From Data-Oriented Designs to Algorithms and Back

José Pedro Correia   João Saraiva   José Nuno Oliveira

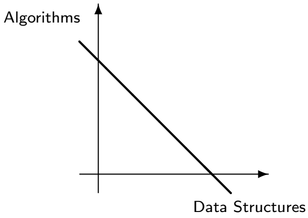
Departamento de Informática
Universidade do Minho

October 12, 2006

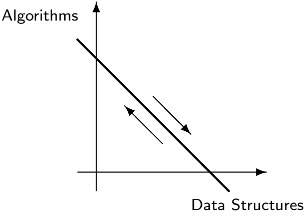
1. Motivation
2. From data-structures to algorithms and back
   - Case study
   - From data-structures to algorithms
   - From algorithms to data-structures
   - Generalizations
3. Recursion removal
   - Case study
   - Introduction of a stack by calculation
   - Introduction of a stack by “intuition”
   - Discussion
4. Future work

Programs = Algorithms + Data Structures

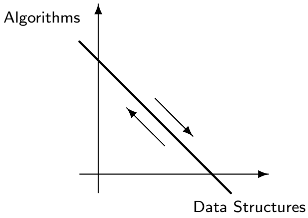
Programs = Algorithms + Data Structures



Programs = Algorithms + Data Structures



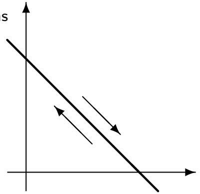
Programs = Algorithms + Data Structures



Conversions

1. From data-structure oriented designs to algorithmic oriented designs

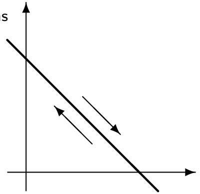
Programs = Algorithms + Data Structures



Conversions

1. From data-structure oriented designs to algorithmic oriented designs $\equiv$ Specialization

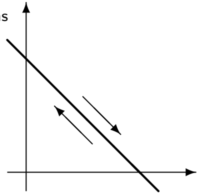
Programs = Algorithms + Data Structures



Conversions

1. From data-structure oriented designs to algorithmic oriented designs ≡ Specialization
2. From algorithmic oriented designs to data-structure dependant designs

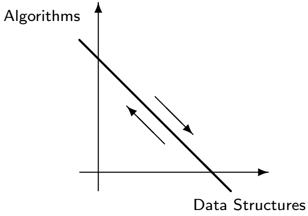
Programs = Algorithms + Data Structures



Conversions

1. From data-structure oriented designs to algorithmic oriented designs ≡ Specialization
2. From algorithmic oriented designs to data-structure dependant designs ≡ Generalization

Programs = Algorithms + Data Structures



Conversions

1. From data-structure oriented designs to algorithmic oriented designs ≡ Specialization
2. From algorithmic oriented designs to data-structure dependant designs ≡ Generalization

Case study

- LL(1) language recognition

LL(1) language recognition

- Let us define a language using a context free grammar (CFG) like the following:

$$
\begin{array}{ccl}
S & \longrightarrow & dAd \\
A & \longrightarrow & aA \\
 & | & BA \\
 & | & \varepsilon \\
B & \longrightarrow & bB \\
 & | & t
\end{array}
$$

LL(1) language recognition

- Let us define a language using a context free grammar (CFG) like the following:

$$\begin{array}{rcl} S & \rightarrow & dAd \\ A & \rightarrow & aA \\ & \mid & BA \\ & \mid & \varepsilon \\ B & \rightarrow & bB \\ & \mid & t \end{array}$$

- From this, one can conceive a function to test if a given string belongs to the language in two common fashions:

LL(1) language recognition

Table driven

A representation of a transition table ("large" data-structure) and a function to consult it ("small" algorithm):

LL(1) language recognition

Table driven

A representation of a transition table ("large" data-structure) and a function to consult it ("small" algorithm):

Transition table

	a	b	t	d
S	$error$	$error$	$error$	dAd
A	aA	BA	BA	ε
B	$error$	bB	t	$error$

LL(1) language recognition

Table driven

Fnal program (where `tt` refers to the transition table):

