CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Typing the Untypeable in Erlang

A static type system for Erlang using Partial Evaluation

Master's thesis in Algorithms, Logic and Languages

Nachiappan Valliappan

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Typing the Untypeable in Erlang

A static type system for Erlang using Partial Evaluation

Nachiappan Valliappan

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Typing the Untypeable in Erlang
A static type system for Erlang using Partial Evaluation
Nachiappan Valliappan

Supervisor: John Hughes, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Erlang is a dynamically typed concurrent functional programming language popular for its use in distributed applications. Being a dynamically typed language by design, the Erlang compiler allows the successful compilation and execution of many programs which would be rejected by a type checker of a statically typed language. This idiosyncrasy of Erlang makes it difficult to retrofit static type checking technology onto the language. In this thesis, we develop a static type system suitable for Erlang using a program specialization technique called partial evaluation.

Keywords: Erlang, Type Inference, Hindley-Milner, Partial evaluation

Acknowledgements

This thesis is based on John's idea to combine type inference and partial evaluation to type Erlang. Many good ideas in this thesis were developed in close collaboration with John. I would like to thank him for being a great guide along the way.

I would also like to thank my parents for being of tremendous support during my education at Chalmers.

Nachiappan Valliappan, Gothenburg, June 2018

Contents

1 Introduction

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 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.

A prominent downside of Erlang, however, 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. 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. A static type system can help catch such errors well in advance, and hence avoid the tedious task of debugging.

1.1 Dialyzer and friends

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.

Dialyzer is a static analysis tool which helps identify type errors in Erlang programs. 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* [LS06]. A key property of such a type system is that it does not produce false positives. The type checker takes an optimistic approach and assumes that a program is well-typed unless it can be prove otherwise. Dialyzer does not require the programmer to supply any type information, a good design choice for checking legacy code.

On the other hand, in stark contrast, a type checker of a statically typed language requires the *programmer* to prove that a program is well-typed. The programmer must fix all type errors which arise from type checking before it can be compiled successfully. Languages such as Java and C++ achieve this by making the programmer specify explicit type annotations. But type annotations can get tedious and often clutter the code. Languages such as Haskell and ML implement *type inference* to free the programmer from having to specify excessive type annotations. The type inference implemented by Haskell and ML are both based on a type system popular for this purpose: the Hindley-Milner type system.

The success of Hindley-Milner based type systems for typing functional languages makes us wonder whether it can be used to type Erlang. Most notable efforts to typing Erlang [LS05][LS06][MW97] reject Hindley-Milner in favour of subtyping. The most commonly cited reason to favour subtyping over the constructor based implementations of Hindley-Milner has been to avoid the restriction that constructors must exclusively belong to one type. This restriction has been considered an inherent property of a type system based on Hindley-Milner. However, we show that this need not necessarily be the case. Moreover, 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 beats the whole purpose of a type checker. The Dialyzer (which uses subtyping) appears to be much slower in practice when compared to a Haskell/ML type checker.

In this thesis, we present a purely inference based type system suitable for Erlang inspired by Haskell's adoption of Hindley-Milner. We show that it is possible to type overloaded data constructors and yet retain the type inference properties of a Hindley-Milner system. To achieve this, we adopt a form of ad-hoc polymorphism similar to Haskell's type class system. This leads to much faster type checking and comprehensible type signatures and errors. Our work helps uncover the subset of Erlang which can be typed using a Hindley-Milner based type system.

1.2 Beyond Type Inference: Partial Evaluation

Type inference alone isn't enough to type Erlang. Since Erlang was not designed with a type system in mind, it allows unrestricted programming of values. For example, a function can return different types values on different inputs. A Hindley-Milner type inferencer will simply reject such functions as it cannot assign a single type to the function. Rejecting such programs can could lead to a large portion of the Erlang/OTP code base being rejected. Hence, we must find a way to type such programs while still maintaining rigorous type correctness.

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. Consider this Erlang program:

```
tuplize([]) -> {};
tuplize([X|Xs]) -> {X,tuplize(Xs)}.
```

The function `tuplize` converts a given list to a tuple by replacing every occurrence of a cons constructor with the constructor of a two tuple. That is, `tuplize([1,2,3]) = {1,{2,{3,{}}}}`. The type of the input to this function is simply a list, but the type of the output depends on the length of the list which is not known until runtime. Hence, this function cannot be assigned a type a compile time. However, the application `tuplize([1,2,3])` can be simplified to `{1,{2,{3,{}}}}`, which can be assigned a type at compile time. The idea that partial evaluation employs is the simplification of programs when the input to this function is available at compile time. This simplification can be used to expand coverage of type inference.

Partial evaluation is not limited to just computations involving constant values (also known as *constant folding*), it is also capable of performing more sophisticated specialization. For example, consider the following functions:

```
server(get_sum,[X,Y]) -> X + Y;
server(get_append,[X,Y]) -> X ++ Y;
server(get_id,[X] -> X.

client(Input) -> server(get_sum,Input).
```

The `server` function pattern matches over the input arguments and returns different values depending on the matched pattern. Notice in the body of `client`, the function call to `server` contains only one known argument. Partial evaluation of this program yields the following residual program:

```
simplified_server([X,Y]) -> X + Y.

client(Input) -> simplified_server(Input).
```

In the residual program, the information known from the first parameter is used to generate the specialized `simplified_server` function. Also, unlike the original `server` function, the body of `simplified_server` is much simpler and its type can be inferred easily.

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 illustrate the types inferred by our type checker for various functions.

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:

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

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 lists of type $[A]$, where A is any type, 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 $([boolean()], [boolean()]) \to [boolean()]$.

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:

$$(+) :: \mathit{Num}\ A \Rightarrow (A, A) \rightarrow A$$

where $\mathit{Num}\ A$ is a constraint on the type A that asserts that A must be a numeric type (that is, an integer or a float). 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 $\mathit{float}()$, and the type constraint is specialized to $\mathit{Num}\ \mathit{float}()$. The type checker knows that $\mathit{Num}\ \mathit{float}()$ is solvable, and hence accepts the expression as well-typed. If A is instantiated with an non-numeric type, such as $\mathit{boolean}()$, the type checker reports a type error as it cannot solve the constraint $\mathit{Num}\ \mathit{boolean}()$.

