English title

(Wymagający złamania wierszy
tytuł pracy w języku polskim)

Konrad Werbliński

Praca licencjacka

Promotor: dr Filip Sieczkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

27 sierpnia 2019

Abstract

. . .

. . .

Contents

Chapter 1

Introduction

We introduce Bestrafer - functional programming language utilizing a novel approach to GADTs proposed by Joshua Dunfield and Neelakantan R. Krishnaswami in the article "Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types"[1].

1.1 Our contributions

Extension of Dunfield's and Krishnaswami's system

We have fine-tuned their system for usage in practical and user-friendly programming language, by extending subtyping and inference and adding extra typechecking rules for typical language features like `if` and `let` expressions, operators and exception handling. We extended their system by adding user defined generalized algebraic data types. Our approach of creating two representations of each constructor (one for typechecking and one for type inference) in combination with bidirectionality of the original system provides full expressive power to the programmer, by enabling parametrization with big types.

Definition of a language and implementation

We created a language with modern syntax and practical features like exception handling, handful of primitive types and standard library with many builtin operators, rich base of IO, utlity, convertion and traditional FP functions. We implemented interter of a Bestrafer language in Haskell. We put a lot of effort into creating reliable, useful and easy to use implementation of our language. To ensure correctness of the implementation we created almost 440 unit tests for the type system. We payed extra attention to producing readable and helpful type checking error messages. For example:

```
prog.br:4:26
Couldn't match expected type 'Int' with actual type 'Float'
```

We are also providing hints for some more complex error cases:

```
sorting.br:17:3
Type variable not in scope: 'k'
While trying to subtype: '(exists k : N . (Vec k a))' < '(Vec n1~ a~)'
Hint: try using let to unpack '(exists k : N . (Vec k a))'
before using it in expression of type '(Vec n1~ a~)'
```

Finally, we provided handful of example Bestrafer codes which showcase language syntax and features.

Chapter 2

Language description

2.1 Language features

Bestrafer's syntax was designed to be concise, expressive, readable and beautiful. It was strongly influenced by Haskell, but modified to be indentation-insensitive for greater flexibility in writing beautiful code and ease of parsing.

```
//Single-line comment
/*
  Multi-line comment
*/

def fac :: Int -> Int
def fac 0 = 1
def fac n = n * fac (n - 1)

def ack :: Int -> Int -> Int
def ack m n = case (m, n) of
  | (0, n) -> n + 1
  | (m, 0) -> ack (m - 1) 1
  | (m, n) -> ack (m - 1) (ack m (n - 1))

def main :: ()
def main =
  printInt (fac 5) `seq`
  printInt (ack 3 1)
```

Our language supports Haskell-like top-level pattern matching in definitions. Type annotations for top-level definitions are obligatory due to bidirectionality of the type system. We use call-by-value evaluation strategy like many mainstream functional languages. Program is evaluated from the top to the bottom of the source file (with a minor subtlety broader described in the chapter 4). All of the definitions on the top-level are mutually recursive. One can also define nested functions using `rec` keyword.

```
def fib :: Int -> Int
def fib n =
  rec :: (Int, Int) -> Int -> Int :
    f lasts n = case n of
      | 0 -> fst lasts
      | n -> f (snd lasts, fst lasts + snd lasts) (n - 1)
  in f (0, 1) n
```

Bestrafer supports all of the typical IO operations including reading and writing files as well as parsing and printing values of primitive types from and to standard input-output. IO operations may be performed at any point in program, following the style of several languages in the ML family. It also supports exception handling with `error` keyword for throwing errors and `try-catch` block for catching user thrown (`RuntimeException`) and builtin (`IOException`, `ArithmeticException`) exceptions. We can use optional variable in exception pattern for extracting the error message.

```
def checkPassword :: String -> ()
def checkPassword s =
  if s == "Rammstein" then
    ()
  else
    error: "Password is incorrect"

def main :: ()
def main =
  try:
    let password = getLine () in
    checkPassword password ‘seq‘
    let x = readLnInt () in
    printInt (1000 / x) ‘seq‘
    let filename = getLine () in
    readFile filename |> putStrLn
  catch:
    | IOException e -> putStrLn e
    | ArithmeticException -> putStrLn "Division by zero"
    | RuntimeException e -> putStrLn e
    | Exception e -> putStrLn e
```

