Master thesis Halftime report

Static typing of Erlang programs using Partial Evaluation

Nachiappan Valliappan, nacval@student.chalmers.se

Supervisor: John Hughes

April 9, 2018

1 Introduction

1.1 Background

Erlang is a concurrent functional programming language popular for its use in distributed applications. Being a dynamically typed language by design, it allows the successful compilation and execution of many programs which would typically be rejected by a type checker of a statically typed language. This idiosyncrasy of Erlang makes it difficult to retrofit existing type checking technology onto the language. Developing a static type system suitable for Erlang is the goal of this thesis.

1.2 Typing Erlang using Partial Evaluation

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

```
- module(sample).
- export([remote_hello/2]).
remote_hello(Node, Module) ->
    spawn(Node, Module, hello, []).
```

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

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

Partial evaluation of the program yields:

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

The unknown value `Module` has been replaced with the known value `sample`. Since our definition of the module `sample` does not define a function `hello/0`, this program can be rejected by the type checker. This is a simple example of a case where partial evaluation can help type check programs in Erlang.

1.3 Aim

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

1.4 Report overview

This document reports on the first half of the thesis: implementing a static type checker for Erlang. Chapter 2 serves as background for Chapter 3: it presents a Hindley-Milner (HM) type system for the Simply Typed Lambda Calculus (STLC) and discusses the type inference algorithm. Chapter 3 further extends this to Erlang: it discusses the design and implementation of new extensions which are required in the type system to type check Erlang.

2 Type Inference for STLC

This section provides an overview of the HM type system by presenting its application for STLC. STLC is a powerful functional programming language with a very small core, and hence serves as a suitable foundation for our study. Many definitions and concepts in this section have been directly inspired from standard literature [Pie02] [Ran12].

2.1 The Hindley-Milner type system

The HM type system was discovered independently by Hindley [Hin69] and Milner [Mil78], and has been proved to be both sound and complete [Dam84]. In imperative programming languages such as Java or C++, the type systems essentially check whether a given program is type correct using the types provided by the programmer. The HM type system, on the other hand, in addition to type checking, also calculates the types of terms. This frees the programmer from having to specify types and hence leads to lesser code. The HM type system has been studied closely with typing functional programming languages [Dam84] [DM82] and hence forms a strong basis for typing them.

The most notable feature of the HM type system is its ability to calculate the *principal type* for a term. Intuitively, the principal type of a term is its *most general* type. When a term has been assigned its principal type, it can be used in many different contexts due to its generality. For example, consider assigning a type to the identity function $\lambda x.x$. It's useful to assign the general polymorphic type $\forall a.a \to a$ rather than a more specific type (such as $Int \to Int$) as it allows us to instantiate a as necessary: a can be instantiated, for example, to $Bool$ to yield a function of type $Bool \to Bool$, or to Int to yield $Int \to Int$. Calculating the principal type of a term is called *type inference*.

2.2 Term language

The core of STLC is very small. It consists of variables, function application, lambda abstraction, and a let construct. A let construct is used to allow *let-polymorphism* (discussed further in 2.5). The entire term language can be succinctly described using the grammar:

$\langle expr \rangle ::= \langle var \rangle$
$\quad | \quad \langle expr \rangle \ \langle expr \rangle$
$\quad | \quad \lambda \ \langle var \rangle . \langle expr \rangle$
$\quad | \quad \text{let} \ \langle var \rangle = \langle expr \rangle \ \text{in} \ \langle expr \rangle$

Note that the variable in the lambda abstraction is not annotated with a type.

2.3 Type language

The type language in the Hindley-Milner type system consists of two kind of types: *mono* and *poly* types. Mono types are concrete types without any quantifiers such as Int (a base type), α (a type variable), or $(\alpha, \beta) \rightarrow Bool$ (a function type). In the rest of this document, unless specified, a "type" simply refers to a monotype.

$\langle mono \rangle ::= \langle base \rangle$
$\quad | \quad \langle tvar \rangle$
$\quad | \quad [\langle mono \rangle] \rightarrow \langle mono \rangle$

Polytypes (also known as type schemes) are monotypes bound by zero or more quantifiers. By definition, any monotype is also a valid polytype.

$\langle poly \rangle ::= \langle mono \rangle$
$\quad | \quad \forall \ \langle tvar \rangle . \langle poly \rangle$

2.4 Type Inference: Background

At the heart of type inference lies *unification*. We define some fundamental concepts involved in unification first.

2.4.1 Type substitution

A *substitution* σ is a mapping of type variables to types. A substitution σ is *applied* to a type t - denoted by $\sigma(t)$ - by replacing all occurrences of the type variables in the domain of the substitution by their corresponding values in the co-domain of the mapping. For example, a substitution $\{x \mapsto Int, y \mapsto Bool\}$ when applied to the type $(x, y) \rightarrow z$ yields the type $(Int, Bool) \rightarrow z$. The notion of substitution here is the standard one which replaces only free variables and accounts for name capture.

Intuitively, a substitution carries known information about a type. During unification, there arises a need to accumulate all known information. This accumulation is done by *composing* two substitutions. The composition of two substitutions is formally defined as $(\gamma \circ \sigma)t = \gamma(\sigma(t))$.

A substitution σ is said to be *more general* than a substitution σ' if there exists a γ such that $\sigma' = \gamma \circ \sigma$. Being more general (or less specific) gives rise to an ordering of substitutions: $\sigma \sqsubseteq \sigma'$ meaning that σ is more general than σ'.