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:

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

This type states quite correctly 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:

$$\mathit{div} :: (\mathit{integer}(), \mathit{integer}()) \rightarrow \mathit{integer}()$$

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

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

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:

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

2.3 Algebraic data types

ADTs are used to define new types using user defined constructors. These constructors 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)}.
```

`nil` is a nullary constructor which constructs a tree of type `tree(A)`, and `node` is a three argument constructor which constructs a tree of type `tree(A)` when given a value of type `A` and two trees of type `tree(A)`.

In a function body, the type checker considers tuples where the first element is an atom to be a constructor application. For nullary constructors, the type checker also allows just the atom to be specified. Now, let's look at an example of this usage:

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

The `findNode` function pattern matches on a tree 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. Also note that, in this case, the type checker would reject the `findNode` function since the second argument would have different types in the first and second clause.

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:

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

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, its return type gets specialized to a list, and when it's called to return a tree, its return type gets specialized to a tree. For example, in the expression `flattenTree(empty())`, since `flattenTree` expects a tree argument, D is instantiated with the type $tree(A)$, and the type of the expression is inferred 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:

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

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:

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

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. This is because the clause bodies of a `receive` expression are not expected to be of the same type unless their return value is used. In this case, the value returned by the `receive` expression is discarded, and hence the bodies need not be of the same type.

3
Typing Erlang

The Hindley-Milner type system in [DM82] was devised for a small expression language which supports functions and let-expressions. A notable property of the type system is its ability to infer the most general type of a given term in the language (the *principal type*) without requiring any annotations. This property makes it a particularly appealing choice for retrofitting a type system to Erlang.

However, the original Hindley-Milner type system in itself 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 (operators) 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 propose 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. The main idea is to allow a constructor to belong to multiple types by using a type constraint. Like type class constraints, these constraints are then later specialized as more information is available. For example, in the `flattenTree` function, when the type checker encounters `nil` as the argument in the first clause, its type is recorded to be a tree or a list in a type constraint. But, when it encounters the argument of type tree in the second clause, this type constraint is specialized to a tree.

We have implemented 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 returns the AST, otherwise it throws a type error using `erlang:error/2`—causing the compilation to crash. In this chapter, we are concerned with the implementation details of the type system.

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 type variables itself until more information is available, i.e, it is also a valid type. The types in Hindley-Milner can be described using the grammar:

```
<type> ::= <base>
```

```
    | <tvar>
    | (<type>,..,<type>) → <type>
```

where ⟨*base*⟩ represents a base type, ⟨*tvar*⟩ represents a type variable, and (⟨*type*⟩, .., ⟨*type*⟩) → ⟨*type*⟩ represents a function type.

During type inference, a type is considered to be a partially known type 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 $(X) \to X$ and $(boolean()) \to Y$ yields the *substitution* $\{X \mapsto boolean(), Y \mapsto boolean()\}$, which when *applied* to the types equalizes them. 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.

A substitution is defined 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 in the previous example is applied to either of the types, it yields the type $(boolean()) \to boolean()$. Note that the notion of applying a substitution is the standard one which replaces only free variables and accounts for name capture.

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, $\sigma(t_1) = \sigma(t_2)$. The unifying substitution is also called the unifier. Note that there may be several (or no) unifiers for any two given types. The computation of the *most general unifier* (mgu) is an essential part of type inference. A substitution σ is said to be the most general unifier of two types if for every other unifier σ' of the types, there exists γ such that $\sigma' = \gamma \circ \sigma$, where $(\gamma \circ \sigma)t = \gamma(\sigma(t))$. That is, σ is the mgu if all other unifiers can be expressed in terms of it.

An important feature of the Hindley-Milner type system which allows generic programming is polymorphism. A function is said to be polymorphic if it can be used in different contexts with different types. To achieve this, the polymorphic function is assigned a generic *type schema*. When the function is applied in a certain context, the type schema is instantiated to yield a unique type of the function, which is then specialized using the context specific information. The type schema acts as a representation of all the valid types that the function can be assigned. It can be described using the grammar:

```
<schema> ::= <type>
    | ∀ <tvar>.<schema>
```

To generate a type schema, type inference employs a technique called generalization. Generalization essentially converts the inferred type of a function to a type schema. An example of generalization is the conversion of $(T) \to T$ to $\forall T.(T) \to T$.

To appreciate the need for generalization, consider the following example:

```
id(X) -> X.
```

Suppose that `id` is assigned the type $T \to T$. Now, consider its applications in the following function:

```
foo(X) ->
        id(5.0),
        id(true).
```

On encountering the first application of `id`, suppose that type inference unifies the argument type T with $float()$. This instantiates the type variable T with the $float()$, and as a result, specializes the type of `id` to $(float(), float()) \rightarrow float()$. But, note how this becomes a problem for the second application of `id`. `true`, which is an argument of type $boolean()$, cannot be applied to a function of type $(float(), float()) \rightarrow float()$, and this occurrence would lead to a type error.

To avoid this problem, `id` is assigned the generalized type schema $\forall T.(T) \rightarrow T$. Now, when type inference encounters a term with a type schema, it replaces all the bound variables in the type with fresh type variables to yield a type, which is used as the type of the term. In the above example, when the first application of `id` is encountered, the bound type variable T is replaced with some type variable, say P, to yield $P \rightarrow P$, and the second application would be instantiated with a (different) fresh type variable Q to yield $Q \rightarrow Q$. As a result, P would be instantiated with $float()$ and Q would be instantiated with $boolean()$, hence avoiding the type error.

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 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 (which assigns types to free variables in a function's body) it's 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's also enforced to be the same as the inferred type.

The case of mutually recursive functions is a little more complex. OCaml [Rém02] 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 [Jon99].

Another requirement beyond Hindley-Milner to type Erlang is the overloading of operators and functions. For this, we implement a simple type class system, which is discussed in the next section.

3.3 Type classes

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 $Num\ A \Rightarrow ...$ 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. To add type constraints to a type schema, we modify the type schema grammar from Hindley-Milner to:

```
<schema> ::= <type>
    | ∀ <tvar>. [<constraint>]. <schema>

<constraint> ::= <class>.<tvar>
```

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 the type $U \rightarrow U$ (for some fresh variable U) and the type constraint $Num\ U$.

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. The difference between a predicate and a class constraint is that the type i of a predicate need not be a type variable. Moreover, class predicates are not the only predicates, as we will see later.

Class predicates are implemented as a three element tuple, where the first field is the atom `class`, the second field is a string representing the class name, and the third field is an Erlang term representing the instance type. For example, the Erlang term $\{class, "Num", integer\}$ is a predicate asserts that the type $integer()$ is an instance of the class Num.

To understand the collection of type class constraints, 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 $Num\ B$. These type constraints are collected as the following predicates:

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

where A is the expected type of the first operand and B is the expected type of the second operand.

The step which happens right after type inference of a function body in a pure Hindley-Milner implementation is generalization. In the presence of type class constraints, however, the collected predicates must be solved before the type is generalized. Solving the predicates means checking if predicates are satisfiable using the premise. It is defined as follows:

```
solveClassPs(Premise,ToSolve) ->
    lists:filter(fun(Predicate) ->
        not lists:member(Predicate,Premise)
    end, ToSolve)
```

The predicate solver for class constraints is a function which accepts the premise (which is implemented as a list of predicates that are known to be true) and a list of predicates to solve as arguments, and returns a list of unsolvable predicates. If the result is an empty list, then the type is generalized without any type constraints. If the result is non-empty and contains a predicate where the instance type is not a type variable, then an "Invalid instance" type error is reported. However, if the result contains predicates with type variables, it means that these predicates cannot be solved at this time, and hence must be preserved in the type for later. Hence, these predicates are generalized along with the type of the function, leading to a type class constraint in the type.

Now, getting back to our `average` example, suppose that the inferred type of `sum(Xs)` is $Num\ T \Rightarrow T$ for some fresh 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:

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

Now, applying `solveClassPs` to the specialized predicates removes the second predicate (as it follows from the premise) resulting in the unsolved predicate:

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

This predicate is then 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:

```
<type> ::= ...
    | <constructor> [<type>]
```

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 $(A_1, ..., A_n) \to V$ (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 list of (non-empty) types $[T_1, ..., T_n]$, the type $(A_1, ..., A_n) \to V$ (discussed in the previous section) may unify with more than one of these types (as the types $(A_1, ..., A_n)$ are only partially known). Since we cannot always make a decision on exactly one type of $[T_1, ..., T_n]$ at this time, we *defer* this unification by generating a new kind of predicate called the deferred unification constraint (duc) predicate:

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

which asserts that the type $(A_1, ..., A_n)$ unifies with any 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 any specializing information which arises during type inference.

Solving a duc predicate later means performing the deferred unification. This is possible only if the list of candidate types is exactly one. Checking if a duc predicate is solvable is implemented as:

```
solvableDucP({duc,_,[_]}) -> true;
solvableDucP(_) -> false.
```

Now, recollect the `findNode` example from earlier:

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

The first clause has an overloaded constructor `nil` as an argument, this generates the predicate:

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

where $() \to V$ is the inferred type of the 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 the V to $tree(C)$ (the return type of `node`, for some type variable C), hence specializing the predicate to

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

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 $(A, tree(A)) \rightarrow boolean()$.

However, if 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:

```
<constraint> ::= ...
     | <type> ~ [<type>]
```

A constraint $T \sim [T_1, T_2, ..T_n]$ specifies that the type T unifies with any 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:

```
<schema> ::= ...
     | <constraint> <schema>
```

For example, in the case of `empty()`, due the 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.

However, unlike the case of class predicates, simply retaining the unsolved duc predicates as type constraints can lead to long and unreadable types, and even type errors being missed. To understand this problem, consider this example:

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

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

`{'EXIT', pid(), R}` the type which 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:

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

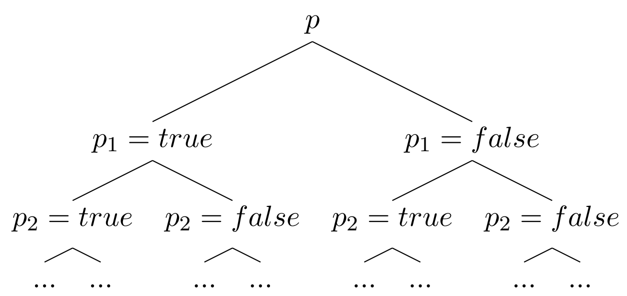
The reason for this is that the generated predicates that cannot be solved have simply been generalized as type constraints. Although it's possible for us to see from the generated type constraints that A always unifies with $pid()$ and B always unifies with B, this information has not been exploited 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 the Appendix (A.5).

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 [SS98] 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:

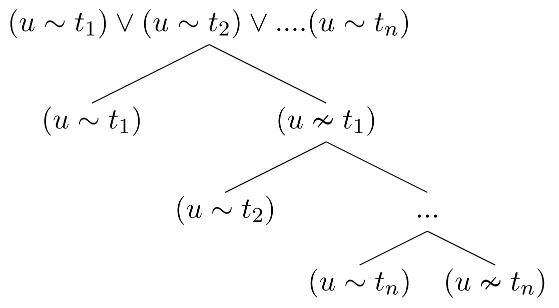
1. If one branch of the proof leads to a contradiction, then the result is the outcome of the other branch
2. If neither branch leads to contradiction, then the result is an intersection of truth assignments in both branches

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:

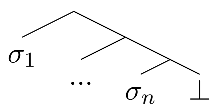
$$(u \sim t_1) \vee (u \sim t_2) \vee ....(u \sim t_n)$$

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 $u \nsim t_i$ means u does not unify with t_i. There is no sub-tree under $(u \sim t_i)$ because if $(u \sim t_i)$ holds, then it is already a valid solution for the entire formula.

A unification $u \sim t_i$ can be performed to yield a substitution σ_i. In the case that all unifications succeed, then we have the following proof tree:



where the rightmost leaf is a contradiction since $(u \nsim t_i)$ for all i. However, if unification fails in a branch, 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

But, what is the intersection of substitutions? Suppose that we treat a substitution σ_i as a propositional formula which is a conjunction of propositions represented by $\langle var \mapsto type \rangle$. For example, a substitution $\{X \mapsto boolean(), Y \mapsto float()\}$ would be a propositional formula of the form $\langle X \mapsto boolean()\rangle \wedge \langle Y \mapsto float()\rangle$. Then we can define the intersection of substitutions as the common propositions in all substitutions.

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

$$\begin{gathered} getReason/1 :: (A, B) \to C \sim \\ [(pid(), B) \to cl(B), (pid(), B) \to sr(B)] \\ \Rightarrow (C) \to B \end{gathered}$$

The duc predicate here can be expressed as:

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

which corresponds to the propositional formula:

$$(A, B) \to C \sim (pid(), B) \to cl(B) \vee (A, B) \to C \sim (pid(), B) \to sr(B)]\}$$

These can be re-written as:

$$(\langle A \mapsto pid()\rangle \wedge \langle C \mapsto cl(B)\rangle) \vee (\langle A \mapsto pid()\rangle \wedge \langle C \mapsto sr(B)\rangle)$$

By applying the Dilemma rule here, we compute the intersection of both the substitution propositions to get that the following proposition is always true:

$$\langle A \mapsto pid()\rangle$$

This proposition can be converted back to substitution and applied to the above type to yield:

$$\begin{gathered} getReason/1 :: (pid(), B) \to C \sim \\ [(pid(), B) \to cl(B), (pid(), B) \to sr(B)] \\ \Rightarrow (C) \to B \end{gathered}$$

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

$$getReason/1 :: C \sim [cl(B), sr(B)] \Rightarrow (C) \to B$$

This is the essence of applying the Dilemma rule to extract type information from the duc predicates.

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 compute the substitution of 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.

4 Partial Evaluation

In this chapter, we discuss the implementation of a simple partial evaluator for Erlang.

4.1 Overview of Partial Evaluation

An evaluation of a function accepts some inputs and produces an output. The required inputs must be made available prior to evaluation of the function. Typically, the availability of all the inputs and the evaluation of the function happens only at runtime. However, one can often find values for some of these inputs at compile time. The availability of these inputs can be used to pre-compute parts of the function body at compile time. This is the essence of partial evaluation.

Consider the following Erlang function which computes the `N`th power of a number `X`:

```
power(0,_) -> 1;
power(N,X) -> X * power(N-1,X).
```

The variables `N` and `X` are the required inputs for this function. The variables whose values are available at the time of partial evaluation are called *static* variables. The variables whose values are not available until runtime are called *dynamic* variables. In the function call, `power(3,2)`, `N` and `X` are both static variables with values `N=3` and `X=2`. Partial evaluation of `power(3,2)` replaces the function call with its pre-computed value 8.

Now, suppose `X` is dynamic and `N` is static with value `N=3`, as in `power(3,X)`. In this case, we cannot pre-compute the function application as the value of `X` is not known at this time. However, we can use the value of `N` to *reduce* the application to a simpler form. By definition of the function, we get that `power(3,X)` equals `X * power (3-1 ,X)`, which can be further simplified to `X * power (2,X)` by constant folding. In a similar fashion, we can reduce `power (2,X)` until we reach the base case `power (0,X)`, which equals `1`. Now by replacing the function applications with their reduced expressions, we get that `power(3,X)` equals `X*X*X*1`. The resulting expression is called the residual expression. Computing such a residual expression is the goal of partial evaluation.

Partial evaluation *specializes* a program to another program by removing the computations involving static variables. The resulting specialized program is visibly simpler (for example, function calls have been removed) and executes faster than the original program since the static parts have been pre-computed. Partial evaluation's ability to specialize programs also has an interesting application in automatic compiler generation. This was observed by Futamura in 1971 [Fut71], and hence called the Futamura projections. In this thesis, we are interested in exploring a novel idea of applying partial evaluation of programs to aid type inference.

4.2 Partial Evaluation for Erlang

Erlang is a call by value (CBV) language which, unlike a pure language, allows arbitrary side-effects to be performed during execution of a function. A partial evaluator for Erlang must ensure that CBV semantics is preserved and that side-effects are not re-ordered.

Another challenge is Erlang's pattern matching, which is quite sophisticated and can be tricky to get right. This section discusses the techniques used to solve these problems.

4.2.1 Setting up the basics

An Erlang program is a list of top level function definitions. As we will see later, we are only interested in partially evaluating some top level functions to aid type inference. Hence, the problem is essentially one of partially evaluating a top level function.

A top level function contains a number of function clauses, each of which have a list of arguments (which might be patterns), guards and a body. A body is a list of expressions. A variable in an expression is *bound* if it has been defined using pattern matching. Pattern matching can be done using `match`, `case`, `try` expressions or using function arguments. A bound variable may be static or dynamic. It is static if the value it is matched against is known at specialization time, and dynamic if the value is not known.

A core subroutine used by partial evaluation of a top level function is called reduce, and is implemented by a function `reduce/2` which partially evaluates a given expression using an *environment* and returns the evaluated expression and a new environment.

```
reduce(Expr,Env) ->
    ...
    {EvaluatedExpr,UpdatedEnv}
```

The environment is the state of partial evaluation, which contains things such as variable bindings, seen variables and function definitions. `reduce` returns the evaluated expression and a new environment as it may update the environment during partial evaluation (for example, reducing a match expression $P = Q$ may bring new variable bindings into scope).

4.2.2 Preserving program semantics

The case for variables in `reduce/2` is implemented by simply looking up the corresponding expression bound to the variable in the environment. If the variable has a value in the environment (because it's static), then the value is returned as the reduced expression. Otherwise, the variable is returned as it is (because it's dynamic).

Given this treatment of variables, it is important to ensure that an expression assigned to a variable in the environment is always a *value* in order to preserve call by value semantics. To understand the need for it, consider the function expression:

```
fun(Y) ->
    X = foo(Y),
    X + X
end
```

Here, if we simply assign the function call `foo(Y)` to `X` in the environment, then `X + X` would be reduced to `foo(Y) + foo(Y)` which would end up in `foo(Y)` being called twice. This not only does not preserve CBV semantics, but might also end up in side-effects being duplicated if `foo(Y)` does some side-effects.

One option is to limit these expressions to static values alone. However, a lot of specialization information can be lost in doing so. For example,