Bestrafer allows user to define his own generalized algebraic data types (GADTs) using the `data` keyword. There are two kinds of parameters in GADT definition:

- named (denoted with a name starting with `'` followed by a capital letter) which work exactly like parameters of standard algebraic data types in languages like Haskell or OCaml.
- unnamed (denoted with their kind: `*` or `N`) which may be set by the user to any type of the specified kind, thus providing GADT functionality.

```
data Maybe 'A where
  | Nothing :: Maybe 'A
  | Just :: 'A -> Maybe 'A
```

Our language also supports defining data types without value constructors, which may be used as annotations in GADTs, like types `Ok` and `Fail` used to annotate type `Either` in the following example.

```
data Ok
data Fail

data Either * 'A 'B where
  | Left  :: 'A -> Either Fail 'A 'B
  | Right :: 'B -> Either Ok 'A 'B
```

A flagship data type of Bestrafer language is a list indexed by its length, traditionally called `Vec`.

```
data Vec N 'A where
  | []  :: Vec 0 'A
  | (:) :: forall n : N . 'A -> Vec n 'A -> Vec (S n) 'A
```

Using the above definition we can write `map` function, which type encodes the proof that the resulting `Vec` has the same length as the input one.

```
def map :: forall n : N, a : *, b : * .
  (a -> b) ->
  Vec n a ->
  Vec n b
def map _ [] = []
def map f (x : xs) = f x : map f xs
```

To give programmer full flexibility and expressive power our language also has a standard non-indexed `List` data type.

```
data List 'A where
  | {}  :: List 'A
  | (;) :: 'A -> List 'A -> List 'A
```

Bestrafer also supports existential types, but unlike in Haskell and OCaml their usage is not tied to data types declarations. Instead they can be used freely like any other type constructor. The following implementation of a `filter` function (taken from Bestrafer's standard library) utilizes existential type to express the fact that we cannot predict length of the resulting `Vec`. We use `let` expression to unpack result of recursive call from the existential type, thus ensuring that the type variable describing length of `tail` is inserted to the context before the subtyping starts.

```
def filter :: forall n : N, a : * .
  (a -> Bool) ->
  Vec n a ->
  exists k : N . Vec k a
def filter _ [] = []
def filter p (x : xs) =
  let tail = filter p xs in
  if p x then
    x : tail
  else
    tail
```

Quantifiers are always explicit to enforce conscious kind specification and emphasize connection to a type theoretic core. To articulate this connection even more instead of writing `forall`, `exists` and `\x -> x` one can write ∀, ∃ and `λ x -> x`.

2.2 GADT examples

Matrix algebra

We can make a great use of Bestrafer's indexed `Vec` type to implement matrix algebra operations. Now the types provide the proof that the matrix operations that we defined produce results with correct dimensions and impose restriction on input arguments which ensures that they also have proper dimensions.

```
def mult :: forall n : N, m : N, k : N .
  Vec (S n) (Vec (S m) Int) ->
  Vec (S m) (Vec (S k) Int) ->
  Vec (S n) (Vec (S k) Int)
def mult a b = map ((flip multVec) b) a
def multVec :: forall n : N, m : N .
  Vec (S n) Int ->
  Vec (S n) (Vec (S m) Int) ->
  Vec (S m) Int
def multVec v m =
  map (foldl1 (\x y -> x + y))
  (map (zipWith (\x y -> x * y) v) (transpose m))

def transpose :: forall n : N, m : N .
  Vec (S n) (Vec (S m) Int) ->
  Vec (S m) (Vec (S n) Int)
def transpose matrix =
  let indices = mapi const (head matrix) in
  map (flip column matrix) indices

def column :: forall n : N, m : N .
  Int ->
  Vec (S n) (Vec (S m) Int) ->
  Vec (S n) Int
def column i = map (nth i)

def nth :: forall n : N, a : * . Int -> Vec (S n) a -> a
def nth 0 (x : xs) = x
def nth _ [x] = x
def nth n (x1 : x2 : xs) = nth (n - 1) (x2 : xs)
```