Table driven recognizer

```
dss = ("abtd", "SAB", tt)

accept inp = aux dss ['S'] inp

aux _ [] [] = True
aux _ _  [] = False
aux _ [] _  = False
aux dss@(t,nt,ft) (ts:rs) (ti:ri)
  | (ts `elem` t) && (ti == ts) = aux dss rs ri
  | ts `elem` nt = maybe False (\rhs -> aux dss (rhs ++ rs) (ti:ri))
                          $ ft (ts,ti)
  | otherwise = False
```

LL(1) language recognition

Recursive descendant

A set of mutually recursive functions ("large" algorithm):

LL(1) language recognition

Recursive descendant

A set of mutually recursive functions ("large" algorithm):

Recursive descendant recognizer

```
accept inp = let (v,ri) = recognize_S inp in if (null ri) then v else False

recognize_S inp@('d':ri) =
    let (v1,ri1) = recognize_d inp
        (v2,ri2) = recognize_A ri1
        (v3,ri3) = recognize_d ri2
     in if (v1 && v2) then (v3,ri3) else (False,inp)
recognize_S inp = (False,inp)

recognize_A inp@('a':ri) =
    let (v1,ri1) = recognize_a inp
        (v2,ri2) = recognize_A ri1
     in if v1 then (v2,ri2) else (False,inp)
recognize_A inp@('b':ri) =
    let (v1,ri1) = recognize_B inp
        (v2,ri2) = recognize_A ri1
     in if v1 then (v2,ri2) else (False,inp)
```

Continues. . .

LL(1) language recognition

Recursive descendant

Recursive descendant recognizer

```
recognize_A inp@('t':ri) =
    let (v1,ri1) = recognize_B inp
        (v2,ri2) = recognize_A ri1
     in if v1 then (v2,ri2) else (False,inp)
recognize_A inp@('d':ri) = recognize_ inp
recognize_A inp = (False,inp)

recognize_B inp@('b':ri) =
    let (v1,ri1) = recognize_b inp
        (v2,ri2) = recognize_B ri1
     in if v1 then (v2,ri2) else (False,inp)
recognize_B inp@('t':ri) = recognize_t inp
recognize_B inp = (False,inp)

recognize_ inp = (True,inp)
```

How to specialize a table driven parser?

How to specialize a table driven parser?

One can calculate *a* recursive descendant recognizer from a table driven recognizer, by the application of:

How to specialize a table driven parser?

One can calculate *a* recursive descendant recognizer from a table driven recognizer, by the application of:

Partial evaluation

What Specialization of a program (function) with respect to a static (known) input

How to specialize a table driven parser?

One can calculate *a* recursive descendant recognizer from a table driven recognizer, by the application of:

Partial evaluation

What Specialization of a program (function) with respect to a static (known) input

Result Set of functions specialized for every possible static input generated by the initial call

Partial evaluation

Let's take a look at a simple example:

Partial evaluation

Let's take a look at a simple example:

Example

The following function:

```
power 0 x = 1
power x n = x * power (n-1) x
```

partially evaluated for the call `power 3 x` yields:

Partial evaluation

Let's take a look at a simple example:

Example

The following function:

```
power 0 x = 1
power x n = x * power (n-1) x
```

partially evaluated for the call `power 3 x` yields:

```
power_3 x = x * power_2 x
```

Partial evaluation

Let's take a look at a simple example:

Example

The following function:

```
power 0 x = 1
power x n = x * power (n-1) x
```

partially evaluated for the call `power 3 x` yields:

```
power_3 x = x * power_2 x

power_2 x = x * power_1 x

power_1 x = x * power_0 x

power_0 x = 1
```

How to specialize a table driven parser?

Application of partial evaluation

Returning to our case study, let's recall our table driven recognizer. . .

