Fun with higher-order functions: continuations 2

Jacob Thomas Errington

20 March 2019

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion?

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!
- How to convert to continuation-passing style.
  1. Change the type.
     `... -> A` becomes

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!
- How to convert to continuation-passing style.
    1. Change the type.
       `... -> A` becomes `... -> (A -> 'r) -> 'r`.

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!
- How to convert to continuation-passing style.
  1. Change the type.
     `... -> A` becomes `... -> (A -> 'r) -> 'r`.
  2. Change the implementation.
     - Instead of returning?

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!
- How to convert to continuation-passing style.
  1. Change the type.
     `... -> A` becomes `... -> (A -> 'r) -> 'r`.
  2. Change the implementation.
     - Instead of returning? Call the continuation!

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!
- How to convert to continuation-passing style.
  1. Change the type.
     `... -> A` becomes `... -> (A -> 'r) -> 'r`.
  2. Change the implementation.
     - Instead of returning? Call the continuation!
     - Work that happens after the recursive call goes where?

Recap

On Monday, we saw a few things.

- Why do we care about tail recursion? Stack space is limited!
- How to convert to continuation-passing style.
  1. Change the type.
     `... -> A` becomes `... -> (A -> 'r) -> 'r`.
  2. Change the implementation.
     - Instead of returning? Call the continuation!
     - Work that happens after the recursive call goes where? In the continuation!

Exercise: find all

Last class, we saw how to find *one* element of a tree satisfying a predicate. What if we want to find *all* elements of a tree satisfying a predicate?

Exercise: find all

Last class, we saw how to find *one* element of a tree satisfying a predicate. What if we want to find *all* elements of a tree satisfying a predicate?

In five minutes, write the function
`find_all : ('a -> bool) -> 'a tree -> 'a list`
that finds all elements of the tree satisfying the predicate.

Hint: use the `@` operator to concatenate the two lists resulting from the recursive calls:
`[1;2] @ [3;4]` $\mapsto$ `[1;2;3;4]`.

Recall:

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Exercise: convert to CPS

Now in five minutes, write `find_all_k`, a CPS version of `find_all`.

Exercise: convert to CPS

Now in five minutes, write `find_all_k`, a CPS version of `find_all`.

Remember the basic strategy:

- Instead of returning, call the continuation.
- Work that would go after the recursive call(s) goes into the continuation.

Regular expressions

What is a regular expression?

A regular expression (regex) is a way of defining a set of *strings*.

For example, to represent the set $\{\text{apple}, \text{apply}\}$, we can write the regular expression $\text{appl}(\text{e}|\text{y})$.

In general, regular expressions are defined inductively.

- Each individual letter is a regex.
  The regex c consisting of a character c represents the singleton set $\{c\}$.

In general, regular expressions are defined inductively.

- Each individual letter is a regex.
  The regex c consisting of a character c represents the singleton set $\{c\}$.
- If R_1 and R_2 are regexes, then $R_1|R_2$ is a regex.
  It represents the union of the two sets.
  For example, "apple" matches the regex apple|pear.

In general, regular expressions are defined inductively.

- Each individual letter is a regex.
  The regex c consisting of a character c represents the singleton set $\{c\}$.
- If R_1 and R_2 are regexes, then $R_1|R_2$ is a regex.
  It represents the union of the two sets.
  For example, "apple" matches the regex apple|pear.
- If R_1 and R_2 are regexes, then $R_1\,R_2$ is a regex.
  It represents the concatenation of strings.
  For example, if "app" matches R_1 and "le" matches R_2, then "apple" matches $R_1\,R_2$.

In general, regular expressions are defined inductively.

- Each individual letter is a regex.
  The regex c consisting of a character c represents the singleton set $\{c\}$.
- If R_1 and R_2 are regexes, then $R_1|R_2$ is a regex.
  It represents the union of the two sets.
  For example, "apple" matches the regex apple|pear.
- If R_1 and R_2 are regexes, then $R_1\, R_2$ is a regex.
  It represents the concatenation of strings.
  For example, if "app" matches R_1 and "le" matches R_2, then "apple" matches $R_1\, R_2$.
- If R is a regex, then R^* is a regex.
  It represents repetition of a string zero or more times.
  Consider the regex ba(na)*. The following strings match it: "ba", "bana", "banana", "bananana", ...