2.4.2 Unification overview

A unifying substitution or a *unifier* of two types t_1 and t_2 is a substitution σ that equates two types when applied to them: $\sigma(t_1) = \sigma(t_2)$. A substitution σ is a *principal unifier* if $\forall\sigma'$, where σ' unifies t_1 and t_2, $\sigma \sqsubseteq \sigma'$. Unification is the process of finding the principal unifier of two types.

2.4.3 Unification algorithm

The unification algorithm is a partial function `unify` which accepts two types as its arguments and returns the principal unifier of the two types.

```
unify ([As1] → B1) ([As2] → B2) =
    let ϕ = unify (As1, As2),
        ψ = unify (ϕ(B1), ϕ(B2)),
    in ψ ∘ ϕ;
unify x T
    | x == T     = {}
    | occurs x T = throw("Failed occurs check")
    | otherwise  = {X ↦ T}
unify T x =
    unify x T
unify T U
    | T == U    = {}
    | otherwise = throw("Unification failed")
```

2.4.4 Unification Constraints

During type inference `unify` is called several times and the resulting substitution is applied to a certain type. An alternative approach is to completely defer unification to be done after type inference. This is done by collecting *unification constraints* (which are essentially the arguments to `unify`) as a pair of types `(U,V)`. In this approach, type inference results in a type for a term and a set of unification constraints. The resulting type is the type of the subject term "under the unification constraints".

To get the principal type of the term, the unification constraints must be solved and the resulting substitution - called a *solution* - is applied to the inferred type. A unifier σ is a solution for a constraint set C if it unifies all constraints in C. A *principal solution* σ is the principal unifier of constraint set C, i.e., for all solutions σ' of C, $\sigma \sqsubseteq \sigma'$.

The primary reason to implement type inference using unification constraints is modularity. The implementation of type inference is completely freed from the clutter caused by repetitive substitutions and compositions. The small price to pay here is useful error messages. Type errors are identified after type inference, and hence any useful context (such as line numbers) to display relevant error messages is lost. In our implementation, we alleviate this to a reasonable extent by constructing a type along with its line number. Hence, every constraint - which is essentially a pair of types - contains the line numbers of the types being unified.

2.5 Type Inference: Algorithm

The goal of type inference is to calculate to the most general type for a term. This requires two input arguments: a subject term and an environment which assigns types to free variables in the term. The result of inference is a type and a set of unification constraints. The solution of the unification constraints is applied to resulting type to yield the *principal type*. Formally, if type inference of a term z yields a type t and a constraint set C, then the principal type of z is $\sigma(t)$, where σ is the principal solution of C. Let's do a case by case analysis of terms for implementing type inference.

2.5.1 Lambda abstraction

The type of a lambda abstraction is a function type from the type of its argument to the type of its body. The type of the body must be inferred in an environment extended with the argument variable as it may occur freely in the body.

```
infer (env, λx.b) =
    let fresht = gen_fresh()
        env'   = extend(x,fresht,env)
        (t,cs) = infer(env',b)
    in ([a] → t, cs)
```

The environment is extended with the variable `x` and a fresh type `fresht` generated using the `gen_fresh()` function. The inference of the body results in a type `t` and a set of constraints `cs`. These set of constraints may restrict the principal type of `x` and `b`, and hence we must return these constraints along with the constructed type $[a] \to t$. For example, the application `infer(env,`λ`x.x)` - for any valid `env` - results in the type $a \to a$ and an empty set of unification constraints, where a is a fresh type.

2.5.2 Function application

The type of a function application $f\ a$ is the return type of the function f given that the type of argument a matches the type of the argument expected by f. We enforce this by adding a constraint to the constraint set that the inferred type of f must unify with with the function type `[t2]`→`v`, where `t2` is the inferred type of `a` and `v` is a fresh type.

```
infer (env, f a) =
    let (t1,cs1)    = infer(env,f)
        (t2,cs2)    = infer(env,a)
        v           = gen_fresh()
    in (v, cs1 ∪ cs2 ∪ {(t1,[t2]→v)})
```

We collect all constraints which occur during inference (including our newly added constraint) and return them along with the type of the function application `v`.

2.5.3 Let

The case of `let` is a little more complex because the variable `x` cannot be just given the type of the expression it's assigned to. To understand why, consider the expression `let id = (lam z.z) in (let y = id 5 in id True)`. Note that `id` is applied to arguments of different types. Had the identifier `id` been assigned a type `t` → `t`, the type variable `t` would get unified with `Int` (the type of 5), and hence type of `id` would be specialized to `Int->Int`. While this seems reasonable, this poses a problem when we try to infer the type of application `id True` as `id` has type `Int` → `Int`. This approach would lead to a unification error. Instead, we would like to assign a valid type to this expression. This is the whole point of having a separate `let` construct in the language (also called as *let-polymorphism*).

To solve this problem, we need some way to record the information that the type of `id` is of the "scheme" `a` → `a`. This is precisely what polytypes are for! The polytype of `id` is ∀`a.a` → `a`. To compute this polytype, we first compute the principal type of `id` and then convert it to a polytype by *generalizing* it. Generalization is achieved by binding all type variables in a type that do not occur freely in the environment. For example, the generalization of a type `a` → `a` in the environment $\{y \mapsto b\}$ yields the polytype ∀`a.a` → `a`, and the generalization of `b` → `b` in the same environment yields `b` → `b`.

```
infer (env, let x = e1 in e2) =
    let (t1, cs1)   = infer(env,e1)
        φ           = solve(cs1)
        t1'         = generalize(φ(t1),φ(env))
        env'        = extend(x,t1',φ(env))
        {t2, cs2}   = infer(env',e2)
    in (t2, cs1 ∪ cs2).
```

The principal solution ϕ for the constraint set `cs1` is computed first, and then the principle type ϕ`(t1)` is generalized to obtain its polytype `t1'`. The environment is extended with an entry for `x` with it's polytype `t1'`. The type of the remaining expression `e2` is then inferred in the extended environment `env'`. As in the previous cases, we collect all constraints which occur during the process and return them along with the inferred type `t2`.

