Exference - Rubbing the Lamp once more

Lennart Spitzner

private work
aug-dec 2014

Abstract

Exference is a tool for automatically generating haskell expressions from a haskell type. It is certainly not the first of its kind: *Djinn* uses a theorem prover applied on type expressions, generating haskell expressions as a by-product; *MagicHaskeller* generates functions given a boolean predicate (think: test-driven generation) (and several more, especially for semantics-driven generation). *Exference* has some differing properties:

- no promises regarding termination (contrast to *Djinn*);
- no control over the semantics (test-cases, etc.) (contrast to *MagicHaskeller*);
- support a larger subset of the haskell type system (type classes including class hierarchies and polymorphic functions) (not supported by the other two);
- extensibility in the environment used; by default using a large subset of the haskell *Prelude* as the environment. Mainly, certain types of functions, like e.g. `id`, must be excluded, as they do not add new solutions while enlarging the search space significantly.

None of the three tools is better than the others in every respect. We will demonstrate some types of queries where *Exference* currently excels, and describe capabilities and limitations.

Motivation

A common meme with Haskell is: Once you fixed any type errors and the program compiles, you are done (and the product is ready for shipping)[citation needed]. If that was True, writing a function with a specified signature would require nothing more than finding any expression of that type. Unfortunately, it is not True. Nonetheless, we can make certain observations:

- There are many cases, especially for polymorphic functions, where the simplest (by some definition) implementation is the only “sensible” implementation of a type.
- Not rarely do haskell programmers “fight” with the compiler to finally get the expression (which has trivial, given the circumstances, semantics) to type-check.
- GHC 7.8 introduced *typed holes*. The programmer can insert these as a placeholder (similar to inserting `undefined`), and GHC will print the exact type expected for an expression filling the hole. The very existence of typed holes indicates that there are cases where the process of implementation can to a significant degree be driven by types.

We can give an example from the implementation of *Exference* (a touch of self-hosting, if you wish): When parsing the *Exference* dictionary (i.e. the set of functions, classes, and instances that the algorithm will "know about"), we transform the haskell syntax tree node (let us call the type `D`) to some internal representation (let us call it `FB`). For error handling and normalization purposes, things are wrapped a bit, more precisely:

- The input is a list of `D`'s;
- We want, in some way, use a transformation function having the type `D -> EitherT e (State s) [FB]`;
- We additionally want to transform the results with a given function `FB -> FB`.
- The result should have the type `State s [Either e [FB]]`

  For the purpose of this motivation we intentionally leave the background of this problem definition vague (e.g. what the `FB->FB` function does).

Essentially we are looking for a (specific) expression of type:

```
   [D]
-> (D -> EitherT e (State s) [FB])
-> (FB -> FB)
-> State s [Either e [FB]]
```

A somewhat experienced haskell programmer might be able to directly tell "oh, I will need `mapM`!" or the like; continue a few more steps; and find, in short time, a short solution. But the effort is certainly non-zero and - in our experience - often some "dialogue" with the type-checker/compiler is necessary.

Yet what we do is nothing more than looking at the expected types, finding appropriate generic functions (like `fmap`, `sequence`, `runEitherT`, `mapM`, ..) and combining them in a way that type-checks. Sometimes, there in one further requirement: That we use all available input, e.g. the parameter with type `FB -> FB` above.[1]

Automatizing this purely type-driven process is exactly the purpose of *Exference*. If we use the full type as input, the tool consumes some CPU seconds[2] and produces:

```
\ b c d ->
  mapM (\ h -> runEitherT (d h >>= (\ n -> pure (fmap b n)))) c
```

Which would be fine; with the right flags (and some more CPU seconds[3]) a yet simpler solution

[1] We leave it to the reader to realize that the solutions below can be simplified if the parameter `d :: FB -> FB` can be ignored.

[2] ~10 seconds for the *first* solution on a single 3 GHz Intel CPU core.

[3] To give the reader some idea: The first solution is found after ~30000 steps. The *better* solution is found after ~100000 steps and ~30 seconds. If we replace `State s` by `Monad m => m`, the search space becomes much smaller: The first solution is found after ~3500 steps and ~2 seconds, the best solution is found after ~9000 steps and ~4 seconds. But, admittedly, the user would not know beforehand how many steps would be necessary to find a better solution..

All times measured on a single 3 GHz Intel CPU core.

```
\ b c d -> mapM (\ h -> runEitherT (fmap (fmap b) (d h))) c
```

can be found. Some notes:

- The current dictionary assumes the AMP[4] and consequently uses the more general `pure` rather than `return`.
- It might seem strange that a longer solution is found first. The reason is that, for best general performance, *Exference* uses some custom order to traverse the search space. In this custom order, the shorter solution does not always come first (See order of solutions).