Plus two special cases

- The empty *regex*, $\emptyset$, which represents the empty set. No string matches this regex.

Plus two special cases

- The empty *regex*, $\emptyset$, which represents the empty set. No string matches this regex.
- The empty *string*, ϵ. This is useful for creating optional parts in a regex.
  For example, the strings "great" and "greatest" match the regex great(est$|\epsilon$).

Our goal

Input: a regex and a string
Output: whether the string matches the regex.

Our goal

Input: a regex and a string
Output: whether the string matches the regex.

Intuitively, we want to implement a function of type
`string -> regex -> bool`

Our goal

Input: a regex and a string
Output: whether the string matches the regex.

Intuitively, we want to implement a function of type
`string -> regex -> bool`

But first, what's `regex`?

Defining `regex` in OCaml

It turns out that encoding inductive definitions in OCaml is a piece of cake.

Defining `regex` in OCaml

It turns out that encoding inductive definitions in OCaml is a piece of cake.

```
type regex =
  | Epsilon (* empty string *)
  | Empty   (* empty regex *)
  | Single of char
  | Cat of regex * regex
  | Alt of regex * regex
  | Star of regex
```

The regex $b(a)^*$ is represented in code as
`let r1 = Cat (Single 'b', Star (Single 'a')).`

A few more things to clear up

`string` doesn't have *structure*! So instead, we'll use a `char list` as input.

A few more things to clear up

`string` doesn't have *structure*! So instead, we'll use a `char list` as input.

We will *generalize* the matching algorithm. Rather than check whether the *whole string* matches the regex, we will check whether a *prefix* of the string matches the regex. If so, we return the remaining characters of the string.

What return type?

For example,

- `accept ['b';'a';'n'] r1` returns `['n']`; the "n" is left over.
- `accept ['b';'a'] r1` returns `[]`; the matched prefix *is* the whole string.
- `accept ['e'] r1` fails, since there is no prefix match.

What return type?

For example,

- `accept ['b';'a';'n'] r1` returns `['n']`; the “n” is left over.
- `accept ['b';'a'] r1` returns `[]`; the matched prefix *is* the whole string.
- `accept ['e'] r1` fails, since there is no prefix match.

Given these examples, what should the return type be?

```
char list -> regex -> ?
```

What return type?

For example,

- `accept ['b';'a';'n'] r1` returns `['n']`; the “n” is left over.
- `accept ['b';'a'] r1` returns `[]`; the matched prefix *is* the whole string.
- `accept ['e'] r1` fails, since there is no prefix match.

Given these examples, what should the return type be?

`char list -> regex -> char list option`

What return type?

For example,

- `accept ['b';'a';'n'] r1` returns `['n']`; the "n" is left over.
- `accept ['b';'a'] r1` returns `[]`; the matched prefix *is* the whole string.
- `accept ['e'] r1` fails, since there is no prefix match.

Given these examples, what should the return type be?

`char list -> regex -> (char list -> 'r) -> (unit -> 'r) -> 'r`

And now we convert to CPS, expanding the `option` into separate success and failure continuations.

Now it's code it!

Basic type theory

`option` vs success and failure continuations

`option` vs success and failure continuations

Or, why are algebraic data types called “algebraic”?

What's a type anyway?

Let's be more precise about what a "type" is.

$$\text{Type } T ::= \mathrm{unit} \mid T_1 * T_2 \mid T_1 + T_2 \mid T_1 \to T_2$$

What's a type anyway?

Let's be more precise about what a "type" is.

$$\text{Type } T ::= \text{unit} \mid T_1 * T_2 \mid T_1 + T_2 \mid T_1 \rightarrow T_2$$

`unit`

What's a type anyway?

Let's be more precise about what a "type" is.

$$\text{Type } T ::= \text{unit} \mid T_1 * T_2 \mid T_1 + T_2 \mid T_1 \to T_2$$

```
t1 * t2
```

What's a type anyway?

Let's be more precise about what a "type" is.

$$\text{Type } T ::= \text{unit} \mid T_1 * T_2 \mid T_1 + T_2 \mid T_1 \to T_2$$

```
type ('a, 'b) either = Left of 'a | Right of 'b
```

What's a type anyway?