LL(1) language recognition

Table driven

Fnal program (where `tt` refers to the transition table):

Table driven recognizer

```
dss = ("abtd", "SAB", tt)

accept inp = aux dss ['S'] inp

aux _ [] [] = True
aux _ _  [] = False
aux _ [] _  = False
aux dss@(t,nt,ft) (ts:rs) (ti:ri)
  | (ts `elem` t) && (ti == ts) = aux dss rs ri
  | ts `elem` nt = maybe False (\rhs -> aux dss (rhs ++ rs) (ti:ri))
                         $ ft (ts,ti)
  | otherwise = False
```

How to specialize a table driven parser?

Application of partial evaluation

- For LL(1) language recognition one can partially evaluate the auxiliary function on `dss` and on the initial state of the stack

How to specialize a table driven parser?

Application of partial evaluation

- For LL(1) language recognition one can partially evaluate the auxiliary function on `dss` and on the initial state of the stack

Calculated recursive descendant

```
accept inp = aux_dss_S inp

aux_dss_S ('d':ri) = aux_dss_Ad ri
aux_dss_S _ = False

aux_dss_Ad ('a':ri) = aux_dss_Ad ri
aux_dss_Ad ('b':ri) = aux_dss_BAd ri
aux_dss_Ad ('t':ri) = aux_dss_Ad ri
aux_dss_Ad ('d':ri) = aux_dss_ ri
aux_dss_Ad _ = False

aux_dss_BAd ('b':ri) = aux_dss_BAd ri
aux_dss_BAd ('t':ri) = aux_dss_Ad ri
aux_dss_BAd _ = False

aux_dss_ [] = True
aux_dss_ _  = False
```

How to specialize a table driven parser?

Application of partial evaluation

We've calculated a recognizer that:

How to specialize a table driven parser?

Application of partial evaluation

We've calculated a recognizer that:

- Does not use auxiliary data-structures

How to specialize a table driven parser?

Application of partial evaluation

We've calculated a recognizer that:

- Does not use auxiliary data-structures
- It's tail recursive (very efficient)

How to specialize a table driven parser?

Application of partial evaluation

We've calculated a recognizer that:

- Does not use auxiliary data-structures
- It's tail recursive (very efficient)

But it's not quite like the one built "directly" from the grammar...

How to go back?
Building a table

One can build a table to capture the behaviour of the calculated recursive descendant using the following approach:

- Each row represents a function

How to go back?
Building a table

One can build a table to capture the behaviour of the calculated recursive descendant using the following approach:

- Each row represents a function
- Each column represents a possible input

How to go back?

Building a table

One can build a table to capture the behaviour of the calculated recursive descendant using the following approach:

- Each row represents a function
- Each column represents a possible input
- Each table entry in position (i, j) is the behaviour of function i for input j. This can be either:

How to go back?

Building a table

One can build a table to capture the behaviour of the calculated recursive descendant using the following approach:

- Each row represents a function
- Each column represents a possible input
- Each table entry in position (i, j) is the behaviour of function i for input j. This can be either:
  - A constant value

How to go back?

Building a table

One can build a table to capture the behaviour of the calculated recursive descendant using the following approach:

- Each row represents a function
- Each column represents a possible input
- Each table entry in position (i, j) is the behaviour of function i for input j. This can be either:
  - A constant value
  - A pair of a recursive reference and a non-recursive pre-processing of the input

How to go back?
Building a table

This technique applied to our case study yields the following:

Table

	('a':xs)	('b':xs)	('t':xs)	('d':xs)	[]
f_0 (aux_dss_S)	False	False	False	(f_1,tail)	False
f_1 (aux_dss_Ad)	(f_1,tail)	(f_2,tail)	(f_1,tail)	(f_3,tail)	False
f_2 (aux_dss_BAd)	False	(f_2,tail)	(f_1,tail)	False	False
f_3 (aux_dss_)	False	False	False	False	True

How to go back?