```
fun(X) ->
    Y = X
    Y
end
```

Here, `X` is dynamic. Evidently this function can be reduced to the identity function. But this cannot be achieved if we limit the expressions assigned to variables in the environment to static values.

A better solution is to treat variables as values. This is because variables are simply considered as references and duplicating them is harmless. Concretely, an expression is a value if the function `isValue/1` returns true for it (implementation for part of which is shown in figure 4.1). As discussed, the case of variable is defined as true. As a result, expressions such as `{X,Y}` are treated as values and this leads to much better specialization.

Erlang allows the expression which is pattern matched against to contain non-value expressions such as function calls. One option is to simply retain this expression as it is in the residual program and this would lead to the semantics being preserved. However, we can do much better than that. For example, consider the following function:

```
fst(A,B) ->
    X = {id(A),id(B)},
    case X of
        {P,Q} -> P
    end.
```

The expression `{id(A),id(B)}` returns false for `isValue`, and hence will not be assigned to `X`. As a result, this specialized version of this program will be the same as the original program. But is it evident that this function can be reduced to return the value of `id(A)`. To achieve this, we convert the expression `{id(A),id(B)}` to a value by replacing all function calls with variables as follows:

```
fst(A,B) ->
    X_1 = id(A),
    X_2 = id(B),
    X = {X_1,X_2},
    case X of
        {P,Q} -> P
    end.
```

Now, since `{X_1,X_2}` is a value, this function gets reduced to:

```
fst(A,B) ->
    X_1 = id(A),
    X_2 = id(B),
    X_1.
```

This conversion is implemented by a function `convertToValue` which accepts an expression as an argument and returns the expression (where all function calls are replaced with fresh variables) and defining bindings for the fresh variables in the expression.

4.2.3 Pattern matching

Pattern matching in performed in Erlang using match (=), `case`, `receive`, `try` or function arguments. In this section, we discuss partial evaluation of the match expression. The

```
isValue({float,_,_})      -> true;
isValue({integer,_,_})    -> true;
...
isValue({var,_,_})        -> true;
isValue({cons,_,H,T})     -> isValue(H) and isValue(T);
isValue({tuple,_,Es})     -> lists:all(fun isValue/1, Es);
isValue(_)                -> false.
```

Figure 4.1: Check if an AST node is a value

essence of pattern matching is the same in the other expressions and the implementation is analogous.

Consider the match expression $P = Q$. P is allowed to have unbound variables, and such variables are considered bound after the match expression. However, Q cannot contain any unbound variables. For example, `3 = X` is illegal if `X` is not already bound. The semantics of the $P = Q$ is as follows: if the value of P and Q are known, then do a runtime check to ensure that their values are equal. Otherwise, assign the variables in P to the corresponding values in Q, *and vice-versa.* The case that unbound variables cannot occur in Q but variables in Q can be assigned values in P applies because Q can contain dynamic variables. Recall that a dynamic variable is a bound variable. For example, the X in the expression `foo (X) -> 3 = X, X + 1.` is legal and is assigned the value 3 after the match expression.

A partial evaluator must implement the exact same thing, except with partially available values: if the value of P is known at specialization time, then assert that it matches Q, otherwise bind variables in P (Q) to the reduced values in Q (P). To achieve this two way instantiating of variables, we use *unification.* Unification, in the context of pattern matching, is very similar to unification in type inference. In type inference, unification takes two partially known types that must be equal and instantiates type variables in them so that it is indeed the case. Instead, in pattern matching, it takes two terms and instantiates term variables in them. For example, the unification of the terms $\{X, [Y]\}$ and $\{5.0, Z\}$ yields a substitution $\{X \mapsto 5.0, Z \mapsto [Y]\}$—which are the expected values of the corresponding variables in pattern matching.

To reduce a match expression $P = Q$, first we reduce P in the given Env to get a reduced P' and an updated Env'. This is because Erlang allows simple expressions which can be computed statically to occur on the left. Then, we reduce Q (in the same Env) to get Q' and an environment which is discarded (as new variables cannot be bound on the left). The reduced Q' is then converted to a value using `convertToValue` to yield Q''. The resulting defining bindings are collected to be returned along with the final reduced expression. Then, we unify P' and Q'', which—if successful—yields a substitution σ which maps variables in P' to their corresponding expressions in Q'' and variables in Q'' to corresponding expressions in P'. Note here that unification is more powerful than the expected semantics of the match expression in Erlang as the substitution may also instantiate an unbound variable on the right. To avoid this, we must ensure that the returned substitution only substitutes variables that are recorded as bound in the environment.

Finally, we update the environment Env' with the substitution σ (because a substitution is a list of assignments to a variables) to yield a new environment Env''—which is the returned environment. If P' is a variable and Q'' is a value, then the final reduced expression is block of expressions created using the accumulated defining bindings and the value Q'' at the end. Otherwise, the block has $P' = Q''$ at the end.

4.2.4 Branching

Branching can be done in Erlang using `case`, `if`, `receive`, `try` and function clauses. In this section, we discuss the implementation of reducing a `case` expression.

A case expression consists of an expression E whose value is used for pattern matching and a list of clauses $C_1, C_2, ..C_n$. Each clause contains a pattern P_i, a list of guards G_i and a clause body B_i (which is a list of expressions). The expected behaviour of a case expression is as follows: the body of the first clause, whose pattern matches with E and whose guards (if any) evaluate to true, is executed. This clause is also called the *matching clause*. If no such clause is found, an error is reported.

To achieve this semantics in the partial evaluator, we first reduce E to yield a reduced E'. E' is then converted to a value, which results in the value E'' and some defining bindings for the generated fresh variables in E''. Then, we reduce the clauses to yield reduced clauses $C'_1, C'_2, ..C'_n$.

Now, we are ready to do the pattern matching on E'', and we must chose the first (reduced) matching clause. If no one matching clause can be determined at specialization time, a new case expression is built using the reduced clauses which may match at runtime. Otherwise, the body of the first matching is returned as the reduced expression.

Finding a matching clause happens in two stages: a *filter* stage and a *build* stage. The filter stage performs pattern matching in each clause and uses the result of unification to reduce the clause C'_i to yield C''_i. When P''_i and E'' are static values, we can determine if the pattern matches: if it does, then we have found a matching clause and filter returns immediately. However, if they aren't static values then we do not have enough information at the time to decide whether this clause will match and hence we must retain it in the result of filter. The other case is when the unification of pattern P'_i E'' fails, in which case we may discard C''_i (since they will never match), and proceeds to the check the next clause.

After this, the build stage packs the resulting clauses from filter into an expression. If the result is exactly one clause, then the body of the clause is returned as the reduced expression. However, if it contains multiple clauses, then it means that the reduction cannot decide over one clause. In this case, all the remaining clauses are packed into a case expression—which is returned as the reduced expression.

4.2.5 Function application

A function in Erlang can have multiple clauses. The selection of a function clause to be executed in a function call is the same as `case`: the body of the first matching clause (where patterns match the arguments and guards evaluate to true) is executed. The difference, however, is that a function can contain multiple patterns in a clause, while a `case` can contain only one.

To reduce a function application, first we reduce the arguments. The current implementation of function application takes a conservative approach and only reduces function applications when all the reduced values of the arguments are static. A reduced value is static if `isStatic/1` returns true for it (implementation for a part which is shown in figure 4.2).

Once the arguments have been reduced, the process is the same as `case`: the body of the matching clause is reduced. To avoid any name clashes, we perform *alpha renaming* of the