Let's be more precise about what a "type" is.

$$\text{Type } T ::= \text{unit} \mid T_1 * T_2 \mid T_1 + T_2 \mid T_1 \rightarrow T_2$$

```
t1 -> t2
```

What's a type anyway?

Let's be more precise about what a "type" is.

$$\text{Type } T ::= \text{unit} \mid T_1 * T_2 \mid T_1 + T_2 \mid T_1 \to T_2$$

These are all the basic ways we can combine types. We can form functions with `->`, form alternatives with `+`, and we can form pairs with `*`.

Combinatorics of types

- How many values of type `unit` are there?

Combinatorics of types

- How many values of type `unit` are there?
  Just one, by definition.

Combinatorics of types

- How many values of type `unit` are there?
  Just one, by definition.
- How many values of type `unit + unit` are there?

Combinatorics of types

- How many values of type `unit` are there?
  Just one, by definition.
- How many values of type `unit + unit` are there?
  Recall that this is as if we had defined in pseudo-OCaml
  `type unit + unit = Left of unit | Right of unit.`
  How many different ways can we make values of this type?

Combinatorics of types

- How many values of type `unit` are there?
  Just one, by definition.
- How many values of type `unit + unit` are there?
  Recall that this is as if we had defined in pseudo-OCaml
  `type unit + unit = Left of unit | Right of unit.`
  How many different ways can we make values of this type?
  Two, one for each constructor, since there's only one possibility for each `unit` inside.

Combinatorics of types

- How many values of type `unit` are there?
  Just one, by definition.
- How many values of type `unit + unit` are there?
  Recall that this is as if we had defined in pseudo-OCaml
  `type unit + unit = Left of unit | Right of unit.`
  How many different ways can we make values of this type?
  Two, one for each constructor, since there's only one possibility for each `unit` inside.
- How many values of type `(unit + unit) * (unit + unit)` are there?

Combinatorics of types

- How many values of type `unit` are there?
  Just one, by definition.
- How many values of type `unit + unit` are there?
  Recall that this is as if we had defined in pseudo-OCaml
  `type unit + unit = Left of unit | Right of unit`.
  How many different ways can we make values of this type?
  Two, one for each constructor, since there's only one possibility for each `unit` inside.
- How many values of type `(unit + unit) * (unit + unit)` are there?
  Four. Two possibilities for each component of the pair.

Combinatorics of types

- Suppose type `A` has n values. How many values of type `unit -> A` are there?

Combinatorics of types

- Suppose type `A` has n values. How many values of type `unit -> A` are there?
  Remember, the values of this type are *functions*. How many different functions can you write having this type?
  `let f (() : unit) : A = (*value of type A *)`

Combinatorics of types

- Suppose type `A` has n values. How many values of type `unit -> A` are there?
  Remember, the values of this type are *functions*. How many different functions can you write having this type?
  `let f (() : unit) : A = (*value of type A *)`
  You can write n different functions; one for each value in `A`.

Combinatorics of types

- Suppose type `A` has n values. How many values of type `unit -> A` are there?
  Remember, the values of this type are *functions*. How many different functions can you write having this type?
  `let f (() : unit) : A = (*value of type A *)`
  You can write n different functions; one for each value in `A`.
- Suppose type `A` has k values and `B` has n values.
  How many values of type `A -> B` are there?
  Hint: to determine the function, we have to choose for each input what its output is. In other words, how many input-output pairs are there?

Combinatorics of types

- Suppose type `A` has n values. How many values of type `unit -> A` are there?
  Remember, the values of this type are *functions*. How many different functions can you write having this type?
  `let f (() : unit) : A = (*value of type A *)`
  You can write n different functions; one for each value in `A`.
- Suppose type `A` has k values and `B` has n values.
  How many values of type `A -> B` are there?
  Hint: to determine the function, we have to choose for each input what its output is. In other words, how many input-output pairs are there?
  There are n^k different such functions.

The algebra of types

Now we understand the type constructors `*`, `+`, and `->` through their combinatorics, i.e. by *counting* the values.

The algebra of types

Now we understand the type constructors `*`, `+`, and `->` through their combinatorics, i.e. by *counting* the values.

Upshot: we can now use our knowledge of arithmetic to refactor types!