A Closer Look at *MagicHaskeller* and *Djinn*

MagicHaskeller by Susumu Katayama

Unfortunately, the *MagicHaskeller* package[5] does not compile with GHC 7.6.3 or GHC 7.8.3. We tested some examples on the web-interface[7] which allows to make certain queries, but which is somewhat limited as the calculation has to run server-sided. For this tool, the user does not specify a type, but an expression that evaluates to `Bool`. This expression contains the function that is to be implemented (on the website, it must be named “f”). *MagicHaskeller* infers the type of “f”, generates a series of programs of that type, and filters the list of programs by testing the given predicate (i.e. evaluating the expression provided).

```
> f "abcde" 2 == "aabbccddee"                       --input
f   =   (\a b -> concat (transpose (replicate b a))) --output
> f [ (+ 3) , (4 -) ] 5 == [8, -1]
f   =   (\a b -> map (\c -> c b) a)
> f [ ( + 1 ) , ( * 3 ) ] [ 2 , 6 ] ~= [ 3 , 18 ]
f   =   (flip (zipWith (\c d -> d c)))
> (   f Nothing              == Nothing
   && f ( Just Nothing )     == Nothing
   && f ( Just ( Just 3 ) ) == Just 3 )
f   =   (maybe Nothing id)
```

Understandably, there are some restrictions, because otherwise the search space becomes large easily. For example, the algorithm will not invent certain constants, and it does not support type classes (or more generally, polymorphism):

```
-- expectation: f = length . filter (=='a')
> (   f ""        == 0 && f "baaa"    == 3
   && f "babaaab" == 4 && f "bbabba"  == 2 )
[no results found]
```

[4] Applicative-Monad-Proposal, see [1].

[5] See relevant papers[2–5]; hackage package[6]; description refers to version 0.9.1

```
-- expectation: f = pure (or return)
> f 1 == Just 1 && f 1 == [ 1 ]
Couldn't match expected type ...

-- expectation: f = length
> f "abc" == 3 && f [ True ] == 1
Couldn't match expected type ...
```

This might be a problem of the interface, though.

–

In summary, *MagicHaskeller* provides the ability not only to generate an implementation from a given type, but also to check the semantics. This is not possible with *Djinn* or *Exference*. The interface, where the input is an expression (not a type as for *Djinn*/*Exference*) means that it is not possible to make certain types of polymorphic requests (such as the one we discussed in the motivation). But this is understandable, as polymorphism generally enlarges the search space, making a hard problem even harder.

Djinn by Lennart Augustsson

"*Djinn* uses an theorem prover for intuitionistic propositional logic to generate a Haskell expression when given a type."[6] If we omit the theory, we could paraphrase "input is a type, output is an expression with that type". However if you are interested, Dan Piponi[10] gives a nice introduction to the underlying mathematical concepts[7].

Djinn really takes a bit more than a type as input: there is an "environment" containing data type, type class and function definitions known to the algorithm when making queries. The default environment is rather small (but the user is free to provide a different one):

```
data Either a b = Left a | Right b
data Maybe a = Nothing | Just a
data Bool = False | True
data Void
type Not x = x -> Void
class Monad m where return :: a -> m a;
                    (>>=) :: m a -> (a -> m b) -> m b
class Eq a where (==) :: a -> a -> Bool
```

Some small example queries with the default environment:

```
> leftToMaybe ? :: Either a b -> Maybe a  -- input
leftToMaybe :: Either a b -> Maybe a      -- output
leftToMaybe a =                                   .
    case a of                                     .
```

[6]hackage package[8]; also see author's announcement [9]; description refers to version 2011.7.23

[7]and do not be discouraged; what we know of the theory we learned only *after* implementing the core of *Exference*.

```
    Left b -> Just b                             .
    Right _ -> Nothing                           -
> impossible ? a -> b                       -- input
-- impossible cannot be realized.           -- output
```

The second example shows what happens when there is no solution possible. *Djinn* always terminates, either with a solution or by determining that the solution is not possible. This can only be guaranteed by imposing certain restrictions on the elements of the dictionary and the class of solutions found.

We want to describe good and bad properties in more detail, and biased as we are, we will start with the criticism:

Polymorphic functions The main limitation of *Djinn* is the inability to unify polymorphic function types from the environment. *Djinn*'s own documentation describes this[8]: "Djinn does *not* instantiate polymorphic functions. It will only use the function with exactly the given type."

With a "lucky" choice of type variables, stuff seems to work:

```
> r ? Monad m => a -> m a
r = return
```

but otherwise:

```
> r ? Monad m => b -> m b
-- r cannot be realized
> r ? Monad n => a -> n a
-- r cannot be realized
```

In appendix 2 we describe in a bit more detail how polymorphic functions are treated in *Djinn*.

On the other hand, type variables in data types are handled just fine, i.e. they can properly be instantiated. The `instance Monad Maybe` example showed this already.

Class contexts It is not possible to specify super-classes in the class definitions in the dictionary, so there is no way to have, for example, a query `Applicative f => (a->b) -> f a -> f b` return `fmap`.

Data type declarations It is not possible to make data types available in an opaque fashion, i.e. to define a new data type without exposing its constructor. Any data type declared without parameters, e.g. simply `data A` or `data D a` is considered equal to a `Void` data type:

```
data A
data B
foo ? A -> B
```

[8]cited from the help text when using the ":verboseHelp" command in the *Djinn* REPL.

Djinn will happily return `foo x = x`.

Pattern matching When a parameter is a data type (other than Void), *Djinn* will use pattern matching to extract the constructor's parameters. This was already shown above in the `leftToMaybe` implementation, where the `Either`-value was deconstructed. To ensure that any solution is found, pattern matching will never be omitted, even if it is unnecessary (see appendix 2 for an example); in some cases unnecessary pattern matching is "optimized away".

