Phantom Types

Marcin Kaczmarek

University of Wrocław
Institute of Computer Science

March 18, 2015

Introduction

Definition

A *phantom type* is a parametrized type whose parameters don't all appear on the left-hand side of its definition.

Introduction

Definition

A *phantom type* is a parametrized type whose parameters don't all appear on the left-hand side of its definition.

Example

```
type 'a foo = Foo | Bar of int
```

Sanitable string builder

Toy Example

String builder supporting a custom cleanup routine.

Sanitable string builder

Toy Example

String builder supporting a custom cleanup routine.

```
module type STRING_BUILDER =
sig
  type t
  val init : (string -> string) -> t
  val append : t -> string -> t
  val clean : t -> t
  val get : t -> string
end
```

Sanitable string builder – 1st approach

```
module StringBuilder : STRING_BUILDER =
struct
  type t = {
    str : string;
    cleaner : string -> string;
    dirty : bool;
  }

  let init cleaner = {
    str = "";
    cleaner = cleaner;
    dirty = true;
  }

  (* . . . *)
end
```

Sanitable string builder – 1st approach

```
let append str sb = { sb with
  str = sb.str ^ str;
  dirty = true
}

let clean sb = { sb with
  str = sb.clean sb.str;
  dirty = false
}

let get sb =
  if sb.dirty then failwith "Dirty!"
  else sb.str
```

Sanitable string builder – 1st approach – drawbacks

Sanitable string builder – 1st approach – drawbacks

- errors discovered at runtime,

Sanitable string builder – 1st approach – drawbacks

- errors discovered at runtime,
- so we need a fallback code,

Sanitable string builder – 1st approach – drawbacks

- errors discovered at runtime,
- so we need a fallback code,
- performance overhead,

Sanitable string builder – 1st approach – drawbacks

- errors discovered at runtime,
- so we need a fallback code,
- performance overhead,
- explicit state maintanance.

Sanitable string builder – 2nd approach

```
module type STRING_BUILDER =
sig
  type 'a t
  type clean
  type dirty
  val init : (string -> string) -> dirty t
  val append : string -> 'a t -> dirty t
  val clean : 'a t -> clean t
  val get : clean t -> string
end
```

Sanitable string builder – 2nd approach

```
module StringBuilder : STRING_BUILDER =
struct
  type 'a t = {
    str : string;
    cleaner : string -> string;
  }
  type clean
  type dirty

  let init cleaner = { str = ""; cleaner; }
  let append str sb = { sb with str = sb.str ^ str }
  let clean sb = { sb with str = sb.cleaner sb.str }
  let get sb = sb.str
end
```

Sanitable string builder – 2nd approach

```
let open StringBuilder in
init (fun x -> x)
|> append "foo"
|> append "bar"
|> get
```

Sanitable string builder – 2nd approach

```
let open StringBuilder in
init (fun x -> x)
|> append "foo"
|> append "bar"
|> get
```

```
Error:
  This expression has type clean t -> bytes
  but an expression was expected of type dirty t -> 'a
  Type clean is not compatible with type dirty
```

GADT

Definition

A *generalized algebraic data type* (GADT) is an extension of parametrized algebraic data type. With this extension, one can freely choose the parameters of the **return** type of data constructor.

GADT

Definition

A *generalized algebraic data type* (GADT) is an extension of parametrized algebraic data type. With this extension, one can freely choose the parameters of the **return** type of data constructor.

OCaml syntax

```
type 'a foo =
  | Foo : 'a foo
  | Bar : int foo
  | Baz : int * 'a foo -> (int * 'a) foo
```

Statically typed abstract syntax

Statically typed abstract syntax

With GADT we can embed a programming language in a type-safe fashion.

Statically typed abstract syntax

With GADT we can embed a programming language in a type-safe fashion.

```
type _ term =
  | Zero : int term
  | Succ : int term -> int term
  | Pred : int term -> int term
  | IsZero : int term -> bool term
  | If : bool term * 'a term * 'a term -> 'a term
```

Statically typed abstract syntax

```
let rec eval : type a . a term -> a = function
  | Zero -> 0
  | Succ expr -> eval expr + 1
  | Pred expr -> eval expr - 1
  | IsZero expr -> eval expr = 0
  | If (cexpr, texpr, fexpr) ->
    if eval cexpr then eval texpr
    else eval fexpr
```

Statically typed abstract syntax

```
let rec eval : type a . a term -> a = function
  | Zero -> 0
  | Succ expr -> eval expr + 1
  | Pred expr -> eval expr - 1
  | IsZero expr -> eval expr = 0
  | If (cexpr, texpr, fexpr) ->
    if eval cexpr then eval texpr
    else eval fexpr
```

- We take advantage of an important feature of GADTs – **pattern matching causes type refinement**.