2.5.4 Variable

Type inference on a variable must return the type of the variable. The type of the varable can be looked up in the environment. But note the environment contains only polytypes (since every monotype is also a polytype). So, to convert a polytype back to a monotype, we define an operation `freshen` which replaces all bound variables with fresh type variables and returns a monotype.

```
infer (Env, X)
    case lookup(X,Env) of
        undefined   -> throw("Unbound variable");
        T           -> (freshen(T),{})
```

Since no new constraints arise during a lookup, we return an empty set of unification constraints.

2.5.5 Putting it together

Now consider the previous example again: `let id = (lam z.z) in (let y = id 5 in id True)`. Given our approach to inference for let, the body of the main let expression is type checked in the environment $\{id \mapsto \forall a.a \rightarrow a\}$. The inference on `id` in `id 5` results in a fresh type `t1` $\rightarrow$ `t1` (where `a` has been "freshened" with `t1`) and the inference on `id` in `id True` results in a fresh type `t2` $\rightarrow$ `t2` (where `a` has been "freshened" with `t2`) . Eventually `t1` gets specialized to `Int` and `t2` to `Bool`. There is no clash of types during the unification phase, but the requirement that `id` returns the same type as its argument is captured perfectly.

3 Type Inference for Erlang

Although Erlang is a dynamically language, it supports type specifications based on the work by Lindahl and Sagonas [LS06]. These type specifications serve as a form of documentation and are used by a static analysis tool called Dialyzer. Dialyzer is shipped with the default OTP distribution and hence plays an important role in how programmers think about data types in Erlang. In an attempt to take advantage of this existing system, we base many elements of our type system on their work.

The existing type specification language and the type system implemented by the Dialyzer are heavily based on subtyping principles. We learn from the evaluation of the work by Marlow and Wadler [MW97] and experience of using the Dialyzer that type inference does not mix well with subtyping, hence we are careful to avoid the subtyping parts of the type system. In effect, their type system solves unification constraints of the form $U \subseteq V$, while our type system solves unification constraints of the form $U = V$.

Extending the type system discussed in the previous section to Erlang requires us to make various extensions. This section discusses these extensions in detail. The type checker presented in this section has been implemented as a parse transform and can be used by adding the compiler directive: `-compile(parse_transform, etc).` (assuming the type checker and it's dependencies have been compiled and are available in the current directory of the module being compiled). The entire source code can be found online at https://github.com/nachivpn/mt.

3.1 Base types

Erlang supports a number of base types: integer, float, string, boolean and atom. This list is not complete, but contains all the types which arise when attempting to type check the "base" language (without functions from the run-time system or the OTP library). The current implementation only supports these types. The other base types which will need to be implemented when attempting to type check the Erlang BIFs (such as `make_ref`, `spawn` etc.) from the run-time system include PID, reference, bit/binary string and port identifier.

Since the type language in section 2.3 already includes a base type representation, we don't need any extensions. Inference for base types is straightforward: every time a value of a certain type is found, then the corresponding base type is returned as the type with an empty list of unification constraints. The story for numeric types (especially int) is actually different and a little more involved, but we defer that discussion to section 3.3.

3.2 Functions

3.2.1 Simple functions

A function type in Erlang contains a list of argument types and a return type. The type language in section 2.3 is already generic enough to capture Erlang function types. However, the unification algorithm in section 2.4.3 must be adapted accordingly as it only handles singleton list arguments. The unification of arguments must now ensure that the number of arguments match and it must carry forward the substitution arising at the unification of arguments at a certain position by applying it to the remaining arguments and the return type. The exact implementation is straight-forward, but tedious to discuss here.

The inference algorithm remains the same: extend the environment with a fresh type variable for every variable argument and type check the function body in the extended environment. The following example demonstrates the types assigned to simple functions:

```
foo(X,Y) -> X.
foo1(F,X) -> F(X).

...> c(test).
{foo,2}  :: ∀A[].∀B[].(A,B)->A
{foo1,2} :: ∀A[].∀B[].((B)->A,B)->A
```

where {`foo,2`} denotes the function `foo` with arity 2.

3.2.2 Function body

The previous example only considers functions with a single expression in its body. A typical Erlang function may have multiple expressions separated by a "`,`". Since the value of the last expression in the function body is returned by the function, the return type of the function is simply the type of the last expression. The other expressions in the function body must be type correct. Moreover, these expressions may bring variables into scope and contain information about the type of the arguments and the return type. Hence, in addition to type inference, we also need *type checking*.

The roles of type checking are three fold: ensuring every expression is well typed, modifying the environment by bringing new variables into scope, and collecting unification constraints from the body. For example, while inferring the type of this function

```
foo2(X) ->
    X = 3.0,
    Y = 1.0,
    X + Y.
```

type checking ensures that the type of the last expression `X + Y` is inferred in an environment updated with with type of variable `Y`. Since the unification constraints are accumulated through type checking, it also ensures that the type of the argument `X` is restricted to float.

Type checking uses type inference to ensure that an expression is well typed. The inferred type is thrown away (because the type of `Y = 1.0`, for example, is useless), but the unification constraints are collected. The final inferred type of the function is rightly: {`foo2,1`}`::(float)->float`.