Data type definitions There is a restriction on the form of data type declarations: They may not be recursive, as in "you can not use the type in the constructors of that type". Examples for recursive data types are lists or trees.

The simple reason for this restriction is that one can pattern match indefinitely often on a value of an recursive data type, thus *Djinn*'s strategy of pattern-matching eagerly would not work:

```
foo :: [Bool] -> Bool
foo xs = case xs of
  [] -> e0
  (x1:xs1) -> case xss of
    [] -> e1;
    (x2:xs2) -> [..] -- indefinitely
```

To be fair, there is no way to ensure termination with recursive data types, as instances of the undecidable Post correspondence problem could be expressed as *Djinn* environments, see appendix 2

Type class instantiation *Djinn* can automatically derive certain instances for type-classes. The default example is `instance Monad Maybe`:

```
> ? instance Monad Maybe
instance Monad Maybe where
   return = Just
   (>>=) a b =
        case a of
        Nothing -> Nothing
        Just c -> b c
```

This is a neat feature, but not something fundamentally new: The algorithm just looks at the type of the class's methods and runs the simple type-to-expression query for each one.

Tooling support *Djinn* can be started non-interactively (with a file - containing both an environment and queries - given as input) or interactively as a REPL where the user can modify the environment and make queries. *Djinn* can be asked to return more than one solution. Furthermore, there are some packages that provide an interface to *Djinn* from other contexts. Package `djinn-ghc` targets the GHC API, while `djinn-th` allows the usage of *Djinn* from template haskell code.

–

In summary, *Djinn* has one defining property: The ability to always either find a solution or to state the impossibility of a solution. But this comes at the cost of rather limited functionality. To some degree, *Djinn* only pretends to support type variables. A reply "foo cannot be realized" means surprisingly little for polymorphic queries.

Exference Overview

The fundamental user-interface of *Exference*[9] is similar to that of *Djinn*. The user makes queries containing a type expression, and *Exference* either provides a suitable implementation or prints `[no result]`. As *Exference* tries to find solutions to an undecidable problem[10], here this means "no result found in the given (time-/step-) limits".

Exference uses an environment that serves as additional input (in conjunction with the actual query) to the core algorithm. Internally, this environment comprises:

- a dictionary containing functions, data types, type classes and type class instances;
- a list of function *penalties* that determine how solutions containing these functions will be rated during the process. These penalties give the user some control over the order in which the search space is traversed;
- a list of constants that determine how the intermediate states are rated internally. Similar to the penalties, the purpose is to improve the order of the search tree traversal to (quicker) find the most sensible solution.

Currently the first two items are read in at run-time; the third item is encoded as record containing a list of `Floats` and requires recompilation for changes. The dictionary is read from a file containing a syntactically valid haskell module (the functions are only declared, not defined, so it would not compile). The penalties are read from a file containing "function-name = float" pairs.

Algorithm Capabilities and Limitations

+ Polymorphism The algorithm fully supports type variables and unification.

+ Undefined data types *Exference* can cope with data types that have not been defined.

+ Type classes / Instances / Hierarchies The dictionary contains the declaration of the type classes and their instances. Classes have contexts; i.e. `Monad` is defined as

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

And if the environment defines `instance Monad (State s)`, `State s` will be instance of `Functor`, `Applicative` and `Monad`.

[9]The description refers to version exference-0.8.0.0, exference-core-0.14.0.0

[10]See the notes to draft proof for undecidability in appendix 2

We have seen these three first properties in action in the example in the motivation. Another trivial, but fancy use-case is the conversion of numerical types (our dictionary contains the necessary instances already):

```
> Int -> Float
realToFrac
> Double -> Integer
truncate
round
ceiling
floor
```

- **Kind checking** While being able to process undefined types might appear to be nice feature, it is more the result of a non-feature: The kinds of types are not checked, not even for data types in the dictionary. For example, we can write

  ```
  > State s a -> State a
  [no result]
  ```

  That is, there is no error of any kind; it even tries to find a solution up to the time limitation.[11]

+ **Pattern matching on (non-recursive) single-constructor data types** For any data type with a single constructor, values will be pattern-matched to extract the parameters. Unfortunately,

- **Pattern matching on multiple-constructor data types** E.g.

  ```
  > Either a b -> Maybe a
  [no result]
  ```

  Adding this is no hard problem, just would need some time. The only issue is the one discussed already for *Djinn*: We need to take care not to pattern-match indefinitely on recursive data types like lists. Solutions are to either restrict pattern matching to non-recursive data types[12], or to make pattern matching an explicit step in the process (for recursive data types).

+ **Provide flags to allow/forbid solutions leaving bindings unused** By default, only solutions that consume all input are allowed (i.e. where any defined variable is used at least once, and no arguments are ignored).

  ```
  > Monoid a => a -> a -> a -> a
  \ b c d -> mappend d (mappend c b)
  ```

[11] This deficit is easy enough to fix; even retaining the ability to handle undefined data types should be possible. But whether we want both features is somewhat questionable and depends on the use-case: If the input comes directly from a human, more checks might be better, including the necessity to define data types in order to detect misspelling in queries;
if the input comes from the compiler (i.e. if we just forward the type of a typed hole to *Exference* to generate an expression to fill the hole), all checks are unnecessary (and unknown types should be allowed).