One could think that the above functions are only useful for some statically defined values, since we cannot predict dimensions of `Vec`s which come from IO. But, that is not true! We can use them in program that reads matrices from IO, but we have to prove, that we handle all cases of invalid input before passing it into our matrix algebra functions.

Statically typed printf function[2]

The well-known `printf` function from the C programming language, uses a string to provide formating of a printed text. However, this approach has a major drawback: formated arguments are not statically type checked. As a result of that, writing `printf("%d", 3.14);` will print meaningless int, without emiting any warning or error. That's where the GADTs come to the rescue. In the following example, we define `Format` data type which is used to express intended formating of a printed string. By chaining constructors together we define type of intended printing function, which is accumulated in unnamed parameter of the `Format` data type. When a value of the type `Format` is applied to the function `printf`, an appropriate printing function is built by step by step deconstruction of the `Format` value. By combining this approach with the function composition operator (.) (for writing more readable chains of constructors), we get a neat and type-safe way of pretty-printing values into the standard output.

```
data Format * where
  | Str :: forall a : * . Format a -> Format (String -> a)
  | Inr :: forall a : * . Format a -> Format (Int -> a)
  | Flt :: forall a : * . Format a -> Format (Float -> a)
  | Bl  :: forall a : * . Format a -> Format (Bool -> a)
  | Chr :: forall a : * . Format a -> Format (Char -> a)
  | Lit :: forall a : * . String -> Format a -> Format a
  | Eol :: forall a : * . Format a -> Format a
  | End :: Format ()

def printf ::  forall a : * . Format a -> a
def printf End = ()
def printf (Lit s format) = putStr s ‘seq‘ printf format
def printf (Eol format) = putStrLn "" ‘seq‘ printf format
def printf (Str format) =
  \x -> putStr x ‘seq‘ printf format
def printf (Inr format) =
  \x -> (putStr . intToString) x ‘seq‘ printf format
def printf (Flt format) =
  \x -> (putStr . floatToString) x ‘seq‘ printf format
def printf (Bl  format) =
  \x -> (putStr . boolToString) x ‘seq‘ printf format
def printf (Chr format) =
  \x -> putChar x ‘seq‘ printf format

def main :: ()
def main =
  putStrLn "What is your name ?" ‘seq‘
  let name = getLine () in
  printf ((Lit "Hello " . Str . Lit "!" . Eol .
           Lit "The answer is: " . Inr . Eol) End) name 42
```

Chapter 3

Type system

3.1 Dunfield's and Krishnaswami's system

Bestrafer uses extended version of Dunfield's and Krishnaswami's bidirectional type system introduced in the article "Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types"[1]. Their unique approach enabling mixing existential and universal quantifiers in higher rank polymorphism is made possible by their novel polarized subtyping rule, which fixes the order in which quantifiers are instantiated, making the problem decidable and keeps the fundamental properties of subtyping, like stability under substitution and transitivity. Their system also features principality tracking, where in following rules: **!** means principal judgement and $\not{!}$ or omitted means non-principal.