Statically typed abstract syntax

```
let rec eval : type a . a term -> a = function
  | Zero -> 0
  | Succ expr -> eval expr + 1
  | Pred expr -> eval expr - 1
  | IsZero expr -> eval expr = 0
  | If (cexpr, texpr, fexpr) ->
    if eval cexpr then eval texpr
    else eval fexpr
```

- We take advantage of an important feature of GADTs – **pattern matching causes type refinement**.
- This interpreter is notably *tag free*.

Statically typed abstract syntax

```
let rec eval : type a . a term -> a = function
  | Zero -> 0
  | Succ expr -> eval expr + 1
  | Pred expr -> eval expr - 1
  | IsZero expr -> eval expr = 0
  | If (cexpr, texpr, fexpr) ->
    if eval cexpr then eval texpr
    else eval fexpr
```

- We take advantage of an important feature of GADTs – **pattern matching causes type refinement**.
- This interpreter is notably *tag free*.

Note on syntax: `type a` declares a *locally abstract type* a.

Statically typed abstract syntax

```
> let one = Succ Zero
val one : int term = Succ Zero
> eval one
- : int = 1
> eval (If (IsZero one, one, Zero))
- : int = 0
> Pred (IsZero Zero)

Error:
  This expression has type bool term
  but an expression was expected of type int term
  Type bool is not compatible with type int
```

Statically typed abstract syntax

The type `'a term` is quite unusual.

Statically typed abstract syntax

The type `'a term` is quite unusual.

- `term` is **not** a container type. An element of `int term` is an expression that evaluates to an integer – not a data stucture that contains integers.

Statically typed abstract syntax

The type `'a term` is quite unusual.

- `term` is **not** a container type. An element of `int term` is an expression that evaluates to an integer – not a data stucture that contains integers.
- We cannot define a mapping function

  ```
  ('a -> 'b) -> 'a term -> 'b term
  ```

  as for many other data types.

Statically typed abstract syntax

The type `'a term` is quite unusual.

- `term` is **not** a container type. An element of `int term` is an expression that evaluates to an integer – not a data stucture that contains integers.
- We cannot define a mapping function

  ```
  ('a -> 'b) -> 'a term -> 'b term
  ```

  as for many other data types.
- The type `'b term` may not event be inhabited. There are, for instance, no terms of type `string term`.

Statically typed abstract syntax

The type `'a term` is quite unusual.

- `term` is **not** a container type. An element of `int term` is an expression that evaluates to an integer – not a data stucture that contains integers.
- We cannot define a mapping function
  ```
  ('a -> 'b) -> 'a term -> 'b term
  ```
  as for many other data types.
- The type `'b term` may not event be inhabited. There are, for instance, no terms of type `string term`.

The type argument of `term` is not related to any component, therefore we also call `term` a phantom type.

Generic programming

We can use phantom types to implement **generic functions**, i.e., functions that work for a family of types.

Generic programming

We can use phantom types to implement **generic functions**, i.e., functions that work for a family of types.

```
type _ typ =
  | Int : int typ
  | Bool : bool typ
  | String : string typ
  | List : 'a typ -> 'a list typ
  | Pair : 'a typ * 'b typ -> ('a * 'b) typ
```

Generic programming

We can use phantom types to implement **generic functions**, i.e., functions that work for a family of types.

```
type _ typ =
  | Int : int typ
  | Bool : bool typ
  | String : string typ
  | List : 'a typ -> 'a list typ
  | Pair : 'a typ * 'b typ -> ('a * 'b) typ
```

```
> List Int
- : int list typ = List Int
> Pair (Int, Bool)
- : (int * bool) typ = Pair (Int, Bool)
```

Generic programming

Now we can implement a generic `show` function.

```
let rec show : type a . a typ -> a -> string =
  fun t x -> match t with
   | Int -> string_of_int x
   | Bool -> string_of_bool x
   | String -> "'" ^ x ^ "'"
   | List t' ->
     let aux s y = s ^ show t' y ^ " " in
     fold_left aux "[ " x ^ "]"
   | Pair (ta, tb) ->
     "("  ^ show ta (fst x) ^ ", " ^ show tb (snd x) ^ ")"
```

Generic programming

Now we can implement a generic `show` function.

```
let rec show : type a . a typ -> a -> string =
  fun t x -> match t with
   | Int -> string_of_int x
   | Bool -> string_of_bool x
   | String -> "'" ^ x ^ "'"
   | List t' ->
     let aux s y = s ^ show t' y ^ " " in
     fold_left aux "[ " x ^ "]"
   | Pair (ta, tb) ->
     "("  ^ show ta (fst x) ^ ", " ^ show tb (snd x) ^ ")"
```

But we still have to explicitly provide the type using an instance of `typ`.

