

Denotation by Transformation

Towards Obtaining a Denotational Semantics by Transformation to Point-free Style

Bernd Braßel, Jan Christiansen

Christian-Albrechts-University of Kiel

Kongens Lyngby, Denmark 23-24 August 2007

The contents of this talk:

- what point-free programs look like
- how to obtain point-free programs building on point-wise primitives
- exemplification for the functional logic language Curry

The merits of point-free style in general:

- equational reasoning about programs

The contribution of the presented approach:

- simpler transformation – results still recognizable
- first approach covering functional *logic* languages
- fewer point-wise primitives because of logical features
- in perspective: a *relation algebraic* semantics

Point-Free Style – Good for Equational Reasoning!

From “Point-free Program Calculation”, Cunha (PhD Thesis 2005)

A final contribution of the thesis is an accurate picture of the state of the art of reasoning by calculation in the point-free style, which hopefully clarifies some common misconceptions.

*Particularly relevant is the study presented in Chapter 8, that somehow **contradicts the general assumption that calculations in this style are more amenable to mechanization than those in pointwise.***

If not automatic, how about human reasoning?

Point-Free Style – Good for Equational Reasoning???

Original

```
data Nat = Zero | Succ Nat
```

```
len :: [a] -> Nat
```

```
len [] = Zero
```

```
len (h:t) = Succ (len t)
```

Translation Cunha et al.

```
len :: [a] -> Nat
```

```
len = hylo (_L :: Mu ((:++) (Const One) Id))
```

```
  (app . (((curry (app . ((either . ((curry (inn . (inl .
  bang)))) /\ (curry (inn . (inr . snd)))))) /\ snd))) .
  bang) /\ id)) (app . (((curry (app . ((either . ((curry
  (inl . bang)) /\ (curry (inr . (snd . snd)))))) /\ (out .
  snd)))) . bang) /\ id))
```

Point-Free Style – Good for Equational Reasoning!!!

Original

```
data Nat = Zero | Succ Nat
```

```
len :: [a] -> Nat
```

```
len [] = Zero
```

```
len (h:t) = Succ (len t)
```

Translation Cunha et al.

```
len :: [a] -> Nat
```

```
len = hylo (_L :: Mu ((:+:) (Const One) Id))
```

```
  (app . (((curry (app . ((either . ((curry (inn . (inl .  
    bang)))) /\ (curry (inn . (inr . snd)))))) /\ snd))) .  
  bang) /\ id)) (app . (((curry (app . ((either . ((curry  
    (inl . bang)) /\ (curry (inr . (snd . snd)))))) /\ (out .  
    snd)))) . bang) /\ id))
```

Point-Free Style – Good for Equational Reasoning???

Original

```
data Nat = Zero | Succ Nat
```

```
len :: [a] -> Nat
```

```
len [] = Zero
```

```
len (h:t) = Succ (len t)
```

Our translation

```
len :: [a] -> Nat
```

```
len = (invert nil * zero)
```

```
  ? (invert cons * snd * len * succ)
```

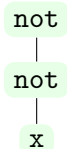
Point-Free = No Variables for Values

The Overall Scheme:

- only **selected primitives** may use variables
- define all other functions based on these primitives

A First Very Simple Example

```
involution :: Bool -> Bool  
involution x = not (not x)
```



A First Primitive: Function Composition

```
(*) :: (a -> b) -> (b -> c) -> a -> c  
(f * g) x = g (f x)
```

Point-Free Version

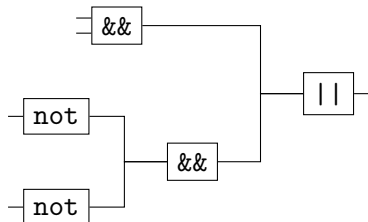
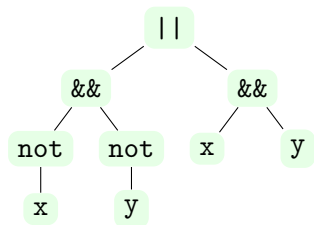
```
involution :: Bool -> Bool  
involution = not * not
```



A Complex Expression

$(\Leftrightarrow) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$x \Leftrightarrow y = (x \ \&\& \ y) \ || \ (\text{not } x \ \&\& \ \text{not } y)$