Typing and subtyping rules

3.2 Our variant of the system

We made some necessary modification to the type system to make our language useful and user friendly. First of all we added typing rules for simple types such as `Int` or `String`, operators, `if` statements, `let` expressions, `error` throwing and `try - catch` blocks, but we omit them in this paper because they are not interesting and straightforward. However, it is important to remark, that `let` expression unpacks the existential types which is necessary to ensure correct order of inserting type variables to the context while defining recursive functions. We also added extra inference rules following the style of Dunfield and Krishnaswami [2013] [3], to mimize boilerplate type annotations and produce better quality typechecking errors. Following remark of Dunfield and Krishnaswami [2019] [1] we extended subtyping to functions and propositional types. The biggest modification is the introduction of user defined generalized algebraic data types. The last section of this chapter covers exhaustively typing rules and implementation details of GADTs. We discarded separate rules for `Vec`, treating it like any other GADT.

Types, monotypes and propositions

We distinguish between types (for clarity sometimes called big types) and monotypes. Basically monotypes are just simplified types (whithout quantification and propositional types) plus inhabitants of kind $\mathbb{N}$ (namely zero - `0` and successor - `S`). As we can see from the following definition quantification and propositions are restricted to monotypes. However use cases for polymorphism on big types seem to be rare in practice. Moreover our extended subtyping reduces number of programs which would not typecheck due to this restriction.

Kinds:

$\kappa ::=$

$\star \mid \mathbb{N}$

Types: (big types)

$A, B, C ::=$

`()` | `Bool` | `Int` | `Float` | `Char` | `String`	simple types
$\mid A_1 \times A_2 \times \cdots \times A_n$	product
$\mid \alpha$	universal variable
$\mid \hat{\alpha}$	existential variable
$\mid \forall t \colon \kappa. A$	universal quantification
$\mid \exists t \colon \kappa. A$	existential quantification
$\mid P \supset A$	guarded type
$\mid A \wedge P$	asserting type
$\mid$ *Type identifier* $\rho_1 \rho_2 \ldots \rho_n$	user defined GADT

GADT parameters:

$\rho ::=$

$A \mid n$	type or monotype

Monotypes:

$t, n ::=$

0	zero
$\mid S\, n$	successor of n
| `()` | `Bool` | `Int` | `Float` | `Char` | `String`	simple types
$\mid t_1 \times t_2 \times \cdots \times t_n$	product
$\mid \alpha$	universal variable
$\mid \hat{\alpha}$	existential variable
$\mid$ *Type identifier* $t_1 t_2 \ldots t_n$	user defined GADT

Propositions:

$P, Q ::=$
$\quad t = t`$

Higher rank polymorphism

One of the key features of the Dunfield's and Krishnaswami's system is higher rank polymorphism. Polymorphic types are treated like any other big type so they can be nested in each ohter arbitrarily many times. The following example uses higher rank universal quantification in GADT constructor to implement Scott's encoding of lists as a two continuations[4].

```
data ListS 'A where
  | ListS :: (forall r : * .
             ('A -> ListS 'A -> r) ->r -> r) ->
             ListS 'A

def nil :: forall a : * . ListS a
def nil = ListS (\co ni -> ni)

def cons :: forall a : * . a -> ListS a -> ListS a
def cons x xs = ListS (\co ni -> co x xs)

def uncons :: forall a : *, r : * .
  (a -> ListS a -> r) -> r -> ListS a -> r
def uncons co ni (ListS f) = f co ni
```

Bestrafer also allows higher rank existential quantification as the following example shows.

```
def heads :: forall n : N, a : * .
  Vec n (exists m : N . Vec (S m) a) ->
  Vec n a
def heads [] = []
def heads (x : xs) = head x : heads xs
```