3.2.3 Top level functions

An Erlang program contains a list of top level function definitions. Once the type of a top level function definition is inferred, the environment is extended with an entry for the function that is uniquely identified by the tuple {<name>,<arity>}. Since a function name is identified along with it's arity in the environment, overloaded functions names are allowed as long as they have different arities - which is exactly the behaviour expected from Erlang programs.

An important point to note here is that the inferred type must be generalized before it's added to the environment. This is to ensure that the type of the function is not specialized to its usage. The reason becomes more apparent in the next section. For now, note that the appearance of the $\forall$ quantifier in the types of examples above is due to generalization of the inferred types.

3.2.4 Function application

Function bodies may contains calls to other functions. When such a function call is encountered in the body of another function, the called function is looked up in the environment (along with its arity) and its type is freshened and unified with the type of the application (similar to the case function application for STLC). If the function type is not generalized and freshened when it goes in and out of the environment, we might lose the polymorphic nature of the function (the problem is the same as in the case of let-polymorphism).

For a function to be looked up in the environment, its type must already be available in the environment. One option here is to assume that functions are listed in order of dependency, but this assumption fails to hold for recursive - especially mutually recursive - functions. Hence, in order to ensure the invariant that the environment contains the type of legal free variables, we must do some dependency analysis. This is discussed further in section 3.4.

3.2.5 Function clauses

An Erlang function may contain multiple clauses separated by a ";" such as:

```
foo3(F,X,Y) -> F(X);
foo3(F,X,Y) -> F(Y).
```

The type of this function is inferred correctly as: {foo3,3} :: ∀A[].∀B[]. ((A)->B,A,A)->B. This is achieved by inferring the type of each clause individually and unifying the inferred types. If the types of the clauses cannot be unified such as in this function:

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

the type checker reports the error as:

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

which represents that the base type `float` in line 1 cannot be unified with the base type `string` on line 2.

3.2.6 Clause guards

Every function clause may contain a `when` clause composed of a disjunction of conjunction of boolean expressions. We must type check these expressions to ensure that these expressions are indeed boolean by unifying their inferred type with the base type `boolean`. As a part of type checking, unification constraints which arise in the process will be collected, hence ensuring that any valuable information on the type of arguments is used for type inference. The following examples demonstrate the behaviour of type inference in the presence of a when clause:

- 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}",..
```

- Collecting useful typing information in a when clause:

```
even (N) when N / 2.0 == 0.0 -> true;
even (_) -> false.

...> c(test).
{even,1} :: (float)->boolean
```

3.2.7 Anonymous functions

Anonymous functions are functions without a name. We simply reuse the existing machinery for top level functions to infer the type of anonymous functions. The following examples demonstrate the behaviour of the type checker for anonymous functions:

- Using higher-order functions:

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].

double(L)  -> map(fun(X) -> 2.0*X end, L).
```

```
add_one(L) -> map(fun(X) -> 1.0 + X end, L).

...> c(test).
{map,2} :: ∀A[].∀B[].((A)->B,List A)->List B
{double,1} :: (List float)->List float
{add_one,1} :: (List float)->List float
```

- Argument mismatch:

```
46. foo6() ->
47.     X = fun(N) when N >= 1.0 -> N end,
48.     X().

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Number of arguments to function
on line 47 do not match arguments on line 48",
```

But there's an issue with the current implementation: anonymous functions are not polymorphic! This is because top level functions are explicitly generalized, but there's nothing which does that for anonymous functions. This perfectly reasonable function is rejected:

```
44. foo7() ->
45.     Id = fun(X) -> X end,
46.     Id(5.0),
47.     Id(true).

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Cannot unify {bt,46,float}
with {bt,47,boolean}",
```

This issue has its roots deep in the implementation of type checking of the match operation. The match operation assigns a fresh type variable to all unbound identifiers (there maybe multiple such as `[X|[Y]]=..`) on the left hand side of a match, and then unifies the inferred type of the left hand side and the inferred type of the right hand side. This unification ensures that the (fresh) types of the identifiers get unified with types of their corresponding values. The main reason to take this approach is that it readily adapts to arbitrary patterns on the left and right which Erlang allows.

But notice how the environment is not extended with the inferred type of an identifier - unlike in the case of top level functions which gave us the opportunity to generalize the inferred type. This is where the problem lies. `Id` gets assigned a fresh type variable `a` which gets unified with the inferred function type `t -> t`, which then gets specialized to `float -> float` after the unification caused by the first application `Id(5.0)`.

In other words, `Id` is not assigned its principal polytype in the environment. Taking inspiration from let generalization in STLC, it should be possible

to fix this rather easily: compute the principal solution by solving all unification constraints which arise from type checking a match expression, apply the principal solution to types of all new identifiers on the left to yield principal types, and modify the environment entries with the generalized principal types. This has not been implemented yet.

3.2.8 Pattern matching

Until now, the described type inference for functions only handles variables as arguments in the function definition. To support pattern matching on function arguments, we must slightly modify how types are assigned to the arguments: first, we type check the arguments - this assigns fresh types to variables in the patterns - and then we infer the type of every argument in the resulting environment. The same environment is then used for type inference on the function body.

3.3 Numeric operators

A full list of types assigned to built-in operators can be found in the Appendix. This section is primarily concerned with the challenge of overloaded numeric operators. Many numeric operators in Erlang allow the mixing of `integer` and `float` values. For example, expressions of the form `<int> + <int>`, `<float> + <float>`, `<int> + <float>` and `<float> + <int>` are all legal. The first two cases call for an overloaded `+` operation, while the latter two require some form of subtype polymorphism. The current implementation supports the first two cases and some restricted forms of the latter cases as demonstrated in the next section.