[12] however this would need detection of mutual recursion

On the other hand, certain solutions are not possible with such a restriction:

```
> (a,b) -> a
[no results]
```

But the user can enable a certain flag to allow unused binds, so that for the last request the expected solution `\ b -> let ((,) d e) = b in d` is found. In appendix 3 we discuss another specific case where neither approach gives the desired result.

Note that *Djinn* tends to "prefer" solutions using all input as well, but certainly is not strict about it: Try to give it `(a -> a -> a) -> a -> a -> a -> a` as input.

- **Type aliases and class instance implementation ("interface candy")** *Exference* currently does not understand type aliases (e.g. no result for query `String -> [Char]`); nor does it support deriving a full type class instance in the way *Djinn* does. This is not a problem of the core algorithm: All synonyms could be expanded (in both the dictionary and the query) *before* running the algorithm. In theory, the solution to both issues seems trivial.

◦ **Type-level extensions** Multi-parameter typeclasses pose no problem, but RankNTypes, GADTs and other advanced type-level extensions are not supported. Also, the *Exference* parser currently does not handle certain cases, such as contexts in data types, existential data types, infix data constructors, records or kind annotations.

- **Performance** Both time and space requirements are large. The reason is the exponential growth of the search space in the complexity of the expressions considered. Low-level optimizations or parallelization will only be able to speed up by constant factors, so their merit is fairly limited. The most promising approach is to improve the search strategy, i.e. the order in which the search space is traversed. That is why we focus on tweaking the parameters (certain constants and per-function penalties) that determine the order.

  Currently we use one static set of these parameters for all the test-cases, with sufficiently good performance results. But for new types of queries, it may be necessary to tweak them in order to achieve reasonable performance.

  Memory performance is also problematic: The program has a maximum residency in the hundreds of MB, and our recommendation for the minimum heap size setting is 2GB (in order to prevent 90% of CPU time being wasted in garbage collector runs).

  In appendix 3 we describe the unsatisfactory results of trying to parallelize the algorithm.

- **Order over search space** The order in which solutions are found is not formally defined; it is mostly guided by what tweaking seemed to give best performance. This means that it is hard to predict if a "desired" solution will be found first.

◦ **Simplifications** As mentioned above, given as input the type of `join`, the result would be: `\b -> ((>>=) b) (\f -> f)`. This can be simplified to `(>>= id)`. This kind of simplification is not the concern of *Exference*; there are other tools such as the `pointfree` program[11] that can serve such a purpose.[13]

[13] One thing, however, would be neat: A program that transforms not to a completely point-free version (like pointfree), but to a sensible level of "point-free-ness". We are not aware of any tool that does this.

The Dictionary

The current dictionary used for testing purposes contains ~16 function bindings from the `Prelude`, ~10 functions from `Control.Monad`, and all the type classes and their member functions from the `Prelude`. This accumulates to a total of ~18 type classes, ~70 functions and ~750 type class instances. The instances were mostly just copied over from the corresponding haddock.

Lists and tuples are currently added to the dictionary automatically as they have special syntax.

Composition

There are certain reasons for excluding specific functions to the dictionary:

1) Functions that implicitly ignore input. For example `fst`, `(>>)`, anything ending with `_` (`mapM_`, `sequence_`, ..). If we allow any such function in the dictionary, the algorithm tends to "abuse" it in order to find a solution that use-but-not-really certain variables even when we want all variables to be used.
2) Certain functions do not add anything to the solution, but enlarge the search space. Examples are `id`, `negate` or `fst` (because we pattern-match on tuples anyway).
3) The more polymorphic the return type, the worse. If the return type is a (constrained) type variable, the function will unify in a lot of cases and enlarges the search space a lot. Examples are `mappend` or `realToFrac`.
4) Functions that are just flipped versions of other functions already in the dictionary, e.g. `forM` to `mapM`.

Penalties

Sometimes it is advantageous to have specific penalties for certain functions to get results in a reasonable time-frame. A higher penalty value means "rather do not use this function". For advanced debugging, the search tree can be printed to see which function applications form large sub-trees in the search-space without yielding any solutions. For these functions, the penalty can be increased. We use a test suite to ensure that changing penalties does not invalidate correct solutions.

For certain functions (most prominently, `(>>=)`) we use a *negative* penalty to increase the likelihood of usage. As many examples use monads and thus make rather frequent use of `(>>=)`, this improves the average run-time a lot.

A negative side-effect of large differences in penalties is that functions that can be expressed in terms of other functions having better penalties may not be used at all. We saw that already when the algorithm chose the `(>>=)/pure` combination instead of an `fmap`.

Large negative penalties are bad, as they can distort the search order over the search space in a way that resembles a depth-first search - the algorithm effectively creates larger and larger expressions containing the negative-rating functions, without ever considering anything else (and without finding solutions).

Implementation

The core idea of the algorithm is to incrementally build a (partial) expression that is type-correct but contains “holes”. There are multiple ways to fill a hole and filling a hole can create multiple new holes; so the search is not linear: Our search space is a tree of expressions.