Generic programming

```
> show Int
- : int -> string = <fun>
> show (Pair (Int, Bool))
- : int * bool -> string = <fun>
> show (List Int) [1; 2; 3]
- : string = "[ 1 2 3 ]"
```

Dynamic values

Dynamic values

Using the `typ` type we can introduce a *universal type* which we call `dyn`. It bundles a value with the representation of its type.

Dynamic values

Using the `typ` type we can introduce a *universal type* which we call `dyn`. It bundles a value with the representation of its type.

```
type dyn = Dyn : 'a typ * 'a -> dyn
and _ typ =
  (* . . . *)
  | Dynamic : dyn typ
```

Dynamic values

Using the `typ` type we can introduce a *universal type* which we call `dyn`. It bundles a value with the representation of its type.

```
type dyn = Dyn : 'a typ * 'a -> dyn
and _ typ =
  (* . . . *)
  | Dynamic : dyn typ
```

- We also injected the representation of `dyn` into `typ` in order to allow for dynamic collections of dynamic values.

Dynamic values

Using the `typ` type we can introduce a *universal type* which we call `dyn`. It bundles a value with the representation of its type.

```
type dyn = Dyn : 'a typ * 'a -> dyn
and _ typ =
  (* . . . *)
  | Dynamic : dyn typ
```

- We also injected the representation of `dyn` into `typ` in order to allow for dynamic collections of dynamic values.
- The polymorphic parameter of the `Dyn` constructor is *existentially quantified*. This is another feature added by GADTs.

Dynamic values

- Checking for type equality, which we clearly need to perform a safe dynamic cast, is a little tricky since we cannot just compare two instances of `typ`.

Dynamic values

- Checking for type equality, which we clearly need to perform a safe dynamic cast, is a little tricky since we cannot just compare two instances of `typ`.
- Even if we could, and the types would match; how can we convince the compiler that the bundled value is indeed of the specified type? Existentially quantified type variables always instantiate as a fresh locally abstract type when matched.

Dynamic values

```
let rec tequal
  : type a b . a typ -> b typ -> (a -> b) option =
  fun t1 t2 -> match t1, t2 with
  | Int, Int -> Some id
  | Bool, Bool -> Some id
  | String, String -> Some id
  | List t1', List t2' -> (
    match tequal t1' t2' with
    | Some f -> Some (map f)
    | None -> None )
  | Pair (ta1, tb1), Pair (ta2, tb2) -> (
    match tequal ta1 ta2, tequal tb1 tb2 with
    | Some f, Some g -> Some (fun (x, y) -> (f x, g y))
    | _ -> None )
  | _ -> None
```

Dynamic values

```
let cast : type a . a typ -> dyn -> a option =
  fun t2 (Dyn (t1, x)) -> match tequal t1 t2 with
  | Some f -> Some (f x)
  | None -> None
```

Dynamic values

```
let cast : type a . a typ -> dyn -> a option =
  fun t2 (Dyn (t1, x)) -> match tequal t1 t2 with
  | Some f -> Some (f x)
  | None -> None
```

Note: a significant drawback of `tequal` is that the *casting* function produced is performing a deep copy of the passed value. A part of Your homework is to fix this.

Dynamic values

```
> let d = Dyn (Int, 60)
val d : dyn = Dyn (Int, <poly>)
> cast Int d
- : int option = Some 60
> cast Bool d
- : bool option = None
> let ds = Dyn (List Int, [1; 2; 3])
val ds : dyn = Dyn (List Int, <poly>)
> cast (List Int) ds
- : int list option = Some [1; 2; 3]
```

Functional unparsing

Functional unparsing

Is it possible to implement a `printf`-like functionality in a statically typed language?

Functional unparsing

Is it possible to implement a `printf`-like functionality in a statically typed language?