Final result

Given a representation of the former table as a function (named `tt`) we can write our acceptance function as:

Final program

```
accept inp = Just (f (tt,"f0") inp)

f (ft,k) d = aux (ft (k,d)) d
  where aux (Left c) d = c
        aux (Right (kr,h)) d = f (ft,kr) (h d)
```

How to go back?
Discussion

We have

- "Extracted" a table from the recursive structure of the functions
- Obtained, thus, *a* table driven recognizer

But

How to go back?
Discussion

We have

- "Extracted" a table from the recursive structure of the functions
- Obtained, thus, *a* table driven recognizer

But

- It's not quite like the one built "directly" from the grammar
- The function to manipulate the table is introduced *ad hoc*, as we know the desired type of the table
- The table contains functions that depend on the input, so it is not completely static

Definitions

"Inspired" by the design of the LL(1) table driven recognizer, let us define a data-oriented design as follows:

Definitions

"Inspired" by the design of the LL(1) table driven recognizer, let us define a data-oriented design as follows:

Data-oriented design

Definition A data-oriented design (DOD) is a tuple $(ds : T, p : D \to O, f : T \to D \to R)$, where:

- ds is a statically known data-structure of type T
- p is the top-level function of the program that takes a dynamical input of type D
- p is expressed as $p = k \cdot f(ds)$
- f is the function that "deals" with values of type T

Definitions

"Inspired" by the design of the LL(1) recursive descendant recognizer, let us define a algorithmic-oriented design as follows:

Definitions

"Inspired" by the design of the LL(1) recursive descendant recognizer, let us define a algorithmic-oriented design as follows:

Algorithmic-oriented design

Definition An algorithmic-oriented design (AOD) is a tuple $(p : D \to O, fs : (D \to R)^*)$

- p is the top-level function of the program that takes a dynamical input of type D
- fs is a set of mutually recursive functions
- given $f_0 \in fs$, p is expressed as $p = k \cdot f_0$

DOD to AOD

Is partial evaluation still applicable?

- Partial evaluation aims to, from $f : S \rightarrow D \rightarrow O$ and a call $f(s_1)$, obtain a function $f_{s_1} : D \rightarrow O$ such that:

$$f_{s_1}(d) = f(s_1)(d)$$

DOD to AOD

Is partial evaluation still applicable?

- Partial evaluation aims to, from $f : S \rightarrow D \rightarrow O$ and a call $f(s_1)$, obtain a function $f_{s_1} : D \rightarrow O$ such that:

$$f_{s_1}(d) = f(s_1)(d) \equiv f_{s_1} = f(s_1)$$

DOD to AOD

Is partial evaluation still applicable?

- Partial evaluation aims to, from $f : S \rightarrow D \rightarrow O$ and a call $f(s_1)$, obtain a function $f_{s_1} : D \rightarrow O$ such that:
$$f_{s_1}(d) = f(s_1)(d) \equiv f_{s_1} = f(s_1)$$
- From our definition of a DOD, $p = k \cdot f(ds)$ can then be partially evaluated yielding $p = k \cdot f_{ds}$

DOD to AOD

Is partial evaluation still applicable?

- Partial evaluation aims to, from $f : S \rightarrow D \rightarrow O$ and a call $f(s_1)$, obtain a function $f_{s_1} : D \rightarrow O$ such that:

$$f_{s_1}(d) = f(s_1)(d) \equiv f_{s_1} = f(s_1)$$

- From our definition of a DOD, $p = k \cdot f(ds)$ can then be partially evaluated yielding $p = k \cdot f_{ds}$
- Moreover, as we saw before, partially evaluation of $f_{ds} : D \rightarrow R$ produces a set of, also specialized functions, $f_* : (D \rightarrow R)^*$

DOD to AOD

Is partial evaluation still applicable?

- Partial evaluation aims to, from $f : S \rightarrow D \rightarrow O$ and a call $f(s_1)$, obtain a function $f_{s_1} : D \rightarrow O$ such that:

$$f_{s_1}(d) = f(s_1)(d) \equiv f_{s_1} = f(s_1)$$

- From our definition of a DOD, $p = k \cdot f(ds)$ can then be partially evaluated yielding $p = k \cdot f_{ds}$
- Moreover, as we saw before, partially evaluation of $f_{ds} : D \rightarrow R$ produces a set of, also specialized functions, $f_* : (D \rightarrow R)^*$
- Thus we have an AOD where $f_0 = f_{ds}$ and $fs = f_*$

