	0.396s
ft_worker_pool	73	1	0.382s

The LOC modification is caused by adding an ADT definition and wrapping a value in the constructor of the ADT. For example, the following definition had to be added to type check the `take/2` function in OTP/orddict:

```
-type maybe(A) :: error | {ok,A}.
```

Moreover, the type checker has also been applied to a 500 LOC test-suite which contains many corner cases for good and bad programs (the test-suite can be found along with the implementation by following the link in section A.1).

Our type checker catches errors that are easily missed by Dialyzer. For example, the ill-typed function `find/0` in section 1 is accepted by Dialyer, but is rejected by our type checker.

5.2 Error Messages

During our evaluation, we found that the error messages are concise, fairly easy to comprehend and often contain enough information to identify the issue. The most common errors reported by the type system are of three kinds: 1) unification errors, 2) class predicate solving errors, and 3) duc predicate solving errors.

Unification errors report that the types which cannot be unified and the lines they originate from. For example, the error *Cannot unify float() (line 1) with string() (line 2)* specifies that the type checker attempted to unify a `float()` on line 1 with a `string()` on line 2.

Class predicate errors report an attempt to solve an invalid class predicate along with the line number. For example, the error *Invalid instance type string() (line 83) for class Num* specifies that a value of type `string()` has been treated as a number on line 83.

Duc predicate errors are also fairly comprehensive and hide much of the underlying constraint solving. For example, when all branches of the proof tree in constraint solving lead to a contradiction, the type checker reports the error: *Unable to find matching overloaded constructors on lines [7,8]*, which specifies that neither of overloaded constructors on lines 7 and 8 match the expected type.

6 Limitations

Erlang allows the programmer to write functions which operate on arbitrarily nested data structures. For example, it allows the programmer to write the following function (taken from the `lists` library):

```
flatlength([H|T], L) when is_list(H) ->
    flatlength(H, flatlength(T, L));
flatlength([_|T], L) ->
    flatlength(T, L + 1);
flatlength([], L) -> L.
```

where the first argument to `flatlength/2` can be an arbitrarily nested list. This function, which looks perfectly reasonable to an Erlang programmer, is rejected by the type system, because it fails the *occurs check*. The problem here is that the type of the first argument is infinitely recursive (since the list may be arbitrarily nested), and the type system is unable to assign it a finite type.

Many existing Erlang and OTP libraries have been written using Dialyzer's type system. This means that these programs rely heavily on the flexibilities of subtyping principles. One example of this is the use of *union types*. A function, for example, can return values of different types, provided a union of all the returned types has been defined as a valid

type. In our type system, however, this can be achieved only by wrapping each returned value in constructors of the same ADT. As a result, this would need modification of such functions in order to pass our type checker.

The current type system does not type check concurrency. The types assigned to messaging primitives are too simple and this can easily lead to uncaught type errors. For example, consider this function:

```
foo() ->
    receive X -> X end.
```

This function type checks successfully and is assigned the type $() \rightarrow A$. Now consider the expression `foo() + 1.0`. It is assigned the type $float()$ by the type system since A is unified with $float()$. But what happens if `foo` receives (and hence returns) a $boolean()$? The execution of the expression leads to runtime error!

7 Related Work

The earliest notable effort to type Erlang is the subtyping system by Marlow and Wadler [8]. Their system types a subset of Erlang by solving unification constraints of the form $U \subseteq V$, which denotes that the type U is a subtype of type V. In contrast, our type system solves unification constraints of the form $U = V$. Although their work increased type awareness among programmers, their system was not adopted as type inference was slow and the inferred types were large and complex.

Agda [1] is a dependently typed functional programming language which allows the use of overloaded constructors. A key difference, however, is that in Agda the user must supply type information for all top level function definitions and most sub-expressions. In such a type rich environment, the type checker takes advantage of user provided types to disambiguate between overloaded constructors. Our type system, on the other hand, has no support for supplying type signatures and is purely based on type inference. Moreover, in Agda, if the type checker cannot disambiguate between two constructors of the same type, it throws a type error. Our type system defers this disambiguation (using type constraints) for later until more information is available, and hence allows the constructors to be applicable for both types.

An alternative to partial evaluation to type functions such as `element/2`, `is_function/2` etc is to use dependent types. This is because the type of the function depends on the value of the arguments. However, type inference for dependent types in general has been shown to be undecidable [3], and we are forced to avoid this path to retain type inference in our type system.

8 Future Work

Our work is far from complete. The type checker works for single module Erlang programs, but lacks features which make it practical for large scale Erlang applications. Support for modules and exceptions in the type system is an important requirement. Typing concurrency is also an interesting goal for future work. Yet another avenue for future work is better integration of partial evaluation and type inference. Currently, partial evaluation is simply a pre-pass to type inference, and information which arises at one stage cannot be exploited in the other.

A Appendix

A.1 Source Code

The source code of the type checker prototype described in this paper can be found at https://github.com/nachivpn/mt

A.2 Loss of Type Information in Duc Constraints

Consider the following ill-typed function (recall the definition of `getReason/1` from section 3.5):

```
foo() ->
    1.0 = getReason({'EXIT',self(),true}).
```

Instead of rejecting `foo()`, the type checker assigns the type:

$$foo/0 :: (A, float()) \rightarrow B \sim \{(pid(), float()) \rightarrow cl(float()),$$
$$(pid(), float()) \rightarrow sr(float())\}$$
$$B \sim \{cl(boolean()), sr(boolean())\} \Rightarrow () \rightarrow float$$

Applying Stålmarck's method helps us catch this error.

References

[1] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[2] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 207–212.

[3] Gilles Dowek. 1993. The undecidability of typability in the lambda-pi-calculus. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 139–145.

[4] Mark P Jones. 1999. Typing haskell in haskell. In *Haskell workshop*, Vol. 7.

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

[6] Tobias Lindahl and Konstantinos Sagonas. 2005. Typer: a type annotator of erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*. ACM, 17–25.

[7] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, 167–178.

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

[9] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

[10] Didier Rémy. 2002. Using, understanding, and unraveling the OCaml language from practice to theory and vice versa. In *Applied Semantics*. Springer, 413–536.

[11] Mary Sheeran and Gunnar Stålmarck. 1998. A tutorial on Stålmarck's proof procedure for propositional logic. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 82–99.