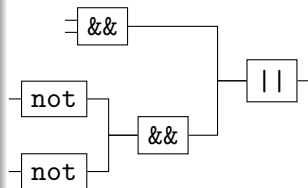


Further Examples – Further Primitives

A Complex Expression

$(\Leftrightarrow) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$x \Leftrightarrow y = (x \ \&\& \ y) \ || \ (\text{not } x \ \&\& \ \text{not } y)$

$$(\Leftrightarrow) \cong \frac{(\&\&)}{\text{not} \quad \text{not}} * (||)$$


Parallel Composition

$(/) :: (a \rightarrow c, b \rightarrow d) \rightarrow (a, c) \rightarrow (b, d)$

$(f / g) (x, y) = (f \ x, g \ y)$

Point-Free Version, $(*)$ binds stronger than $(/)$

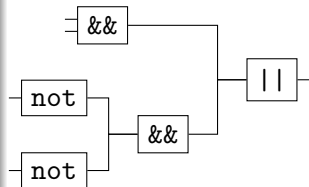
$(\Leftrightarrow) \cong ((\&\&) / (\text{not} / \text{not}) * (\&\&)) * (||)$

Further Examples – Further Primitives

A Complex Expression

$(\Leftrightarrow) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$x \Leftrightarrow y = (x \ \&\& \ y) \ || \ (\text{not } x \ \&\& \ \text{not } y)$

$$(\Leftrightarrow) \cong \frac{(\&\&)}{\text{not} \quad \text{-----} \ * \ (||)}$$
$$\frac{\text{not} \quad \text{-----} \ * \ (\&\&)}{\text{not}}$$


Parallel Composition

$(/) :: (a \rightarrow c, b \rightarrow d) \rightarrow (a, c) \rightarrow (b, d)$

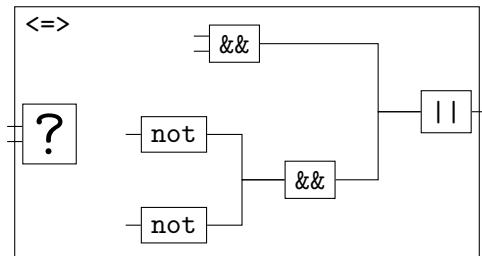
$(f / g) (x, y) = (f \ x, g \ y)$

Point-Free Version, $(*)$ binds stronger than $(/)$

$(\Leftrightarrow) \cong ((\&\&) / (\text{not} / \text{not}) * (\&\&)) * (||)$

Two Inputs vs. Four Inputs

$(\Leftarrow\Rightarrow) \cong \text{fork} * ((\&\&) / (\text{not} / \text{not}) * (\&\&)) * (||)$



Forking Arguments

```
fork :: a -> (a,a)
fork x = (x,x)
```



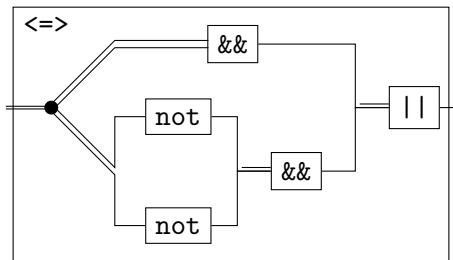
... and not Forking them

```
id :: a -> a
id x = x
```



Two Inputs vs. Four Inputs

$(\Leftarrow) \cong \text{fork} * ((\&\&) / (\text{not} / \text{not}) * (\&\&)) * (||)$



Forking Arguments

`fork :: a -> (a,a)`
`fork x = (x,x)`



... and not Forking them

`id :: a -> a`
`id x = x`



No Curried Functions – If Possible

Type of ($\langle \Rightarrow \rangle$) Changed – but still isomorphic

$\langle \Rightarrow \rangle :: (\text{Bool}, \text{Bool}) \rightarrow \text{Bool}$

$\langle \Rightarrow \rangle = \text{fork} * ((\&\&) / (\text{not} / \text{not}) * (\&\&)) * (||)$

In General: Curried \mapsto Uncurried

$\&\& :: (\text{Bool}, \text{Bool}) \rightarrow \text{Bool}$

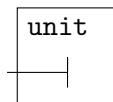
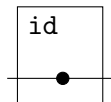
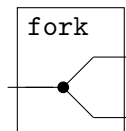
$|| :: (\text{Bool}, \text{Bool}) \rightarrow \text{Bool}$