```
isStatic({float,_,_})    -> true;
isStatic({integer,_,_}) -> true;
...
isStatic({var,_,_})      -> false;
isStatic({cons,_,H,T})   -> isStatic(H) and isStatic(T);
isStatic({tuple,_,Es})   -> lists:all(fun isStatic/1,Es);
isStatic(_)              -> false.
```

Figure 4.2: Check if an AST node is a static value

entire function body. However, if a matching clause cannot be identified at specialization time, then an error is reported.

This approach to partial evaluation for function application only works when the body of the function is available. For built-in functions, this is not the case. For such functions, we use the Erlang meta-interpreter `erl_eval`. `erl_eval` is an OTP library which can be used to evaluate Erlang expressions where all values to bindings are available. As a result, the partial evaluator does not need the body of built-in functions for static function calls—a rather nice property to have in practice.

4.3 Termination

Given our conservative approach to function call unfolding (which happens only when all arguments static), partial evaluation always terminates *unless* the original program actually crashes or the static computation contains an infinite loop—in which case the original program will also never terminate at runtime.

4.4 Combining with type inference

The partial evaluator has been implemented as a parse transform in Erlang. To control the effects of partial evaluation, the programmer is given control over which functions are partially evaluated. The programmer must specify a `-etc(pe)` annotation above the function to the type checker to indicate that a function must be partially evaluated. The partial evaluation parse transformation is performed as a pre-pass to type inference by partially evaluating only the annotated functions.

This approach of partial evaluation prior to type inference, brings up two interesting questions:

1. Can partial evaluation lead to loss of information, i.e., can partial evaluation of a well typed term make it ill-typed?
2. Can partial evaluation of an ill-typed term become well-typed by partial evaluation.

We address these questions in order.

Preservation (also called *subject reduction*) is a property of a type system which states that if a term e is of type T, and e evaluates to e', then e' is also of type T. In light of this property, partial evaluation of a term cannot make a well-typed term ill-typed. However, preservation needs to be proved for this claim to hold. Typically, this is done by a proof by induction on the typing judgments by using the evaluation rules of the language. For this, we would need to formalize the typing judgments and the rules of partial evaluation for Erlang. This task, although interesting, is a large one since Erlang is a sophisticated language with complicated typing and evaluation rules. Hence, it is outside the scope of this thesis and we do not prove it.

The opposite property (called *subject expansion*), is if a term e evaluates to e', and e' is of type T, then e is of type T. This property does not hold almost always in type systems. Consider this counter example:

```
foo() ->
    if
        true -> 1.0;
        false -> ""
    end.
```

Partial evaluation of this function yields the residual function:

```
foo() ->
    1.0.
```

The reduced expression is evidently typeable (it has the type $foo/0 :: () \rightarrow float()$, but the original expression cannot be typed as the bodies of the clause have different types.

Hence, subject expansion does not hold in our type system, which means that an ill-typed expression can become well-typed by partial evaluation. This is actually an advantage and the whole point of partial evaluation prior to type inference. An expression is ill-typed because the type system cannot construct a proof for the expression using the typing judgments, and partial evaluation will only help with simplifying this by reducing the expression for which the type system is able to construct a proof. The other outcome is that partial evaluation crashes, in which case, the type checker throws an error. This is also inline with the philosophy of rejecting programs which may crash at runtime.

5 Results

Evaluating a type checker is a tricky problem. One way to do this is by running it against various libraries. However, applying our type checker to popular Erlang/OTP libraries demands a much larger coverage of the language. For example, most Erlang programs use features such as remote function calls (functions defined in other modules), records and error handling. These features have not been implemented yet.

5.1 Evaluation

In this thesis, we evaluate our type checker by running it against many example functions (shown in earlier chapters) and some small single module Erlang libraries. These libraries have been selected on the basis of the subset of the language implemented by the type checker. The selection includes a couple of OTP libraries and a library which implements a fault tolerant distributed resource pool. The following table shows the number of lines of code (LOC) added/modified in the module to make the type checker accept it.

Library	**LOC**	**LOC added**	**LOC modified**
OTP/ordsets	179	0	0
OTP/orddict	150	1	1
ft_worker_pool	73	2	0

The LOC added represents the addition of ADT definitions. The LOC modified, in the case of OTP/orddict, is caused by a function called `take/2`. The function `take/2` takes a key and a dictionary, and returns the value corresponding to the key in the dictionary. If a value is found, it returns a tuple of the value and a dictionary without the value. Otherwise, it returns the atom `error`—clearly a different type—and hence, the type checker fails to unify these types. To mitigate this, the tuple is wrapped using a `ok` constructor defined by the following ADT (which explains the added LOC):

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

What is more interesting is the types inferred by the type checker and the information we gain from it. The type inferencer makes a lot of implicit information explicit. This serves a useful tool to understand the function. Let's look at a few examples from these modules and some others which illustrate the use of partial evaluation.

5.2 More Examples

Ex 1
```
find(Key, [{K,_}|_]) when Key < K -> error;
find(Key, [{K,_}|D]) when Key > K -> find(Key, D);
find(_Key, [{_K,Value}|_]) -> {ok,Value};
find(_, []) -> error.
```

The inferred type for this function from orddict is:

$$(A, [\{A, B\}]) \to \mathit{maybe}(B)$$

Ex 2
```
intersection([E1|Es1], [E2|_]=Set2) when E1 < E2 ->
    intersection(Es1, Set2);
intersection([E1|_]=Set1, [E2|Es2]) when E1 > E2 ->
    intersection(Es2, Set1);
intersection([E1|Es1], [_E2|Es2]) ->
    [E1|intersection(Es1, Es2)];
intersection([], _) ->
    [];
intersection(_, []) ->
    [].
```

The inferred type of this function from ordsets is:

$$([A], [A]) \to [A]$$

Ex 3
```
fold(F, Acc, [{Key,Val}|D]) ->
    fold(F, F(Key, Val, Acc), D);
fold(F, Acc, []) when is_function(F, 3) -> Acc.
```

The inferred type of this function from orddict is:

$$((A, B, C) \to C, C, [\{A, B\}]) \to C$$

Note that even though the function uses a function such as `is_function` which would be disallowed in most statically typed languages, this function type checks successfully. This is because the arity of the function is available statically, and the type checker takes advantage of this.

Ex 4
```
fetch(Key, [{K,_}|D]) when Key > K -> fetch(Key, D);
fetch(Key, [{K,Value}|_]) when Key == K -> Value.
```

The inferred type of this function from orddict is:

$$(A, [\{A, B\}]) \to B$$

Ex 5 The `nthFields` function is originally untypeable since the type of the result depends on the input arguments. However, using partial evaluation, its application in `evalNthFields` can be typed successfully as shown:

```
-etc(skip).
nthFields(_,[]) ->
    [];
nthFields(N,[A|As]) ->
    [element(N,A) | nthFields(N,As)].

-etc(pe).
evalNthFields() -> nthFields(1,[{1,2},{3,4}]).
```

The inferred type of the latter function is:

$$evalNthFields/0 :: Num\ D \Rightarrow () \rightarrow [D]$$

Ex 6 The following function replaces a cons constructor by a two-tuple, and a nil by an empty tuple. The original application (as in the previous example) is untypeable. However, we can type its application when the arguments are available:

```
-etc(skip).
tuplize([]) ->
    {};
tuplize([X|Xs]) ->
    {X,tuplize(Xs)}.

-etc(pe).
evalTuplize() -> tuplize([1,2,3]).
```

The inferred type is:

$$evalTuplize/0 :: (Num\ E, Num\ F, Num\ G) \Rightarrow () \rightarrow \{E, \{F, \{G, \{\}\}\}\}$$

Ex 7 This example demonstrates that partial evaluation is not limited to static values.

```
-etc(pe).
foo(F,G,X) ->
    T = {F(X),G(X)},
    element(1,T).
```

In spite of `F`, `G` and `X` being dynamic variables, the inferred type of this function is:

$$foo/3 :: ((A) \rightarrow B, (A) \rightarrow C, A) \rightarrow B$$

This is because the structure of `T` to determine that that the return type must be the type of `F(X)`.

6
Discussion

6.1 Missing features

The current implementation is far from complete and there are many more features which are required to type large Erlang programs. Most importantly, this includes modules, records and error handling.

Currently, remote function calls (calls to functions in other modules) are handled as follows: the type checker creates a module interface for a module which has been type checked successfully, and when a remote function call is encountered, it reads the interface file if it exists and gets the type from it. However, if it does not exist, it simply spawns a fresh set of type variables for the function and its arguments—creating an avenue for uncaught type errors. This is more of an in-place mechanism rather than a solution to type checking remote function calls. Given this simple approach, it also does not handle module level dependencies.

6.2 Limitations

The main limitation of the current system is that it does not type check concurrency. Although it does not omit it, the current approach is very simplistic and can lead to uncaught type errors. For example, consider this program:

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

baz() ->
    foo() + 1.0.
```

This program type checks successfully. This is because `foo` is assigned the type $() \to A$. Evidently, this need not be the case, and if `foo` receives a non-numeric type, this program crashes at `baz`.

6.3 Future work

6.3.1 Records

Records can be typed by generating an ADT for them. For example, for the following record

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

we could generate the following ADT:

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

The ADT has a single constructor where the arguments to the constructors are the types of the fields. When the type of a field is not defined, it is parametrized over the type of the ADT. In this fashion, a record field access can simply return the type of the constructor argument corresponding to the field.

A record update, on the other hand, 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 specified in the record definition, then updated value must be of the same type as the specified 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 value, we can assign this function the type

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

6.3.2 Concurrency

The most interesting next step would be to type check concurrency. For typing concurrency in Erlang, an idea which has been "around for a while" is adding effects to the type system [MW97] [Hug02] and typing the pid of the process which receives a message of a certain type, say $float()$, as $pid(float())$. 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. The types would look as follows:

$$spawn/1 :: (() \to \tau \,\mathbf{receives}\, \alpha) \to pid(\alpha)$$
$$(!) :: (pid(\alpha),\ \alpha) \to \alpha$$

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` is a function which, when called with a function as an argument, spawns a process and returns the pid. When a function of type $() \to e\,\mathbf{receives}\,\alpha$ 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 $pid(\alpha)$.

6.3.3 Partial Evaluation

The current partial evaluator is very simple and far from complete. It does not specialize functions where only some of the arguments are available statically. These are in fact the most interesting cases and would lead to much better residual code. For example, the reduction of `power(3,X)` to `X*X*X*1` (discussed earlier) is an example of this because only one of the arguments are static. The partial evaluator also needs to be implemented for more parts of the language such as higher order functions and records.

Currently, although partial evaluation strives to not re-order or duplicate side-effects, the preservation of semantics requires a more thorough analysis. The behaviour in the presence of concurrency has not been tested, and is also an avenue for further work. Partial evaluation of concurrent programs is a challenging problem and appears to have very little literature which is directly applicable for Erlang.

6.3.4 Integrating Partial Evaluation and Type Inference

Another avenue for exploration is better integration of the partial evaluator and the type checker. Currently, partial evaluation is simply a pre-pass to type checking, and this approach fails to take advantage of information in one stage in the other. For example, consider the following program

```
foo(Z) ->
     [{A,B}|[X|Xs]] = Z,
     element(2,X)
```

Type inference knows that `X` is of the same type as `{A,B}`, but partial evaluation does not know that, and hence it does not reduce `element(2,X)` further.