AOD to DOD

Building a table

Generalizing the strategy used for LL(1) recognition, one can build a table to capture the relations between the functions in $fs : (D \rightarrow R)^*$ using the following criteria:

- Each function $f_i \in fs$ yields a row identified by $iden(f_i)$ [1]

[1] where $iden : (D \rightarrow R) \rightarrow I$

AOD to DOD

Building a table

Generalizing the strategy used for LL(1) recognition, one can build a table to capture the relations between the functions in $fs : (D \rightarrow R)^*$ using the following criteria:

- Each function $f_i \in fs$ yields a row identified by $iden(f_i)$ [1]
- The columns correspond to all the possible input patterns for all $f_i \in fs$

[1] where $iden : (D \rightarrow R) \rightarrow I$

AOD to DOD

Building a table

Generalizing the strategy used for LL(1) recognition, one can build a table to capture the relations between the functions in $fs : (D \rightarrow R)^*$ using the following criteria:

- Each function $f_i \in fs$ yields a row identified by $iden(f_i)$ [1]
- The columns correspond to all the possible input patterns for all $f_i \in fs$
- Each table entry in position $(iden(f_i), d_j)$ represents the behaviour of function f_i for input d_j by a sequence of references, wich are either:
  - A function $g : D \rightarrow R$ where $g \notin fs$

[1] where $iden : (D \rightarrow R) \rightarrow I$

AOD to DOD

Building a table

Generalizing the strategy used for LL(1) recognition, one can build a table to capture the relations between the functions in $fs : (D \to R)^*$ using the following criteria:

- Each function $f_i \in fs$ yields a row identified by $iden(f_i)$ [1]
- The columns correspond to all the possible input patterns for all $f_i \in fs$
- Each table entry in position $(iden(f_i), d_j)$ represents the behaviour of function f_i for input d_j by a sequence of references, wich are either:
  - A function $g : D \to R$ where $g \notin fs$
  - A function $h : D \to D$ where $h \notin fs$

[1] where $iden : (D \to R) \to I$

AOD to DOD

Building a table

Generalizing the strategy used for LL(1) recognition, one can build a table to capture the relations between the functions in $fs : (D \rightarrow R)^*$ using the following criteria:

- Each function $f_i \in fs$ yields a row identified by $iden(f_i)$ [1]
- The columns correspond to all the possible input patterns for all $f_i \in fs$
- Each table entry in position $(iden(f_i), d_j)$ represents the behaviour of function f_i for input d_j by a sequence of references, wich are either:
    - A function $g : D \rightarrow R$ where $g \notin fs$
    - A function $h : D \rightarrow D$ where $h \notin fs$
    - A value $k \in I$ for calls to $f_k \in fs$ such that $k = iden(f_k)$

[1] where $iden : (D \rightarrow R) \rightarrow I$

AOD to DOD

Final program

After building the table that captures the relations in *fs* (let's name it *tt*), we now need a function to "work" with it:

Auxiliary function

```
f (ft,k) d = (composeAll . map aux) (ft (k,d)) $ d
  where aux (Left g) = g
        aux (Right (Left h)) = h
        aux (Right (Right k1)) = f (ft,k1)
        composeAll = foldr1 (.)
```

AOD to DOD

Final program

After building the table that captures the relations in *fs* (let's name it *tt*), we now need a function to "work" with it:

Auxiliary function

```
f (ft,k) d = (composeAll . map aux) (ft (k,d)) $ d
  where aux (Left g) = g
        aux (Right (Left h)) = h
        aux (Right (Right k1)) = f (ft,k1)
        composeAll = foldr1 (.)
```

We then just define $p = k \cdot f(tt, iden(f_0))$ and we have a DOD that corresponds to the initial AOD

AOD to DOD

Final program

After building the table that captures the relations in *fs* (let's name it *tt*), we now need a function to "work" with it:

Auxiliary function

```
f (ft,k) d = (composeAll . map aux) (ft (k,d)) $ d
  where aux (Left g) = g
        aux (Right (Left h)) = h
        aux (Right (Right k1)) = f (ft,k1)
        composeAll = foldr1 (.)
```

We then just define $p = k \cdot f(tt, iden(f_0))$ and we have a DOD that corresponds to the initial AOD (no proof yet!)