3.3.1 Examples of number overloading

- Overloaded addition:

```
% polymorphic add
add(A,B) -> A + B.

% addition of floats
float3() -> add(1.0,2.0).

% addition of numbers
num3() -> add(1,2).

% specializes number value as float
num3_as_float3() -> 3.0 = num3().

% specializes number value as integer
num3_as_integer3() -> 3 div 1 = num3().
```

```
...> c(test).
{add,2} :: ∀A[Num A;].(A,A)->A
{float3,0} :: ()->float
{num3,0} :: ∀A[Num A;Num A;Num A;].()->A
{num3_as_float3,0} :: ()->float
{num3_as_integer3,0} :: ()->integer
```

A *number* is an int or a float. Numeric values that are not known to be float are treated as numbers until they are specialized to a float or an integer value. The match operation `3 div 1 = num3()` in `num3_as_integer3()` specializes the number return type of `num3()` to `integer` since div has the type `(integer,integer) -> integer`

- Mixing "integer" constants and float values:

```
float5() -> add(float3(),2).
float6() -> add(3,float3()).
...> c(test).
{float5,0} :: ()->float
{float6,0} :: ()->float
```

Although values `2` and `3` appear to be integers, they aren't within the type system. The treatment of `2` and `3` as numbers allows us to mix them with float values in numeric operations. But, as the next example demonstrates, this flexibility fails to hold if the value has already been specialized to an integer or a float.

- Mixing integer and float values:

```
76. float3() -> add(1.0,2.0).
77. integer3() -> 3 div 1.
78. heter_add() -> add(float3(),integer3()).

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: Cannot unify {bt,76,float}
with {bt,0,integer}",
```

The unification fails to unify the integer value `integer3()` with the float value `float3()`. This stems from the type assigned to addition: `+::∀A[Num A;].(A,A)->A`. Ideally, in order to allow true mixing of integer and float values, we would need a type like this: `+::∀A[Num A;].∀B[Num B;].(A,B)->Max(A,B)`. But this appears to need some form of restricted subtype polymorphism for numbers which has not been implemented yet.

- 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}}",
```

In this case, type checker rejects the program (as expected) since `"Hello"` is not a number.

3.3.2 The Num type class

Type classes are a feature of the Haskell type system which achieve *ad-hoc polymorphism* such as overloading of operators and functions. Although it's tempting to implement a full-blown Haskell like type class system for Erlang, we only need to achieve overloading of certain numeric operations. Hence, we restrict ourselves to a primitive type class system: we implement single parameter type classes without class hierarchy. These type classes can neither be defined nor extended by the programmer. They are a purely built-in feature. Currently, the only implemented type class is `Num` and it has two instances: the `integer` and `float` base types.

3.3.3 Implementing type classes

The implementation of the type class system in Haskell has been documented (in Haskell!) by MP Jones [Jon99]. We simply borrow a restricted version of this. The main idea is to add *predicates* to the type system. A class predicate is a 3-tuple $\{class, c, i\}$ which asserts that the type i is an instance of class c (where the constant $class$ is used as an identifier to differentiate it from other predicates which arise later). Naturally, the built-in type class instances are represented by a list of predicates that are known to be true. Let's call this the *premise*. In the current implementation, the premise contains the predicates $\{class, Num, integer\}$ and $\{class, Num, float\}$.

Type inference, in addition to collecting unification constraints, now also collects predicates which arise when terms are encountered. That is, type inference over a term z results in a 3-tuple $\{t, cs, ps\}$, where t is a type, cs is a set of unification constraints, and ps is a set of predicates. The unification constraints cs are solved to obtain a substitution, which is applied to t and ps to obtain the principal type t' and a new set of predicates ps'. Now, the predicates which readily follow from the premise are removed to yield a new set ps''.

If the types in the predicates are type variables, then the ps'' cannot be solved at the current time, and we must defer this till more information is available. Intuitively, the type of z is t' provided that the predicates ps'' can be solved at some point - denoted by $ps'' \Rightarrow t'$. We need a way to retain these predicates along with the type scheme in the environment. To do this this we modify the type language to hold predicates over a type variable:

$\langle poly \rangle ::= \langle mono \rangle$
$\quad | \quad \forall \langle tvar \rangle \langle predicates \rangle . \langle poly \rangle$

where $\langle predicates \rangle$ are the set of predicates over the type variable $\langle tvar \rangle$. The type of the multiplication operator `'*'::∀A[Num A;].(A,A)->A` is an example of such a polytype.

It may be the case that the instance type in some predicate in ps'' is not a type variable. In this case, we throw an `"Cannot solve predicate"` error as the predicate cannot be solved ever. $\{class, Num, String\}$ is such a predicate.

3.4 Recursive functions

During type inference, when a function application is encountered in the body of a function, the type of the called function is looked up in the environment. But what happens when a function calls itself? We the need the type of the function to infer the type of the function!

There are two kinds of recursion which are relevant to the problem of type checking here: *direct recursion* and *mutual recursion*. Direct recursion is when a function recursively calls itself. Mutual recursion is when a group of functions call each other.

3.4.1 Direct recursion

In functional languages such OCaml, the user is required to explicitly annotate a function with a `rec` keyword to indicate that it is recursive. This could be useful to indicate to a type checker that a recursive function must be treated specially. But Erlang has no such construct as this problem is irrelevant for a dynamically typed language. Hence, we need an approach to type checking 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 also enforced to be the same as the inferred type. Once the unification constraints have been solved, the solution is applied to the inferred type and the result is generalized to obtain the principal polytype of the function. This polytype is the one which finally goes into the environment for this function.

3.4.2 Mutual recursion

OCaml requires that programmers define mutually recursive functions using the same recursive let. But Haskell has no such requirement, so we once again turn to Haskell's implementation [Jon99] to free the user from the burden of

manually grouping all mutually recursive functions. The two main challenges here are identifying such groups and type checking each of them in "in the right order". This requires us to perform *dependency analysis* of functions.

A mutually recursive group is a group of functions in which every function calls every other function. If the functions were treated as nodes of a graph, and calls between them edges of the form $A \rightarrow B$ (where B is called by A, and hence the type of B is needed to infer the type of A), then this forms a *dependency graph*. A group of mutually recursive functions would be a *strongly connected component* (SCC) in this graph. The graph may have many SCCs and functions in one SCC may call functions from another SCC, so the SCCs must be topologically sorted and type inference for SCCs be performed in this order.

To perform type inference on an SCC, the unification constraints which arise from all member functions in this SCC are collected and solved together. The resulting principal solution is applied to the inferred types of the functions to yield their corresponding principal types. The resulting principal types are then generalized and added to the environment. Notice how this is different from our previous approach where unification constraints were solved individually for every function.

3.5 Algebraic Data Types

We use the type specification language (written using `-type`) to implement Algebraic Data Types (ADTs) in an Erlang program. All ADTs must be defined before they are used. The following example demonstrates an implementation and the inferred types of some tree operations:

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

emptyTree () -> {nil}.

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

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

...> c(test).
{nil,0} :: ∀A[].()->tree A
{node,3} :: ∀A[].(A,tree A,tree A)->tree A
{emptyTree,0} :: ∀A[].()->tree A
{findNode,2} :: ∀A[].(A,tree A)->boolean
```

```
{flattenTree,1} :: ∀A[].(tree A)->List A
```