We can tackle it with dedicated format directives. Here's our desired result:

```
> format (Lit "Richard")
- : string = "Richard"
> format Int
- : int -> string = <fun>
> format Int 60
- : string = "60"
> format (String ++ Lit " is " ++ Int)
- : string -> int -> string = <fun>
> format (String ++ Lit " is " ++ Int) "Richard" 60
- : string = "Richard is 60"
```

Functional unparsing

Ideally we would like to have a parametrized type `'a dir` and a function `format` of type `'a dir -> 'a`.

Functional unparsing

Ideally we would like to have a parametrized type `'a dir` and a function `format` of type `'a dir -> 'a`.

A directive can be seen as a binary tree with `( ++ )`-es as inner nodes and simple directives, i.e., `Lit`, `String` and `Int`, as leaves.

Functional unparsing

Ideally we would like to have a parametrized type `'a dir` and a function `format` of type `'a dir -> 'a`.

A directive can be seen as a binary tree with `( ++ )`-es as inner nodes and simple directives, i.e., `Lit`, `String` and `Int`, as leaves.

```
type _ dir =
  | Lit : string -> string dir
  | String : (string -> string) dir
  | Int : (int -> string) dir
  | Node : 'a dir * 'b dir -> (* ?? *) dir
```

Functional unparsing

Ideally we would like to have a parametrized type `'a dir` and a function `format` of type `'a dir -> 'a`.

A directive can be seen as a binary tree with `( ++ )`-es as inner nodes and simple directives, i.e., `Lit`, `String` and `Int`, as leaves.

```
type _ dir =
  | Lit : string -> string dir
  | String : (string -> string) dir
  | Int : (int -> string) dir
  | Node : 'a dir * 'b dir -> (* ?? *) dir
```

But we cannot express *concatenation* of functional types.

Functional unparsing

A clever technique resembling type level difference lists solves this one.

```
type (_, _) dir =
  | Lit : string -> ('a, 'a) dir
  | String : (string -> 'a, 'a) dir
  | Int : (int -> 'a, 'a) dir
  | Node : ('a, 'b) dir * ('b, 'c) dir -> ('a, 'c) dir

let ( ++ ) d1 d2 = Node (d1, d2)
```

Functional unparsing

```
let format : type a . (a, string) dir -> a = function
  | Lit s -> s
  | String -> id
  | Int -> string_of_int
  | Node (d1, d2) -> (* ?? *)
```

Functional unparsing

```
let format : type a . (a, string) dir -> a = function
  | Lit s -> s
  | String -> id
  | Int -> string_of_int
  | Node (d1, d2) -> (* ?? *)
```

We suffer the same defficiency as with the naive approach for the `dir` type.

Functional unparsing

```
let format : type a . (a, string) dir -> a = function
  | Lit s -> s
  | String -> id
  | Int -> string_of_int
  | Node (d1, d2) -> (* ?? *)
```

We suffer the same defficiency as with the naive approach for the `dir` type.

This time we are saved by continuations. $\rightarrow$

Functional unparsing

```
let rec format_aux
  : type a b . (a, b) dir -> (string -> b)
             -> string -> a =
  fun d k r -> match d with
  | Lit s -> k (r ^ s)
  | String -> fun s -> k (r ^ s)
  | Int -> fun n -> k (r ^ string_of_int n)
  | Node (d1, d2) -> format_aux d1 (format_aux d2 k) r
```

Functional unparsing

```
let rec format_aux
  : type a b . (a, b) dir -> (string -> b)
             -> string -> a =
  fun d k r -> match d with
  | Lit s -> k (r ^ s)
  | String -> fun s -> k (r ^ s)
  | Int -> fun n -> k (r ^ string_of_int n)
  | Node (d1, d2) -> format_aux d1 (format_aux d2 k) r
```

- `d` is the directive slice. By the difference list analogy, we have to handle what's contained in `a - b`.

Functional unparsing

```
let rec format_aux
  : type a b . (a, b) dir -> (string -> b)
             -> string -> a =
  fun d k r -> match d with
  | Lit s -> k (r ^ s)
  | String -> fun s -> k (r ^ s)
  | Int -> fun n -> k (r ^ string_of_int n)
  | Node (d1, d2) -> format_aux d1 (format_aux d2 k) r
```

- `d` is the directive slice. By the difference list analogy, we have to handle what's contained in `a - b`.
- `r` is the result produced by the preceding part of the directive.