Guarded types and asserting types

Bestrafer supports guarded types $P \supset A$ (P *implies* A) and asserting types $A \wedge P$ (A *with* P). Although the usage of propositions is in most cases implicit and hidden in typechecking GADTs, there are some use cases for propositional types. The following example uses guarded types to express GADT in continuation passing style[4].

```
data SomeC * where
```

```
  | SomeC :: forall a : *. (forall r : * .
             (a = Int => Int -> r) ->
             (a = String => String -> r) ->
             (a -> r) -> r) -> SomeC a

def int :: Int -> SomeC Int
def int x = SomeC (\i s o -> i x)

def string :: String -> SomeC String
def string x = SomeC (\i s o -> s x)

def other :: forall a : * . a -> SomeC a
def other x = SomeC (\i s o -> o x)

def unsome :: forall a : *, r : * .
  (Int -> r) ->
  (String -> r) ->
  (a -> r) ->
  SomeC a -> r
def unsome i s o (SomeC f) = f i s o

def main :: ()
def main =
  let x = other 3.14 in
  printInt <| unsome id intFromString floatToInt x
```

Extended subtyping

We added extra subtyping rules to improve flexibility and expressiveness of the type system. Without these rules the above example would not work, since we wouldn't be able to subtype `(Int -> r)` $<$ `(a = Int => Int -> r)`.

Function subtyping:

$$\frac{\Gamma \vdash A' \leq^{+} A \dashv \Theta \qquad \Theta \vdash [\Theta]B \leq^{-} [\Theta]B' \dashv \Delta}{\Gamma \vdash A \to B \leq^{-} A' \to B' \dashv \Delta}$$

Propositional types subtyping:

$$\frac{B\ \textit{not guarded} \qquad \Gamma \vdash P\ \textit{true} \dashv \Theta \qquad \Theta \vdash [\Theta]A \leq^- [\Theta]B \dashv \Delta}{\Gamma \vdash P \supset A \leq^- B \dashv \Delta}$$

$$\frac{\Gamma, \blacktriangleright_P / P \dashv \Theta \qquad \Theta \vdash [\Theta]A \leq^- [\Theta]B \dashv \Delta, \blacktriangleright_P, \Delta'}{\Gamma \vdash A \leq^- P \supset B \dashv \Delta} \qquad \frac{\Gamma, \blacktriangleright_P / P \dashv \bot}{\Gamma \vdash A \leq^- P \supset B \dashv \Gamma}$$

$$\frac{\Gamma, \blacktriangleright_P / P \dashv \Theta \qquad \Theta \vdash [\Theta]A \leq^+ [\Theta]B \dashv \Delta, \blacktriangleright_P, \Delta'}{\Gamma \vdash A \land P \leq^+ B \dashv \Delta} \qquad \frac{\Gamma, \blacktriangleright_P / P \dashv \bot}{\Gamma \vdash A \land P \leq^+ B \dashv \Gamma}$$

$$\frac{A\ \textit{not asserting} \qquad \Gamma \vdash P\ \textit{true} \dashv \Theta \qquad \Theta \vdash [\Theta]A \leq^+ [\Theta]B \dashv \Delta}{\Gamma \vdash A \leq^+ B \land P \dashv \Delta}$$

3.3 Our contribution - user defined GADTs

Named and unnamed parameters

Let's take a look again at the definition of data type Vec:

```
data Vec N 'A where
  | []  :: Vec 0 'A
  | (:) :: forall n : N . 'A -> Vec n 'A -> Vec (S n) 'A
```

One could wonder why do we need named parameters in our type system. Couldn't we just use unnamed parameters and quantifiers, like in the example below ?

```
data Vec N * where
  | []  :: forall a : * . Vec 0 a
  | (:) :: forall n : N, a : * . a -> Vec n a -> Vec (S n) a
```

That's true, we can define Vec like that, but there is a drawback to that approach. Since we are using quantification on a, our definition of Vec is restricted to monotypes, so, for example, we wouldn't be able to typecheck vector of mixed length vectors: `Vec n (exists m : N . Vec m a)`. That's where named parameters come into play. They are capable of storing big types, but that also means that they cannot be involved in type equations. As we can see, the combination of both kinds of parameters is essential to provide full expressive power to the programmer.