Isomorphic types

For example, unit $\rightarrow A$ has the same number of values as A, because $n^1 = n$.

This suggests that A and unit $\rightarrow A$ are isomorphic types; we can convert from one to the other and back.

Isomorphic types

For example, unit $\to A$ has the same number of values as A, because $n^1 = n$.

This suggests that A and unit $\to A$ are isomorphic types; we can convert from one to the other and back.

Proof:

```
let oneway (f : unit -> 'a) -> 'a =
  f ()

let otherway (x : 'a) : unit -> 'a =
  fun () -> x
```

All algebra you know now applies to types

$$\text{unit} * \text{unit} \cong \,?$$

All algebra you know now applies to types

$$\text{unit} * \text{unit} \cong \text{unit}$$

All algebra you know now applies to types

unit $*$ unit $\cong$ unit
because $1 \times 1 = 1$

All algebra you know now applies to types

$$(\text{unit} + \text{unit}) * (\text{unit} + \text{unit}) \cong ?$$

All algebra you know now applies to types

$$(\text{unit} + \text{unit}) * (\text{unit} + \text{unit}) \cong \text{unit} + \text{unit} + \text{unit} + \text{unit}$$

All algebra you know now applies to types

$$(\text{unit} + \text{unit}) * (\text{unit} + \text{unit}) \cong \text{unit} + \text{unit} + \text{unit} + \text{unit}$$

because $2 \times 2 = 4$

All algebra you know now applies to types

$$(A * B) \to C \cong A \to B \to C$$

All algebra you know now applies to types

$(A * B) \to C \cong A \to B \to C$
because $n^{k_1 \times k_2} = (n^{k_2})^{k_1}$

All algebra you know now applies to types

$(A * B) \to C \cong A \to B \to C$
because $n^{k_1 \times k_2} = (n^{k_2})^{k_1}$
This is a combinatorial justification for currying

Refactoring `option`

Now we can fully understand the connection between `option` and success / failure continuations.

Refactoring `option`

Now we can fully understand the connection between `option` and success / failure continuations.

First, notice that A option is the same as unit $+ A$.

- `None` is represented by the `unit` in the left branch.

Refactoring `option`

Now we can fully understand the connection between `option` and success / failure continuations.

First, notice that A option is the same as unit $+ A$.

- `None` is represented by the `unit` in the left branch.
- `Some x` holds a value `x : A` and that's in the right branch.

Refactoring `option`

1. A option

Refactoring `option`

1. A option
2. $(\mathrm{unit} + A)$

Refactoring `option`

1. A option
2. $(\text{unit} + A)$
   Since A option $\cong \text{unit} + A$

Refactoring `option`

1. A option
2. $(\text{unit} + A)$
   Since A option $\cong \text{unit} + A$
3. $((\text{unit} + A) \to R) \to R$
   by CPS conversion.

Refactoring `option`

1. A option
2. $(\text{unit} + A)$
   Since A option $\cong$ unit $+ A$
3. $((\text{unit} + A) \to R) \to R$
   by CPS conversion.
4. $((\text{unit} \to R) * (A \to R)) \to R$

Refactoring `option`

1. A option
2. $(\text{unit} + A)$
   Since A option $\cong \text{unit} + A$
3. $((\text{unit} + A) \to R) \to R$
   by CPS conversion.
4. $((\text{unit} \to R) * (A \to R)) \to R$
   because $n^{k_1 + k_2} = n^{k_1} \times n^{k_2}$.

Refactoring `option`

1. A option
2. $(\text{unit} + A)$
   Since A option $\cong$ unit $+ A$
3. $((\text{unit} + A) \to R) \to R$
   by CPS conversion.
4. $((\text{unit} \to R) * (A \to R)) \to R$
   because $n^{k_1+k_2} = n^{k_1} \times n^{k_2}$.
5. $(\text{unit} \to R) \to (A \to R) \to R$

Refactoring `option`

1. A option
2. $(\text{unit} + A)$
   Since A option $\cong$ unit $+ A$
3. $((\text{unit} + A) \to R) \to R$
   by CPS conversion.
4. $((\text{unit} \to R) * (A \to R)) \to R$
   because $n^{k_1+k_2} = n^{k_1} \times n^{k_2}$.
5. $(\text{unit} \to R) \to (A \to R) \to R$
   by currying.