Each data constructor in an ADT definition is separated by a `|`. Since Erlang does not have a separate language construct for data constructors, we mock a constructor using an atom as the first element of a tuple. In the ADT definition, the remaining elements of the tuple indicate the type of the arguments to the constructor. In the constructor application, these elements are the arguments to the constructor.

3.5.1 Type constructors

To implement ADTs in the type system, we need a way to add arbitrary new types at compile time. This is achieved by extending the type language to include *type constructors*. A type constructor "constructs a type" by accepting some types as arguments and yielding a type. The syntax of a type constructor is its name (a string) followed by its arguments (a list of types).

⟨*mono*⟩ ::= ...
  | ⟨*constructor_name*⟩ [⟨*mono*⟩]

In the above example, `tree` is a type constructor which accepts a type variable `a` to yield the type `tree a`. Since we add a new monotype, the unification algorithm must also be extended: two type constructors are unified by ensuring that the names match and by unifying the arguments.

3.5.2 Type Inference for ADTs

Within the type system, a data constructor is treated as a function which accepts a list of arguments (as specified by the constructor definition) and returns a value of type of the defined ADT. For example, the `node` constructor in the above example is assigned the type `{node,3}::∀A[].(A,tree A,tree A)->tree A`. Prior to type inference of top level functions, the types of all constructors are added to environment.

During type inference, when an atom is encountered in the first position of a tuple, it's considered to be some constructor, and its type is looked up in the environment. If the lookup returns no values, then an `"Unbound constructor"` error is thrown. If it returns a single value, then it's handled in the same way that function application is handled: the type of the constructor application is unified with its type in the environment. It may also be the case that a lookup returns multiple values. Section 3.6 handles this case as it calls for constructor overloading.

3.5.3 List and Tuples

Lists are merely a special case of the ADT implementation. Tuples also use the same machinery: a tuple is represented by a type constructor `Tuple` which accepts a number of arguments. For example, the value {`1.0,"hello"`} is of type `Tuple float string`.

3.6 Overloaded data constructors

The type system described so far is essentially an extension to the Hindley-Milner type system and is heavily inspired from Haskell. Similar to Haskell, it does not support overloading of data constructors. But this restriction is not adhered to by Erlang programmers [MW97] given the lack of a type system, and hence we must find a way to support overloaded data constructors.

In their subtyping system, Marlow and Wadler [MW97] cite overloading of constructors as the main reason for using subtyping. However, our investigation indicates that it could be possible to implement overloaded constructors via a form of ad-hoc polymorphism. This section presents this investigation and a work-in-progress implementation.

3.6.1 Ad-hoc polymorphism for overloading

An overloaded data constructor is a constructor which has multiple types in the environment. Let's calls these types the *candidate types* for a constructor. An overloaded constructor has multiple candidate types because there exists more than one ADT definition which defines a constructor with the same name and same arity. Note that since constructors are coupled with their arities in the environment, constructors with different arities will be treated differently and hence are a non-issue for overloading.

The ultimate goal is to unify the type of a constructor application with exactly one candidate type. During type inference (after the lookup), the type of the constructor application may unify with multiple candidate types. This is because we don't have enough information on the type of constructor application since other unification constraints have not been solved. Hence, we must wait for all other unification constraints to be solved, and the apply the solution to the type of constructor application before we inspect for unifiable candidates. This is achieved by carrying the type of the constructor application and the candidate types in the form of a predicate for solving after unification.

3.6.2 *oc* predicate

We add a new predicate - called the *oc* predicate - for overloaded constructors: a 3-tuple $\{oc, t, cts\}$ where *oc* is a constant predicate identifier, t is the type

of the constructor application as found in the program, and *cts* is the list of candidate types. For example, when a constructor application {`just,3.0`} is encountered, a predicate $\{oc, float \rightarrow a, [b \rightarrow Maybe\ b]\}$ is returned along with it's inferred type a and unification constraints.

