KTH ROYAL INSTITUTE OF TECHNOLOGY

Universal Collection Types

"One Collection Type to Rule Them All"

Financially supported by the Swedish Foundation for Strategic Research

digital futures

Viktor Palmkvist (vipa@kth.se)
Anders Ågren Thuné (aathn@kth.se)
Elias Castegren (elias.castegren@it.uu.se)
David Broman (dbro@kth.se & broman@stanford.edu)

Performance

Writing fast programs

`Difficult`

Compiler can help

`2*a ⇒ a << 2`

Bigger gains:

1. Switching algorithms
2. Switching data-structures

$O(n^2) \Rightarrow O(n * \log n)$

Performance: Data Structures

Collection | Sequence

	Linked List	Array List / Vector / …
Read 1st	O(1)	O(1)
Read Nth	O(n)	O(1)
Insert 1st	O(1)	O(n)

Switching Data Structures

Manually test and switch everywhere	`Tedious`
Edits easily change optimal structure	`Re-test, re-switch, tedious`
Can the compiler help?	`Well...`

Challenges

Accuracy, overhead (compile-time or run-time), extensibility, non-leaky abstractions

Representation switching

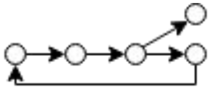
```
Operations: A, B, C, D
```

```
A, B, A, …  ──────────▶  C, D, C, …
```

```
Operations: A, B    Convert    Operations: C, D
```

Key Ideas

Functional/immutable/persistent interface makes data-flow apparent



vs.

Pick operation implementations, representations are “just” constraints

```
peek : LinkedList a -> Option a
```

vs.

```
LinkedList{peek, ...}
```

Example: show_seq

```
let show_seq
  : ('a -> char seq) -> 'a seq -> char seq
  = fun f xs ->
    match split_first xs with
    | Some (x, xs) ->
      let work acc x = concat (concat acc ", ") (f x) in
      let mid = foldl work (f x) xs in
      concat (concat "[" mid) "]"
    | _ -> "[]"
```

Abstract type

Operations

`split_first`

`concat`

`foldl`

```
let ex1 = show_seq string_of_int [1; 2; 3]
(* ex1 : char seq = "[1, 2, 3]" *)
let ex2 = show_seq (fun x -> x) ["hello"; "world"]
(* ex2 : char seq = "[hello, world]" *)
```

Example: show_seq

'a seq
↓
split_first
↓
foldl

```
let show_seq
  : ('a -> char seq) -> 'a seq -> char seq
  = fun f xs ->
    match split_first xs with
    | Some (x, xs) ->
      let work acc x = concat (concat acc ", ") (f x) in
      let mid = foldl work (f x) xs in
      concat (concat "[" mid) "]"
    | _ -> "[]"
```

'a -> char seq
↓
concat
↓
concat
↓
char seq

```
let ex1 = show_seq string_of_int [1; 2; 3]
(* ex1 : char seq = "[1, 2, 3]" *)
let ex2 = show_seq (fun x -> x) ["hello"; "world"]
(* ex2 : char seq = "[hello, world]" *)
```

Universal Collection Type

```
type ('elem, 'prop) coll = ...
```

This is it

Properties

```
type keep_all
type keep_last
type order_seq
type order_sorted
```

Type Aliases

```
type       'a seq = ('a      , keep_all      * order_seq) coll
type ('k, 'v) map = ('k * 'v, keep_last_key * _        ) coll
```

“Don’t care”

Operations

```
letop empty   : ('a, 'p) coll
letop append  : ('a, 'p) coll -> 'a -> ('a, 'p) coll
letop prepend : 'a -> ('a, 'p) coll -> ('a, 'p) coll
letop foldl   : ('acc -> 'a -> 'acc) -> 'acc -> ('a, 'p) coll -> 'acc
```

reprs and impls

```
letrepr rlist {('a, _) ucoll = 'a list}
```

Representation:
abstract type ⇒ concrete type

```
letimpl[0.0] empty : !rlist = []
letimpl[1.0] prepend : _ -> !rlist -> !rlist = fun head tail ->
  head :: tail
letimpl[n] append : !rlist -> _ -> !rlist = fun init last ->
  List.fold_right (fun h t -> h :: t) init [last]
letimpl[n] foldl : _ -> _ -> !rlist -> _ = List.fold_left
```

cost
operation
type
body

```
letimpl[1.0] map = fun f xs ->
  foldl (fun acc x -> @n append acc (f x)) empty xs
letimpl[1.0] map = fun f xs ->
  foldr (fun x acc -> @n prepend (f x) acc) empty xs
```

Implementations can use other operations

Thanks for listening!