We will first describe the conceptual search space, then give the actual process for the traversal of the tree, and lastly explain how type-correctness is ensured.

The Search Tree

Each node in the search tree is a closed[14] partial expression. A partial expression is a normal haskell expression that can contain any number of “holes”, denoted as `_`, at any location in the expression tree. Holes have a type but not yet an implementation.

The root of the tree is just a hole. All children of a node replace a specific hole (the first by some order over the holes) in the parent’s expression with a new partial expression. Thus any node without holes is a leaf (and a potential solution).

Currently, a new expression filling a hole can have one of the following forms:

1)	`\a -> _`	where `a` is a new variable
2)	`x`	where `x` is a value from the dictionary or variable in scope.
3)	`f _ ... _`	where f is a value or variable of function type (number of holes $n \geq 1$)
4)	`let` `(C` $a_1 \dots a_n$ `) = x` `in _`	where `C` is a constructor, where $a_1 \dots a_n$ are new variables, where `x` is a variable in scope ($n \geq 1$)
5)	`let` `a = f _` `in _`	where `a` is a new variable, where `f` is a value or variable of function type,

These transformations are either mandatory (lambda transformation (1) and pattern matching (4)) or guided by the type expected for the hole (atom (2) and application (3)). The “random” application (5) is less often necessary; it falls in neither category.

A (simplified) search tree for the query `Monoid a => a -> a -> a` (with types of holes annotated, but ignoring the constraints) is:

```
+           (_ :: a -> a -> a)
\-+         \x -> (_ :: a -> a)
  \-+       \x y -> (_ :: a)
    |-+     \x y -> x                       -- potential solution
    |-+     \x y -> y                       -- potential solution
    |-+     \x y -> mzero                   -- potential solution
```

[14] No free variables (functions from dictionary do not count)

```
\-+     \x y -> mappend (_::a) (_::a)
  |-+   \x y -> mappend x (_::a)
  | |-+ \x y -> mappend x y            -- potential solution
  | [..]
  |-+   \x y -> mappend y (_::a)
  | |-+ \x y -> mappend y x            -- potential solution
  | [..]
  [..] -- (more combinations with mzero's or mappend's)
```

Practically, the nodes in the search tree are more complex; they contain some additional data like the type-class constraints that must be met and a `Float` describing the state's "complexity".

Potential solutions are filtered in two ways: Firstly we ensure that all type-class constraints are provable. Secondly we check that all variables are used at least once. The latter constraint can be omitted with a flag, though.

The Traversal

We use a best-first-search, using a heuristic evaluation function `rating :: NodeState -> Float`, implemented using a priority-queue of `NodeState`s. Each step, we take the first element of the queue, calculate the set of child-nodes, and re-insert those into the queue or return them as potential solutions.

The search order is influenced by two components: By the choice of the order of the holes in the expressions and by the rating function.

The Order of Holes in Expressions

How we insert new holes in the expression filling the hole depends on the kind of step performed:

1) The new hole is inserted at the *front*;
2) (no new hole;)
3) The new holes are inserted at the *back*;
4) The new hole is inserted at the *front*;
5) The parameter hole is inserted at the *back*, the "in-hole" at the *front*.

The main reason for this choice is that it gave the best results - the solutions are found faster. We did not put time into analyzing the reasons for this behavior.

The Rating Function

The rating function is calculated from two components:

1) The "stateful" complexity rating at each node; this node complexity rating monotonically increases when moving down the search tree, depending on the modification performed at each step;

2) A rating of different parts of the node state, such as

   - the complexity of the holes,
   - the number of type class constraints to prove,
   - usage statistics for the variables in the expression (exactly one usage is generally best).

   This rating is not monotonic.

Both components have a lower limit. This means (considering the monotonically increasing component) that the ratings generally get worse the deeper the node is in the search tree; the search should yield the same set of solutions as a breadth-first search would. But the ratings can favor areas of the search tree.

The Order of Solutions

We can consider three orders:

1) The order in which the algorithm returns solutions is a total one;
2) We retrieve a semi-order over solutions using the rating function (which maps solutions to `Float`). In general it is different from the previous (even when we ignore the semi-versus-total part);
3) An informal order over solutions is given by which solutions have the correct semantics (for some definition of "correct") and are the most beautiful (yes, we said "informal").

Order (2) should capture (3) as closely as possible. Preferably (1) should match (2), but the main goal is that $best_2$ either is $best_1$ or at least one of the first results found.

Orders (1) and (2) being distinct explains why sometimes searching past the first found solution yields better results. The reason for (1) and (2) being distinct is that there may be nodes in the path of $best_2$ which have a unfavorable rating (for example, there may temporarily be more holes in the partial expression). We *could* define the rating of intermediate nodes in a way so that (1) and (2) become equal, but that gave significantly worse performance in our experience.

Type-Correctness

The algorithm considers two issue separately: the checking of constraint-free types and the checking of type constraints. We will apply the same separation for this description.