Once all unification constraints have been solved, the solution is applied to the type t in the predicate. This may reduce the list of unifiable candidate types (as, for example, a may have been substituted with *integer* hence making it impossible to unify $float \rightarrow integer$ with $b \rightarrow Maybe\ b$). We throw away all the candidate types which do not unify with t. If we are left with none, we are done with a type error. If we are left with exactly one, then we apply the unifying substitution to the inferred type. If we are left with more than one candidate type, this means that we do not have enough information at this time and hence the function must be polymorphic on all the remaining candidate types!

3.6.3 Implementing constructor overloading

In this program

```
-type maybe_either(A) :: {left, A}
    | {right, A}
    | {nil}.
-type dir(A) :: {left,A}
    | {right,A}
    | {fwd,A}
    | {bwd,A}.

extract({left,A}) -> A;
extract({right,A}) -> A.

...>c(test)
{extract,1} :: ∀A[].∀B[].
        (A)->B ∼ {(A)->dir A , (A)->maybe_either A}
        (A)->B ∼ {(A)->dir A , (A)->maybe_either A}
    (B)->A
```

the constructors {`left,1`} and {`right,1`} are overloaded. It's impossible to decide from the given information whether `extract` is applicable to `either` or `dir`. Hence, it must be polymorphic over both types. For this we add these these (unsolved) predicates to the type of `extract`. `(A)->B` $\sim$ {`(A)->dir A, (A)->maybe_either A`} is such a predicate (where the types after $\sim$ are the candidate types) in the above type. This is achieved by extending the polytype grammar:

$\langle poly \rangle ::= \ldots$
$\quad | \quad \langle predicates \rangle\ \langle poly \rangle$

where $\langle predicates \rangle$ is all the *oc* predicates on the polytype $\langle poly \rangle$.

Given the polymorphic nature of `extract`, both the following functions must be legal:

```
foo9({bwd,A}) -> A;
foo9({fwd,A}) -> A;
foo9(T) -> extract(T).

foo10({nil}) -> [];
foo10(T) -> [extract(T)].

...> c(test).
{foo9,1} :: ∀A[].(dir A)->A
{foo10,1} :: ∀A[].(maybe_either A)->List A
```

As expected, our current implementation leverages the information from the surrounding unification constraints. In `foo10`, from the pattern match on {`nil`} we know that the `foo10` is applied to a `maybe_either` value, and hence the type of `extract` is specialized to `maybe_either`. Similarly, in {`foo9`} we infer that `extract` is applied to a `dir` value. On the other hand, the following program must be rejected since it is neither a `dir` nor a `maybe_either` value:

```
% polymorphic extract
86. extract({left,A}) -> A;
87. extract({right,A}) -> A.

96. foo11() -> extract(true).

...> c(test).
test.erl: error in parse transform 'etc':
{"Type Error: No matching constructors for
{funt,86,[{tvar,96,A}],{bt,96,boolean}}",
```

The type checker identifies that there is something wrong as it is unable to find a unifiable candidate for the type `(A)->boolean`. The current error message is cryptic as it simply displays the type in the unsolved *oc* predicate. The implementation is far from complete.

3.6.4 Current issues

In the presence of useful restrictive information, the types of the constructors are specialized as expected. But the above discussed solution alone does not work with generic code very well. For example, consider the following program:

```
-type either(A) :: {left,A} | {right, A}.
-type dir(A) :: {left,A} | {right, A}
    | {fwd,A} | {bwd,A}.
```

```
extract({left,A}) -> A;
extract({right,A}) -> A.

foo12() -> extract({left,true}).
```

We would expect this `foo12()` to be assigned the type `()->boolean`. But this is a lot less obvious than it seems. It is currently assigned the following type:

```
...> c(test).
{extract,1} :: ∀A[].∀B[].
        (A)->B ∼ {(A)->dir A , (A)->either A}
        (A)->B ∼ {(A)->dir A , (A)->either A}
    (B)->A
{foo,0} :: ∀A[].∀B[].
        (A)->B ∼ {(A)->dir A , (A)->either A}
        (A)->B ∼ {(A)->dir A , (A)->either A}
        (boolean)->B ∼ {(boolean)->dir boolean ,
            (boolean)->either boolean}
    ()->A
```

This means that `foo` has type `()->A` where `(A)->B` must unify with `(A)->dir A` or `(A)->dir A`, and `(boolean)->B` must unify with `(boolean)->dir boolean` or `(boolean)->dir boolean`. When we work out the unifying substitutions for all possible cases, we can notice that all successful unifications yield a substitution containing $\{A \mapsto boolean\}$. And this is also what we want in our type!

Hence, one solution to this problem is to: calculate all possible unifying substitutions and take an "intersection" of them. Since they occur on all possible paths, the solution cannot be incorrect. In this case, after the intersection $\{A \mapsto boolean\}$ has been calculated, the predicates can be thrown away as they do not apply on any type variable in the resulting type `()->boolean`. One evident performance problem with this solution is the exponential blow up in search space: the number of unifications to be performed can grow very rapidly. The treatment of this problem requires further investigation.