Building and typechecking constructors

We build GADT representations between parsing and typechecking process. We start by checking well-formedness of constructors. We define well formed constructor in the following manner:

$Well\, formed\, constructor ::= Universal$

$Universal ::=$
$\quad \forall\alpha : \kappa\, .\, Universal \mid Arrow$

$Arrow ::=$
$\quad A \rightarrow Arrow \mid Well\, formed\, result\, type$

By $Well\, formed\, result\, type$ we mean the type that matches type signature of currently defined GADT, where positions of named parameters and kinds of types associated with unnamed parameters also match the type signature. After that we build two representations of each constructor: template representation which is used when we check constructor expression against known GADT type and functional which is used in all other cases (namely, partial application and passing constructor as an argument to a function).

Template representation

For the purpose of template representation we defined type templates, which basically are types with indexed holes, which may be filled with any big type or monotype.

Type templates:

$A_\dagger, B_\dagger, C_\dagger ::=$

`()` | `Bool` | `Int` | `Float` | `Char` | `String`	simple types
$\mid A_{\dagger 1} \times A_{\dagger 2} \times \cdots \times A_{\dagger n}$	product
$\mid \alpha$	universal variable
$\mid \hat{\alpha}$	existential variable
$\mid \forall t \colon \kappa . A_\dagger$	universal quantification
$\mid \exists t \colon \kappa . A_\dagger$	existential quantification
$\mid P_\dagger \supset A_\dagger$	guarded type
$\mid A_\dagger \wedge P_\dagger$	asserting type
$\mid$ *Type identifier* $\rho_{\dagger 1} \rho_{\dagger 2} \ldots \rho_{\dagger n}$	user defined GADT
$\mid 1, 2, 3, \ldots$	index of GADT parameter

GADT parameter templates:

$\rho ::=$
$\quad A_\dagger \mid n_\dagger$ type template or monotype template

Proposition templates:

$P_\dagger, Q_\dagger ::=$
$\quad t_\dagger = t`_\dagger$

We define monotype templates similarly to big type templates, so we omit the formal definition for space reasons.

Template representation consists of list of universally quantified variables, list of propositions and list of constructor arguments represented as type templates. We substitute named parameters identifiers and unnamed parameters in the result type with *parameters indices*, which correspond to adequate parameters in the type signature. We generate propositions automatically based on constructor's result type.

When typechecking a constructor, we start by checking if its result type matches the type against which we are typechecking. Then we check the arity of the constructor. After that we substitute constructor's universally quantified variables with fresh existential variables. Then we convert arguments' type templates and propositions' templates to types and propositions by replacing parameters' indices with types and monotypes from checked type's parameters. Next, we check propositions. Finally, we check constructor's arguments against generated types. The following, quite lenghty, rule describes that process in the formal way:

$$prnc(A,\ p) = \begin{cases} \not{!} & when\ FEV(A) \neq \emptyset \\ p & when\ FEV(A) = \emptyset \end{cases}$$