Types Without Constraints

The algorithm basically is an inversion of the type checking algorithm by Milner[12]: Instead of determining if a given expression is well-typed, we obtain a list of (partial) expressions that are well-typed. We will now go into a bit more detail but will not provide a proof of correctness. Firstly, some definitions:

```
result :: TypeExpression -> TypeExpression
result (a->b) = result b
result x      = x

params :: TypeExpression -> [TypeExpression]
params (a->b) = a : params b
params _      = []

shift :: TypeExpression -> TypeExpression
shift replaced all type variables by new ones, so that they are
distinct from any variables for any holes in the current
expression.

unify :: TypeExpression -> TypeExpression -> Maybe Substitutions
unify is the usual type unification function
```

What happens at each step of the algorithm? Let `e` be the expression in the current search tree node. `e` contains a non-empty list of holes, each with a type. In this step the first hole will be filled in some way. Let `t` be the type of the first hole. If `t` has the form `a -> b`, insert a lambda expression `\x -> _`, where `x :: a` is a new variable, and `_ :: b` is a new hole[15].

Otherwise (if `t` is not of the form `a -> b`), we consider possible values to insert: Either variables in scope or values from the dictionary. For each such value `v :: vt`, we test applicability by evaluating

```
shifted = shift vt
maybeSubsts = unify t (result shifted)
```

If maybeSubsts is `Nothing`, no application is possible. Otherwise, we replace the hole with `v _ .. _` where we create one new hole for each `p` in `params shifted`. Finally, we apply the substitution on *all* the type expressions (i.e. in the types for all holes of the current expression). Consider a dictionary containing

```
createA :: A X Y
createB :: B a a
createC :: A a b -> B a b -> C
```

and the query for type `C`. After one step, the node with the partial expression `createC (_::A a b) (_::B a b)` is encountered. When we fill one of the holes with `createA` or `createB`, It is important that the substitutions are applied globally, yielding either `createC createA (_::B X Y)` or `createC (_::A c c) createB`. In either case, no solution will be found because `X` and `Y` do not unify.

Another interesting query is `a -> b`. Note that `a` and `b` do unify, but the result `\ b -> b` would be wrong. The query, more verbosely, is `forall a b . a -> b`. So we must ensure that the type variables in the query do *not* unify. Effectively, the type variables in the query *are not to be treated as variables.* This is easy to achieve: Only in the query's type expression we treat type variables as data types. This sounds like *Djinn*'s approach, but it is not: Type variables in types of functions from the dictionary are correctly treated as variables.

[15]the order of holes is determined by how we insert the new hole, but this is irrelevant for type-correctness

Checking the Constraints

We use the same algorithm as the haskell compiler as described by M.P. Jones[13]. Conceptually, the checking is performed whenever a potential solution is found; the relevant constraints are accumulated during the search. So the relevant steps are:

1) First of all, we note the initial constraints of the query (such as `Functor f` for the query `Functor f => f Int -> f String`). These "given" constraints will not be modified;
2) We start with an empty set of unproven constraints;
3) Whenever a function with a constrained type is inserted, we add the relevant constraint(s) to the set. E.g. when inserting `fmap` we add a constraint `Functor` ϕ for the appropriate type variable ϕ;
4) When type substitutions are applied globally, the type variables in unproven constraints are affected as well;
5) When the process reaches a potential solution - i.e. when there are no holes left in the expression - we try to prove all the (yet) unproven constraints using
    a) the type class relations and the type class instances in the dictionary;
    b) the given constraints.

As always we attempt to reduce the size of the search space: After steps (3) and (4) we search for unproven constraints that we can determine to be non-provable. For this purpose, we consider all constraints that do *not* contain type variables. These will not be affected by future steps, and thus are either non-provable (and we can abort the search below this node) or proven (and we can remove it from the set).

Future Work

There are certain functional deficiencies that are not hard to fix:

- Support for type synonyms
- Pattern-matching on multi-constructor data types
- Kind checking

For extensions of the type system (like RankNTypes, GADTs, TypeFamilies) we currently will not make promises if adding support for them is feasible.

On the side of performance, there probably is no limit for improvements, but few will have a big impact. The most promising idea we can think of is the following:

> For a partial expression of the form "f _a _b" the algorithm will, for each solution found in the process for _a, execute the search for _b independently. This is not necessary, or rather: the process can be optimized to a large degree. Let us assume there are two solutions for _a and two solutions for _b. Then we *do* need to test all four combinations, because only one combination of solutions might be using all the input parameters. But we do *not* need to generate these solutions multiple times. This distinction is important, because the number of steps performed during a search is much larger than the number of solutions found.

Conclusion

Exference can be used to generate expressions from types. We have provided some exemplary queries where *Exference* is able to find the expected solutions, where other tools such as *MagicHaskeller* and *Djinn* are unable to do so.

We gave an overview of these three programs. Each program has certain functionality not provided by the others; we highlighted the features in order to give the reader an idea of the relative advantages: *MagicHaskeller* allows semantic predicates, *Djinn* promises termination and is well-embedded in the ecosystem, and *Exference* fully supports polymorphic functions and type classes.

It is no surprise that the main problem of the approach chosen by *Exference* is the size of the search space. It is indicative that finding the solution for the example query from the motivation has a run-time in the order of seconds, and a slightly more complex query can easily add a factor to the run-time. The strategy and proper tuning of the heuristics used internally allows *Exference* to find solutions for a certain set of queries, but it would be easy to find queries where no solution will be found without adapting the heuristics. On the other hand it should be noted that for all tests of performance we were able to use the same general configuration and an environment that contains a good amount of elements. We described one possible optimization that might be able to improve the performance significantly.