Another issue with the current implementation is that class predicates and *oc* predicates don't mix well with each other. For example, the following program - which is evidently not well typed - type checks:

```
foo12() -> extract(1).

...> c(test).
{foo12,0} :: ∀A[].∀B[Num B;].
        (A)->∼ ∼ {(A)->dir A , (A)->either A}
        (A)->B ∼ {(A)->dir A , (A)->either A}
    ()->A
```

It's not hard to work out manually that `B` - irrespective of being a `dir A` or `either A` - cannot be an instance of `Num`. But it's also not obvious how this is to be implemented for general cases.

4 Related Work

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

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

5 Reflections

5.1 Open goals

More examples of type-checked programs can be found in the Appendix. The current type system only works for single-module toy Erlang programs and is far from complete. It does not support modules, the ERTS (Erlang Run Time System), OTP libraries or messaging. To type check any real world Erlang program all of these would need to be implemented.

References

[Dam84] Luis Damas. Type assignment in programming languages. 1984.

[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.

[Hin69] Roger Hindley. The principle type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[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.

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

[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.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

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

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[Ran12] Aarne Ranta. *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012.

6 Appendix

6.1 Types of built-in operators

```
'+' :: ∀A[Num A;].(A,A)->A
'-' :: ∀A[Num A;].(A,A)->A
'*' :: ∀A[Num A;].(A,A)->A
'/' :: ∀A[Num A;].(A,A)->A
'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,B)->boolean
'/=' :: (A,B)->boolean
'=<' :: (A,B)->boolean
'<'  :: (A,B)->boolean
'>=' :: (A,B)->boolean
'>'  :: (A,B)->boolean
'=:=' :: (A,B)->boolean
'=/=' :: (A,B)->boolean
'++'  :: ∀A[].(List A,List A)->List A
'--'  :: ∀A[].(List A,List A)->List A
```

6.2 Examples

6.2.1 Tree (Continued)

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

nT({node,_,Lt,Rt},N) ->
    {Lt_,X} = nT(Lt,N+1),
    {Rt_,Y} = nT(Rt,X+1),
    {{node,N,Lt_,Rt_},Y}.
nT({nil},N) -> {{nil},N}.

numberTree (T) ->  {TA,_} = nT(T,0), TA.

...> c(test).
{nT,2}::∀A[Num A..].∀B[].(tree B,A)->Tuple tree A A
{numberTree,1}::∀A[Num A..].∀B[].(tree B)->tree A
```

6.2.2 Recursion and lists

```
reverse(L) -> reverse(L,[]).

reverse([],R) -> R;
reverse([H|T],R) -> reverse(T,[H|R]).


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


tail_recursive_fib(N) ->
```

```
    tail_recursive_fib(N, 0, 1, []).

tail_recursive_fib(0, _Current, _Next, Fibs) ->
    reverse(Fibs);
tail_recursive_fib(N, Current, Next, Fibs) ->
    tail_recursive_fib(N - 1, Next,
        Current + Next, [Current|Fibs]).


sum(L) -> sum(L, 0).

sum([H|T], Sum) -> sum(T, Sum + H);
sum([], Sum)    -> Sum.

nubSorted([]) -> [];
nubSorted([X]) -> [X];
nubSorted([X|[X|Xs]]) -> nubSorted([X|Xs]);
nubSorted([X|Xs]) -> [X | nubSorted(Xs)].

find(X,[]) -> false;
find(X,[X|Xs]) -> true;
find(X,[_|Xs]) -> find(X,Xs).


qsort([]) -> [];
qsort([H | Xs]) ->
    {L, E, G} = partition(H, [H|Xs], {[], [], []}),
    qsort(L) ++ E ++ qsort(G).

partition(_, [], {L, E, G}) ->
    {L, E, G};
partition(Pivot, [H | List], {L, E, G})
    when Pivot > H ->
    partition(Pivot, List, {[H | L], E, G});
partition(Pivot, [H | List], {L, E, G})
    when Pivot < H ->
    partition(Pivot, List, {L, E, [H | G]});
partition(Pivot, [H | List], {L, E, G})
    when Pivot == H ->
    partition(Pivot, List, {L, [H | E], G}).

...> c(test).
{append,2} :: ∀A[].(List A,List A)->List A
{sum,2} :: ∀A[Num A;].(List A,A)->A
{nubSorted,1} :: ∀A[].(List A)->List A
{find,2} :: ∀A[].(A,List A)->boolean
{sum,1} :: ∀A[Num A;Num A;].(List A)->A
{reverse,2} :: ∀A[].(List A,List A)->List A
{reverse,1} :: ∀A[].(List A)->List A
{tail_recursive_fib,4} ::
```

```
    ∀A[Num A;].∀B[Num B;].∀C
    [Num C;Num C;Num C;].
    (C,A,A,List A)->List A
{tail_recursive_fib,1} ::
    ∀A[Num A;].∀B[Num B;Num B;Num B;].
    ∀C[Num C;Num C;Num C;].
    (C)->List B
{qsort,1} :: ∀A[].(List A)->List A
{partition,3} :: ∀A[].∀B[].
    (B,List A,Tuple List A List A List A) ->
    Tuple List A List A List A
```

6.2.3 Overloaded constructors

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

empty () -> {nil}.

ex0 () -> {node, 1, empty (),empty ()}.

ex1 ({nil}) -> false;
ex1({just,_}) -> true.

ex2({nil}) -> false;
ex2({just,1}) -> true.


...> c(test)
{empty,0} ::
    ∀A[].∀B[].∀C[].∀D[].
    ()->A ∼ {()->tree B , ()->maybe C , ()->myList D}
    ()->A
{ex0,0} :: ∀A[Num A;].()->tree A
{ex1,1} :: ∀A[].(maybe A)->boolean
{ex2,1} :: ∀A[Num A;].∀B[Num B;].(maybe A)->boolean
```