AOD to DOD

Discussion

We have a strategy for conversion, but:

- It's based on intuition by generalizing the LL(1) strategy
- It imposes restrictions on the covered AOD's, namely:

AOD to DOD

Discussion

We have a strategy for conversion, but:

- It's based on intuition by generalizing the LL(1) strategy
- It imposes restrictions on the covered AOD's, namely:
    - Control flow has to rely only in pattern matchings
    - Every function $f_i \in fs$ should be entire
    - Calls must be sequential compositions of functions applied to the input
    - References to other functions from *fs* must be "direct"

AOD to DOD

Discussion

We have a strategy for conversion, but:

- It's based on intuition by generalizing the LL(1) strategy
- It imposes restrictions on the covered AOD's, namely:
  - Control flow has to rely only in pattern matchings
  - Every function $f_i \in fs$ should be entire
  - Calls must be sequential compositions of functions applied to the input
  - References to other functions from *fs* must be "direct"
- The table contains functions that depend on the input, so it is not completely static

Postorder tree traversal

Let's consider the following Haskell datatype and a postorder traversal function over it:

Example

```
data T a = T1 | T2 a (T a) | T3 a (Ta) (T a) (T a)

postorder :: T a -> [a]
postorder T1              = []
postorder (T2 a t1)       = postorder t1 ++ [a]
postorder (T3 a t1 t2 t3) = postorder t1 ++
                            postorder t2 ++
                            postorder t3 ++ [a]
```

Postorder tree traversal

Let's consider the following `Haskell` datatype and a postorder traversal function over it:

Example

```
data T a = T1 | T2 a (T a) | T3 a (Ta) (T a) (T a)

postorder :: T a -> [a]
postorder T1              = []
postorder (T2 a t1)       = postorder t1 ++ [a]
postorder (T3 a t1 t2 t3) = postorder t1 ++
                            postorder t2 ++
                            postorder t3 ++ [a]
```

How can we turn this function into a tail recursive equivalent by the introduction of an auxiliary data-structure?

Adding continuations

- Continuations make the order of evaluation explicit

Adding continuations

- Continuations make the order of evaluation explicit
- The objective is to obtain a defintion `postorder' :: T a -> Cont a -> [a]` such that `postorder' t c = c (postorder t)` where `type Cont a = [a] -> [a]`

Adding continuations

- Continuations make the order of evaluation explicit
- The objective is to obtain a defintion `postorder' :: T a -> Cont a -> [a]` such that `postorder' t c = c (postorder t)` where `type Cont a = [a] -> [a]`
- This can be obtained by calculation

Adding continuations

Let's calculate the definition of `postorder'` with continuations:

Case T1

```
  postorder' T1 c
= { specification of postorder' }
  c (postorder T1)
= { definition of postorder }
  c []
```

Adding continuations

Case T2 a t1

```
    postorder' (T2 a t1) c
=   { specification of postorder' }
    c (postorder (T2 a t1))
=   { definition of postorder }
    c (postorder t1 ++ [a])
=   { abstraction over postorder t1 }
    (\x -> c (x ++ [a])) (postorder t1)
=   { specification of postorder' }
    postorder' t1 (\x -> c (x ++ [a]))
```

Adding continuations

Omitting the calculation for the case `T3 a t1 t2 t3`, one obtains:

Result

```
postorder' :: T a -> Cont a -> [a]
postorder' T1              c = c []
postorder' (T2 a t1)       c = postorder' t1 (\x -> c (x ++ [a]))
postorder' (T3 a t1 t2 t3) c =
     postorder' t1
          (\x -> postorder' t2
               (\y -> postorder' t3
                    (\z -> c (x ++ y ++ z ++ [a]))))

postorder t = postorder' t (\x -> x)
```

Defunctionalizing

What we got

- Our function is already tail-recursive, due to the passing of extra recursive calls as continuations