$$\frac{\begin{array}{c} typeName_{constrName} = typeName \qquad \alpha_1, \alpha_2, \ldots, \alpha_m \leftarrow uvars_{constrName} \\ P_{\dagger 1}, P_{\dagger 2}, \ldots, P_{\dagger l} \leftarrow props_{constrName} \qquad A_{\dagger 1}, A_{\dagger 2}, \ldots, A_{\dagger k} \leftarrow args_{constrName} \\ P'_{\dagger 1}, P'_{\dagger 2}, \ldots, P'_{\dagger l} \leftarrow [\hat{\alpha}_1/\alpha_1, \hat{\alpha}_2/\alpha_2, \ldots, \hat{\alpha}_m/\alpha_m] P_{\dagger 1}, P_{\dagger 2}, \ldots, P_{\dagger l} \\ P_1, P_2, \ldots, P_l \leftarrow [\rho_1/1, \rho_2/2, ..., \rho_n/n] P'_{\dagger 1}, P'_{\dagger 2}, \ldots, P'_{\dagger l} \\ A'_{\dagger 1}, A'_{\dagger 2}, \ldots, A'_{\dagger k} \leftarrow [\hat{\alpha}_1/\alpha_1, \hat{\alpha}_2/\alpha_2, \ldots, \hat{\alpha}_m/\alpha_m] A_{\dagger 1}, A_{\dagger 2}, \ldots, A_{\dagger k} \\ A_1, A_2, \ldots, A_k \leftarrow [\rho_1/1, \rho_2/2, ..., \rho_n/n] A'_{\dagger 1}, A'_{\dagger 2}, \ldots, A'_{\dagger k} \\ \Gamma \vdash P_1\ true \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1] P_2\ true \dashv \Theta_2 \qquad \cdots \qquad \Theta_{l-1} \vdash [\Theta_{l-1}] P_l\ true \dashv \Theta_l \\ \Theta_l \vdash e_1 \Leftarrow [\Theta_l] A_1\ prnc(A_1,\ p) \dashv \Delta_1 \qquad \Delta_1 \vdash e_2 \Leftarrow [\Delta_1] A_2\ prnc(A_2,\ p) \dashv \Delta_2 \\ \cdots \qquad \Delta_{k-1} \vdash e_k \Leftarrow [\Delta_{k-1}] A_k\ prnc(A_k,\ p) \dashv \Delta_k \end{array}}{\Gamma \vdash constrName\ e_1\ e_2\ \ldots\ e_k \Leftarrow (typeName\ \rho_1\ \rho_2\ \ldots\ \rho_n)\ p \dashv \Delta_k}$$

Functional representation

Supplementary to the template representation, we represent constructors as a polymorphic functions. To build functional representation we change named parameters into universally quantified variables. For example cons of `Vec`:

```
(:) :: forall n : N . 'A -> Vec n 'A -> Vec (S n) 'A
```

is represented as:

```
(:) :: forall a : *, n : N . a -> Vec n a -> Vec (S n) a
```

Since we are using universal quantification on all parameters, when using functional representation every parameter must by monotype. This is why we put so much effort into creating template representation based on named parameters, which is not restricted to monotypes.

Chapter 4

Remarks on semantics

4.1 GADT constructors

Partially applied constructors are changed into (interpreted as) lambda expressions, that take as input missing arguments of the constructor. For example, if we consider type:

```
data Pair 'A 'B where
  | Pair :: 'A -> 'B -> Pair 'A 'B
```

partial aplication: `Pair 42` will be changed into `\b -> Pair 42 b`.

4.2 Function definitions and evaluation order

Top-level pattern matching definitions are just syntax sugar for lambda with a match expression inside. For example function definition:

```
def map _ [] = []
def map f (x : xs) = f x : map f xs
```

will be desugared into:

```
\x0 x1 -> case (x0, x1) of
  | (_, []) -> []
  | (f, x : xs) -> f x : map f xs
```

The top-level definitions are all mutually recursive and each of them is evaluated once starting from the beginning down to the end of the source file. This may raise a question, what happens if some function or definition uses the definition which is located lower in the source file. In this situation the lower definition will be evaluated at the time when it was called in higher definition, its value will be remembered and the definition won't be evaluated for the second time. Another concern is what would happend, if in the previous situation lower function would throw an error and the higher function would handle it with a `try-catch` block. Because lower definition failed to evaluate we treat it as non evaluated and try to evaluate it on its next use or at the top-level.

Chapter 5

Future work

Bibliography

[1] J. Dunfield and Neel Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism and indexed types. In *Principles of Programming Languages (POPL)*, January 2019. `http://www.cl.cam.ac.uk/~nk480/gadt.pdf`.

[2] Andrew Kennedy and Claudio Russo. Generalized algebraic data types and object-oriented programming. volume 40, 10 2005.

[3] J. Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming (ICFP)*, September 2013. `arXiv:1306.6032[cs.PL]`.

[4] `https://ocharles.org.uk/guest-posts/2014-12-18-rank-n-types.html`.