It remains to evaluate how *Exference* performs when project-specific bindings are added to the environment, and what range of (complexity of) queries *Exference* is suitable for.

There is one interesting use-case that *Exference* certainly is suitable for: It can be used as a search system similar to *hoogle*, providing answers that require knowledge about type class connections. Prime example is the request `Int -> Float` returning `realToFrac`, which has type `(Real a, Fractional b) => a -> b`. For any libraries providing large numbers of polymorphic methods/operators, *Exference* might be used as a type-class-aware search engine.

Appendix 1

Package Structure

The project is currently split into three packages:

exference-core	library	contains the algorithm and the data structures used internally
exference	library and executable	translates from and to haskell (haskell types to the type data structure used internally; pretty printer for the resulting expression) — provides some wrappers around the core function (to retrieve only the first or the best result) — contains the default environment and the default settings

exference-bot	executable	an IRC bot providing access to *Exference* functionality

(order reflects dependency between the packages)

Module Structure

In the first column, the common prefix "Language.Haskell.Exference" was abbreviated to "L.H.E". The "origin" column names the package providing the module ("e-core" for "exference-core", "exf" for "exference").

module	origin	description
L.H.ExferenceCore	e-core	exposes the core interface in an appropriate form (i.e. re-exporting selected elements from `Internal.*` modules)
L.H.Exference	exf	a wrapper around the `ExferenceCore` module; adding some extended functionality
L.H.E.Internal.*	e-core	non-exposed parts of the core algorithm, including certain data structures as well as the core process implementation
L.H.E.Type	e-core	representation of Haskell types
L.H.E.ConstrainedType	e-core	wrapper around type representation t allow contexts for types, i.e. `Monad m => m`
L.H.E.TypeClasses	e-core	data types and functions concerning type classes, instances etc.
L.H.E.ExferenceStats	e-core	some data types intended for capturing performance statistics for the core process
L.H.E.Expression	e-core	representation of Haskell expressions
L.H.E.FunctionBinding	e-core	data type used for the representation of the *Exference* environment
L.H.E.SearchTree	e-core	data structure used for capturing the search space of the core process
L.H.E.SimpleDict	exf	a very basic, hard-coded environment used for testing initially. Not really needed anymore.
L.H.E.ExpressionToHaskellSrc	exf	conversion between representations (*Exference* and haskell-src-exts)
L.H.E.TypeFromHaskellSrc	exf	conversion
L.H.E.BindingsFromHaskellSrc	exf	conversion
L.H.E.ClassEnvFromHaskellSrc	exf	conversion
L.H.E.EnvironmentParser	exf	wrapper for environment parsing/conversion

Appendix 2

Polymorphic Functions in *Djinn*

That *Djinn* does not instantiate polymorphic functions means that, effectively, type variables in function declarations are *no* variables, but instead act like *opaque data types* that need not be declared anywhere. The only variable property is that they may or may not, on a per-query basis, be treated as an instance of a type class (but only if the type class contained the same exact variable). So the correct way to read the environment and queries of *Djinn* is:

- read `foo :: a -> b` as

  ```
  data A -- opaque data types
  data B
  foo :: A -> B
  ```

- Contexts work as usual: Whether the query contains `Monad m =>` determines if `m` is instance of Monad. But the respective variable *must* be `m`, because Monad is defined with `class Monad m where ...`

Example of Unnecessary Pattern-matching in *Djinn*

```
> data A a = A a            -- some elements for environment
> data B b = B b
> foo :: A a -> B b
> bar ? A a -> b            -- the actual query
bar :: A a -> b             -- result
bar a =
       case a of                 -- unnecessary
       A b -> case foo (A b) of  -- (necessary)
              B c -> c
```

Sketch of a Proof of Undecidability with Recursive Data-Types

Claim: If there are recursive data types in an environment for *Djinn*, finding solutions (including pattern-matching on those recursive data types) for a query is an undecidable problem.

Concept: We encode instances of the undecidable Post correspondence problem in a *Djinn* environment containing recursive data types. Solutions to the query contained in the environment can be transformed back to solutions of the Post correspondence problem.

Consider the following instance of the Post correspondence problem:[16]

	1	2	3
α	a	ab	bba
β	baa	aa	bb

[16]Taken from [14]

We can encode these three pairs in three constructors of a recursive data type:

```
data PCP a b x y = PCP (PCP a b (a x) (b (a (a y))))
                       (PCP a b (a (b x)) (a (a y)))
                       (PCP a b (b (b (a x))) (b (b y)))
```

Note how the third and fourth parameter inside the first constructor, i.e. `a x` and `b (a (a y))`, correspond to "a" and "baa" in the instance. This data type is the core of the representation, but the full environment is somewhat larger:

```
data PCPStart a b x y = PCPStart (PCP a b (a x) (b (a (a y))))
                                 (PCP a b (a (b x)) (a (a y)))
                                 (PCP a b (b (b (a x))) (b (b y)))

data PCP a b x y = PCP (PCP a b (a x) (b (a (a y))))
                       (PCP a b (a (b x)) (a (a y)))
                       (PCP a b (b (b (a x))) (b (b y)))

reduceA :: PCP a b (a x) (a y) -> PCP a b x y
reduceB :: PCP a b (b x) (b y) -> PCP a b x y

solution ? PCPStart a b c c -> PCP a b c c
```

We need an almost-copy of the `PCP` data type that adds one level of indirection (`PCPStart` contains `PCP`s, `PCP` contains more `PCP`s). Without it, the query would be `solution ? PCP a b c c -> PCP a b c c` with the trivial solution `id` (corresponding to the empty sequence, by definition not a solution to the problem).