Defunctionalizing

What we got

- Our function is already tail-recursive, due to the passing of extra recursive calls as continuations
- Nevertheless, we would like to capture the continuations, not by functions but by a data-structure

Defunctionalizing

What we got

- Our function is already tail-recursive, due to the passing of extra recursive calls as continuations
- Nevertheless, we would like to capture the continuations, not by functions but by a data-structure

This can be done in three steps

Defunctionalizing

Collect the continuations

First we collect the different forms of continuations used

Continuations

```
c1 :: Cont a
c1 = \x -> x

c2 :: a -> Cont a -> Cont a
c2 a c = \y -> c (y ++ [a])

c3 :: a -> [a] -> [a] -> Cont a -> Cont a
c3 a x y c = \z -> c (x ++ y ++ z ++ [a])

c4 :: a -> [a] -> T a -> Cont a -> Cont a
c4 a x t3 c = \y -> postorder' t3 (c3 a x y c)

c5 :: a -> T a -> T a -> Cont a -> Cont a
c5 a t2 t3 c = \x -> postorder' t2 (c4 a x t3 c)
```

Defunctionalizing

Defining the datatype

Now we can define a datatype that represents all the continuations

Datatype

```
data CONT a =
        C1
      | C2 a (CONT a)
      | C3 a [a] [a] (CONT a)
      | C4 a [a] (T a) (CONT a)
      | C5 a (T a) (T a) (CONT a)
```

and we can represent this, from another point of view, as

```
type CONT a = [INST a]
data INST a =
        I1 a
      | I2 a [a] [a]
      | I3 a [a] (T a)
      | I4 a (T a) (T a)
```

Defunctionalizing

Putting things together

In order to keep correspondence to the former representation of continuations, we need a function that applies our datatype to a result of type `[a]`, wich results in our final version:

Final program

```
postorder' :: T a -> CONT a -> [a]
postorder' T1                c = apply c []
postorder' (T2 a t1)         c = postorder' t1 ((I1 a):c)
postorder' (T3 a t1 t2 t3) c = postorder' t1 ((I4 a t2 t3):c)

apply :: CONT a -> [a] -> [a]
apply [] r = r
apply ((I1 a):t) r = apply t (r ++ [a])
apply ((I2 a r1 r2):t) r = apply t (r1 ++ r2 ++ r ++ [a])
apply ((I3 a r1 t3):t) r = postorder' t3 ((I2 a r1 r):t)
apply ((I4 a t2 t3):t) r = postorder' t2 ((I3 a r t3):t)

postorder t = postorder' t []
```

Introduction of a stack by "intuition"

If we would try to write a postorder function with a stack to "manage" the recursive calls, one could produce something like the following:

Postorder with stack

```
postorder' :: T a -> [Either (a,T a) (a,T a,T a)] -> [a]
postorder' T1 [] = []
postorder' T1 ((Left (a,t3)):t)      = postorder' t3 t ++ [a]
postorder' T1 ((Right (a,t2,t3)):t) = postorder' t2 ((Left (a,t3)):t)
postorder' (T2 a t1) t              = postorder' t1 t ++ [a]
postorder' (T3 a t1 t2 t3) t        = postorder' t1 ((Right (a,t2,t3)):t)

postorder t = postorder' t []
```

Comparison

- The second is “partially evaluatable” to the original definition

Comparison

- The second is “partially evaluatable” to the original definition
- The first has two mutually recursive function

Comparison

- The second is “partially evaluatable” to the original definition
- The first has two mutually recursive function
- The second is just one tail-recursive function

But

Comparison

- The second is "partially evaluatable" to the original definition
- The first has two mutually recursive function
- The second is just one tail-recursive function

But

- The first is obtained by calculation

Future work

For the first section

- Address the problem of termination of partial evaluation
- Supply a proof for the general strategy of conversion AOD $\longrightarrow$ DOD

Future work

For the first section

- Address the problem of termination of partial evaluation
- Supply a proof for the general strategy of conversion AOD $\longrightarrow$ DOD

For recursion removal

- Try to make a connection to the other subject
- Determine a relation between the calculation and the "intuition"
- Analyse the relation with *derivatives of containers*