Functional unparsing

```
let rec format_aux
  : type a b . (a, b) dir -> (string -> b)
             -> string -> a =
  fun d k r -> match d with
  | Lit s -> k (r ^ s)
  | String -> fun s -> k (r ^ s)
  | Int -> fun n -> k (r ^ string_of_int n)
  | Node (d1, d2) -> format_aux d1 (format_aux d2 k) r
```

- `d` is the directive slice. By the difference list analogy, we have to handle what's contained in `a - b`.
- `r` is the result produced by the preceding part of the directive.
- `k` is the continuation that handles what's to the right of `b`.

Functional unparsing

Therefore `let format d = format_aux d id ""`.

Functional unparsing

Therefore `let format d = format_aux d id ""`.

That's a nice quadratic procedure we have here...

Generic traversals and queries

Generic traversals and queries

Suppose we have to write a function that traverses a complex data structure, e.g., representing a company's organizational structure, and acts upon the data, say, increases the age of each employee by one.

Generic traversals and queries

Suppose we have to write a function that traverses a complex data structure, e.g., representing a company's organizational structure, and acts upon the data, say, increases the age of each employee by one.

We would like the boilerplate part, that is responsible for traversing, to be generic and reusable with different data structures.

Generic traversals and queries

Recall the *type representation* type.

```
type _ typ =
  | Int : int typ
  | Bool : bool typ
  | String : string typ
  | List : 'a typ -> 'a list typ
  | Pair : 'a typ * 'b typ -> ('a * 'b) typ
```

Generic traversals and queries

Let's introduce a new type for *traversal* objects.

```
type trav = 'a . 'a typ -> ('a -> 'a)
```

Generic traversals and queries

Let's introduce a new type for *traversal* objects.

```
type trav = 'a . 'a typ -> ('a -> 'a)
```

An instance of `trav` may be seen as a function that given a type representation returns a mapping.

Generic traversals and queries

Let's introduce a new type for *traversal* objects.

```
type trav = 'a . 'a typ -> ('a -> 'a)
```

An instance of `trav` may be seen as a function that given a type representation returns a mapping.

```
> let m = bump_ints (List (Pair (Int, Bool)))
val m : (int * bool) list -> (int * bool) list = <fun>
> m [(1, true); (2, false)]
- : (int * bool) list = [(2, true); (3, false)]
```

Generic traversals and queries

Let's introduce a new type for *traversal* objects.

```
type trav = { f : 'a . 'a typ -> ('a -> 'a) }
```

An instance of `trav` may be seen as a function that given a type representation returns a mapping.

```
> let m = bump_ints.f (List (Pair (Int, Bool)))
val m : (int * bool) list -> (int * bool) list = <fun>
> m [(1, true); (2, false)]
- : (int * bool) list = [(2, true); (3, false)]
```

Generic traversals and queries

Here's a simple traversal that increments an `int`.

```
let bump =
  let aux : type a . a typ -> (a -> a) = function
    | Int -> ( + ) 1
    | _ -> id
  in { f = aux }
```

Generic traversals and queries

Here's a simple traversal that increments an `int`.

```
let bump =
  let aux : type a . a typ -> (a -> a) = function
    | Int -> ( + ) 1
    | _ -> id
  in { f = aux }
```

In fact, it's not much of a traversal. The only data structure it supports is a sole integer.

```
> bump.f Int 5
- : int = 6
> bump.f (List Int) [1; 2; 3]
- : int list = [1; 2; 3]
```

Generic traversals and queries

In order to build more general traversals we introduce a *traversal transformer* `imap : trav -> trav` which produces a traversal acting upon immediate children of an object.

```
> (imap bump).f Int 5
- : int = 5
> (imap bump).f (List Int) [1; 2; 3]
- : int list [2; 3; 4]
```

Generic traversals and queries

```
let imap { f } =
  let aux : type a . a typ -> (a -> a) = function
    | Int -> id
    | Bool -> id
    | String -> id
    | List t -> map (f t)
    | Pair (at, bt) -> fun (x, y) -> (f at x, f bt y)
  in { f = aux }
```

Generic traversals and queries

Let `( >> )` be an opertor for sequencing traversals.

```
let ( >> ) tr1 tr2 =
  let aux t x = tr2.f t (tr1.f t x)
  in { f = aux }

val ( >> ) : trav -> trav -> trav = <fun>
```

Generic traversals and queries

Let `( >> )` be an opertor for sequencing traversals.

```
let ( >> ) tr1 tr2 =
  let aux t x = tr2.f t (tr1.f t x)
  in { f = aux }
```

```
val ( >> ) : trav -> trav -> trav = <fun>
```

The sequencing operator has `copy` as its identity.

```
let copy = { f = fun t -> id }
tr >> copy = copy >> tr = tr
```