It might be possible to engineer the integration in more interesting ways. One way is to use partial evaluation as a sub-routine in type inference and call it when a term cannot be typed. This way, the partial evaluator can be supplied with information from type inference. This approach may also avoid the need for annotations. A more closely coupled approach is *type specialization* [Hug96], which achieves much better exploitation of static information. However, this is a completely different line of work, and this thesis takes a more modular approach to this problem by developing the type inferencer and partial evaluator separately.

6.4 Conclusion

In this thesis, most notably, we develop ways to type overloaded constructors in a Hindley-Milner type system and also illustrate the novel idea of applying partial evaluation to aid type inference. Overall, the results to type Erlang look promising. The inferred types are very informative and appear to expose a lot of the program structure. Partial evaluation also appears to be a suitable to choice to allow some restricted form of flexibility that is enjoyed by Erlang programmers.

Bibliography

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation process an approach to a compiler-compiler. *Systems, computers, controls*, 2(5):45–50, 1971.

[Hug96] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In *Partial Evaluation*, pages 183–215. Springer, 1996.

[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.

[Jon99] Mark P Jones. Typing haskell in haskell. In *Haskell workshop*, volume 7, 1999.

[LS05] Tobias Lindahl and Konstantinos Sagonas. Typer: a type annotator of erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25. ACM, 2005.

[LS06] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 167–178. ACM, 2006.

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

[Rém02] Didier Rémy. Using, understanding, and unraveling the ocaml language from practice to theory and vice versa. In *Applied Semantics*, pages 413–536. Springer, 2002.

[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.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In *International Conference on Formal Methods in Computer-Aided Design*, pages 82–99. Springer, 1998.

A

Appendix 1

A.1 Source code

The entire source code of the implementation in this thesis is open source and can be found at: `https://github.com/nachivpn/mt`

It is worth noting that the implementation is a prototype of the ideas in this thesis and hasn't been tested thoroughly. Hence, it may contain undetected issues.

A.2 Types of built-in operators

```
'+'       :: Num A ⇒ (A,A) → A
'-'       :: Num A ⇒ (A,A) → A
'*'       :: Num A ⇒ (A,A) → A
'/'       :: (Num A, Num B)⇒ (A,B) → float()
'div'     :: (integer(),integer()) → integer()
'rem'     :: (integer(),integer()) → integer()
'band'    :: (integer(),integer()) → integer()
'bor'     :: (integer(),integer()) → integer()
'bxor'    :: (integer(),integer()) → integer()
'bsl'     :: (integer(),integer()) → integer()
'bsr'     :: (integer(),integer()) → integer()
'not'     :: (boolean()) → boolean()
'and'     :: (boolean(),boolean()) → boolean()
'or'      :: (boolean(),boolean()) → boolean()
'xor'     :: (boolean(),boolean()) → boolean()
'orelse'   :: (boolean(),boolean()) → boolean()
'andalso'  :: (boolean(),boolean()) → boolean()
'=='      :: (A,A) → boolean()
'/='      :: (A,A)→boolean()
'=<'      :: (A,A)→boolean()
'<'       :: (A,A)→boolean()
'>='      :: (A,A)→boolean()
'>'       :: (A,A)→boolean()
'=:='     :: (A,A)→boolean()
'=/='     :: (A,A)→boolean()
'++'      :: ([A],[A])→ [A]
'--'      :: ([A],[A])→ [A]
'!'       :: Padd A ⇒ (A,B)→ B
```

A.3 Types of built-in functions

```
length/1 :: [A] → integer()
is_atom/1 :: atom() → boolean()
is_integer/1 :: integer() → boolean()
is_list/1 :: [A] → boolean()
is_boolean/1 :: boolean() → boolean()
self/0 ::() → pid()
spawn/1 :: (() → A) → pid()
spawn/2 :: (atom(),() → A) → pid()
spawn_link/2 ::(atom(),() → A) → pid()
exit/1 :: (A) → B
node/0 :: () → atom()
nodes/0 :: () → [atom()]
process_flag/2 :: (atom(),boolean()) → boolean()
unlink/1 :: Port A ⇒ (A) → boolean()
make_ref/0 ::  () → reference()
```

A.4 Built-in type classes and instances

1. Class: `Num`; Instances: `integer()`, `float()`
2. Class: `Padd`; Instances: `pid()`, `atom()`, `{atom(),atom()}`
3. Class: `Port`; Instances: `pid()`, `port()`

A.5 The need for Stålmarck's method

As noted earlier, in the absence of applying Stålmarck's method, some type errors might be missed due to loss of unification information in the type constraints. Here's an example of such a case:

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

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

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

The type checker should reject this program since a float is assigned to a boolean (returned by `getReason`). Instead, it infers the following type:

```
foo/0 ::
    (A,float()) → B ∼
        {(pid(),float())  → cl (float()),
        (pid(),float())  → sr(float())}
    B ∼ {cl(boolean()) || sr(boolean())}
    ⇒ () → float
```

Note that the type constraint is indeed a valid, but also unsolvable. A manual examination of the type constraint tells us irrespective of whether B unifies with $cl(float())$ or $sr(float())$, the next constraint creates a problem because then it cannot unify with *cl*(*boolean*()) or *sr*(*boolean*(). Applying Stålmarck's method helps us extract this information from the type constraints to reject this program as all branches of the proof tree would lead to a contradiction.

A.6 Type errors

1. Ensuring all function clauses return the same type:

```
1. foo4() -> 3.0 ;
2. foo4() -> "hello".

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Cannot unify
{bt,1,float} with {bt,2,string}",..
```

2. Ensuring a when clause is boolean:

```
41. foo5 (N) when 1.0 -> true.

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Cannot unify
{bt,41,float} with {bt,41,boolean}",..
```

3. Mixing integer and float values:

```
269. float3() -> (1.0 + 2.0).
270. integer3() -> 3 div 1.
271. heter_add() -> (float3() + integer3()).

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Cannot unify
{bt,269,float} with {bt,270,integer}",..
```

4. Mixing numeric with non-numeric values:

```
82. foo8() -> 1;
83. foo8() -> "Hello".

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Cannot solve predicate:
{class,"Num",{bt,83,string}}",
```