The (first/shortest) solution for the query in the last line is:

```
solution = reduceA
         . reduceA
         . reduceB
         . reduceB
         . reduceB
         . reduceA
         . reduceA
         . reduceB
         . reduceB
         . p3 . p2 . p3 . ps1
```

where `p1`, `p2`, `p3` are selectors for the `PCP` data, and `ps1` is the first selector for `PCPStart`. The order of selectors in the last line (3-2-3-1) directly describes the solution for the Post correspondence problem. For a complete proof, we would need to show next that the set of solutions of any instance of the Post correspondence problem is similar to the set of solutions for the query in the corresponding *Djinn* environment.

Further Notes:

1) From the claim follows that recursive data types are incompatible with *Djinn*'s general approach (solving a decidable problem).
2) The general idea does not require recursive data types; allowing "opaque" data types (i.e. where no constructor/pattern matching is possible) is sufficient, by manually providing the selectors (`p1`, `ps2`, ...). This is possible in *Exference*. Hence the problem *Exference* tries to find solutions for as well is undecidable.

Appendix 3

Allowing/Denying Unused Variables: a Hard Query

In most cases, there are much simpler solutions possible when allowing unused binds. There are types of queries where we would like a specific unused variables, e.g. `Monad m => m a -> m a -> m a`. We would like the solution `\ a b -> a >>= \_ -> b`[17]. The lambda argument is ignored, but when we allow unused variables, the solution `\ b c -> c` will be found instead. Currently, there is no nice solution to this problem.

Result of Parallelizing the Exference Algorithm

Unfortunately, our implementation does not yield sufficient improvements. For certain cases, we observed a 30% speedup with four cores instead of one, but for other cases, slowdowns by more than 100%.

1) On a high level, the search space traversal is intrinsically not parallelizable. The next node considered depends on the result of the previous node. It is possible to relax these dependencies between processings of nodes, but it has consequences:
   - When relaxing the dependency, we "relax" the search order, and effectively make the results non-deterministic (yuk).
   - The exact effects on the search order are hard to reason about.
2) The overhead of parallelization is large. This may be caused by our implementation; if the threads processed their parts of the search space more independently, the results might be better.

References

[1] haskellwiki, Functor-Applicative-Monad Proposal, https://www.haskell.org/haskellwiki/Functor-Applicative-Monad_Proposal.

[2] S. Katayama, Power of Brute-Force Search in Strongly-Typed Inductive Functional Programming Automation, PRICAI 2004: Trends in Artificial Intelligence, 8th Pacific Rim

[17] `(>>)` is not in the dictionary as it is a "bad" duplicate of `(>>=)`

International Conference on Artificial Intelligence, Auckland, New Zealand, August 9-13, 2004, Proceedings. (2004) 75–84. doi:10.1007/978-3-540-28633-2_10.

[3] S. Katayama, Systematic search for lambda expressions, Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, {TFP} 2005, Tallinn, Estonia, 23-24 September 2005. (2005) 111–126.

[4] S. Katayama, Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening, PRICAI 2008: Trends in Artificial Intelligence, 10th Pacific Rim International Conference on Artificial Intelligence, Hanoi, Vietnam, December 15-19, 2008. Proceedings. (2008) 199–210. doi:10.1007/978-3-540-89197-0_21.

[5] S. Katayama, An analytical inductive functional programming system that avoids unintended programs, Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, Philadelphia, Pennsylvania, USA, January 23-24, 2012. (2012) 43–52. doi:10.1145/2103746.2103758.

[6] S. Katayama, *MagicHaskeller* package on hackage, (2013). https://hackage.haskell.org/package/MagicHaskeller.

[7] S. Katayama, MagicHaskeller online access, http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html.

[8] L. Augustsson, *djinn* package on hackage, https://hackage.haskell.org/package/djinn.

[9] L. Augustsson, mailing list announcement for Djinn, http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747.

[10] D. Piponi, Adventures in Classical-Land, The Monad.Reader Issue 6. (2007) 17–34. https://www.haskell.org/wikiupload/1/14/TMR-Issue6.pdf.

[11] T. Jäger, *pointfree* package on hackage, https://hackage.haskell.org/package/pointfree.

[12] R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences. 17 (1978) 348–375.

[13] M.P. Jones, Typing Haskell in Haskell, Haskell Workshop. (1999). http://web.cecs.pdx.edu/~mpj/thih/.

[14] en.wikipedia.org, Post correspondence problem, http://en.wikipedia.org/wiki/Post_correspondence_problem.