Generic traversals and queries

We can now build a transformer `allover` so that

```
> let bump_ints = allover bump
val bump_ints : trav = {f = <fun>}
> bump_ints.f Int 5
- : int = 6
> bump_ints.f (List Int) [1; 2; 3]
- : int list = [2; 3; 4]
```

Generic traversals and queries

There are two flavours

```
let rec allover  tr = tr >> imap (allover tr)
let rec allover' tr = imap (allover tr) >> tr
```

Generic traversals and queries

There are two flavours

```
let rec allover  tr = tr >> imap (allover tr)
let rec allover' tr = imap (allover tr) >> tr
```

But with eager evaluation the recursion will explode.

Generic traversals and queries

There are two flavours

```
let rec allover  tr = tr >> imap (allover tr)
let rec allover' tr = imap (allover tr) >> tr
```

But with eager evaluation the recursion will explode.

```
let lazy_trav ltr = { f = fun t -> (force ltr).f t }
let rec allover tr =
  tr >> imap (lazy_trav (lazy (allover tr)))
```

Generic traversals and queries

For generic queries we introduce a similar type but the result of the mapping function has a fixed type.

```
type 'b query = { q : 'a . 'a typ -> ('a -> 'b) }
```

Generic traversals and queries

For generic queries we introduce a similar type but the result of the mapping function has a fixed type.

```
type 'b query = { q : 'a . 'a typ -> ('a -> 'b) }
```

From now on we will work only with integer queries. It will be Your task to generalize it.

Generic traversals and queries

In place of `imap` there is

```
let isum { q }  =
  let null _ = 0 in
  let aux : type a . a typ -> (a -> int) = function
    | Int -> null
    | Bool -> null
    | String -> null
    | List t -> fold_left (fun a x -> a + q t x) 0
    | Pair (at, bt) -> fun (x, y) -> q at x + q bt y
  in { q = aux }
```

Generic traversals and queries

For merging the results of two queries we have

```
let ( ++ ) qr1 qr2 =
  let aux t x = qr1.q t x + qr2.q t x
  in { q = aux }
```

Generic traversals and queries

For merging the results of two queries we have

```
let ( ++ ) qr1 qr2 =
  let aux t x = qr1.q t x + qr2.q t x
  in { q = aux }
```

And finally

```
let rec total qr = qr ++ isum (total qr)
```

Generic traversals and queries

For merging the results of two queries we have

```
let ( ++ ) qr1 qr2 =
  let aux t x = qr1.q t x + qr2.q t x
  in { q = aux }
```

And finally

```
let rec total qr = qr ++ isum (total qr)
```

or rather

```
let lazy_query lqr = { q = fun t -> (force lqr).q t }
let rec total qr =
  qr ++ isum (lazy_query (lazy (total qr)))
```

Generic traversals and queries

```
let sizeof =
  let one _ = 1 in
  let aux : type a . a typ -> (a -> int) = function
    | Int -> one
    | Bool -> one
    | String -> String.length
    | List _ -> List.length
    | Pair (_, _) -> one
  in { q = aux }
```

Generic traversals and queries

```
let sizeof =
  let one _ = 1 in
  let aux : type a . a typ -> (a -> int) = function
    | Int -> one
    | Bool -> one
    | String -> String.length
    | List _ -> List.length
    | Pair (_, _) -> one
  in { q = aux }

> (total sizeof).q (List String) ["Hello"; "World!"]
- : int = 13
```

Phantom Types

Marcin Kaczmarek

University of Wrocław
Institute of Computer Science

March 18, 2015

Further Reading I

Ralf Hinze.
Fun with phantom types.
In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*. Pages 245–262. Palgrave Macmillan. 2003.
`http://www.cs.ox.ac.uk/ralf.hinze/talks/FOP.pdf`

Ralf Hinze.
Fun with phantom types.
Slides for the talk during *The Fun of Programming*, A symposium in honour of Professor Richard Bird's 60th birthday. 2003.
`http://www.cs.ox.ac.uk/ralf.hinze/publications/With.pdf`

Further Reading II

The OCaml manual
Chapter 7 *Language extensions*. Sections 7.13, 7.18.
http://caml.inria.fr/pub/docs/manual-ocaml-400/manual021.html

Existentially quantified types
Wikibooks – Haskell
http://en.wikibooks.org/wiki/Haskell/Existentially_quantified_types

Phantom type
The Haskell Wiki
http://wiki.haskell.org/Phantom_type