Primitives for Interface Adaption

$\text{fork} :: a \rightarrow (a, a)$

$\text{id} :: a \rightarrow a$

$\text{unit} :: a \rightarrow ()$



Data Types and Pattern Matching

One Function for...

```
data Bool = True -> False
true, false :: () -> Bool
true () = True
false () = False
```

...Each Constructor

```
data [a] = [] | a:[a]
nil () = []
cons :: (a,[a]) -> [a]
cons (x,xs) = x:xs
```

Primitive for Pattern Matching

```
invert :: (a -> b) -> (b -> a)
invert f = f' where f' (f x) = x
```

Matching Boolean Values

```
invert true ≡ invert () = True
not = invert true * false
not = invert false * true
```

Matching Lists

```
head = invert cons * fst
tail = invert cons * snd
```

Data Types and Pattern Matching

One Function for...

```
data Bool = True -> False
true, false :: () -> Bool
true () = True
false () = False
```

...Each Constructor

```
data [a] = [] | a:[a]
nil () = []
cons :: (a, [a]) -> [a]
cons (x,xs) = x:xs
```

Primitive for Pattern Matching

```
invert :: (a -> b) -> (b -> a)
invert f = f' where f' (f x) = x
```

Matching Boolean Values

```
invert true  $\equiv$  invert () = True
not = invert true * false
not = invert false * true
```

Matching Lists

```
head = invert cons * fst
tail = invert cons * snd
```

Invert is a Jack-of-all-Trades

Definition of join

```
join :: (a,a) -> a  
join (x,y) | x==y = x
```

Free Variables

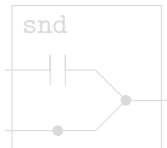
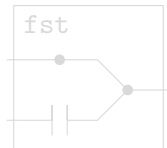
```
unknown :: () -> a  
unknown () = x where x free
```

Definition of fst and snd

```
fst :: (a,_) -> a  
fst (a,_) = a  
snd :: (_,b) -> b  
snd (_,b) = b
```

Unification

```
(==) :: (a,a) -> Success
```



Invert is a Jack-of-all-Trades

Definition of join

```
join :: (a,a) -> a  
join = invert fork
```

Free Variables

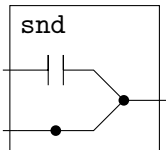
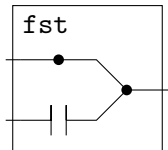
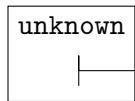
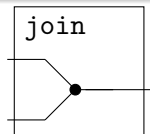
```
unknown :: () -> a  
unknown = invert unit
```

Definition of fst and snd

```
fst :: (a,_) -> a  
fst = (id / unit * unknown) * join  
snd :: (_,b) -> b  
snd = (unit * unknown / id) * join
```

Unification

```
(=:=) :: (a,a) -> Success
```



Matching on more than one Argument

```
(&&) :: Bool -> Bool -> Bool
True  && y = y
False && _ = False
```

```
(&&) :: (Bool,Bool) -> Bool
(&&) = invert (true / id) * snd
(&&) = invert (false / id) * fst * false
```

Rule combination

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

(?) has least precedence.

Don't throw away the Matching

```
snd :: (_,b) -> b
snd = (unit * unknown / id) * join

snd' :: (_,b) -> b
snd' = (unknown / id) * join
```

Matching on more than one Argument

```
(&&) :: Bool -> Bool -> Bool
True  && y = y
False && _ = False
```

```
(&&) :: (Bool,Bool) -> Bool
(&&) = invert (true / id) * snd
      ? invert (false / id) * fst * false
```

Rule combination

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

(?) has least precedence.

Don't throw away the Matching

```
snd :: (_,b) -> b
snd = (unit * unknown / id) * join

snd' :: (_,b) -> b
snd' = (unknown / id) * join
```

Matching on more than one Argument

```
(&&) :: Bool -> Bool -> Bool
True  && y = y
False && _ = False
```

```
(&&) :: (Bool,Bool) -> Bool
(&&) = invert (true / id) * snd'
      ? invert (false / id) * fst * false
```

Rule combination

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

(?) has least precedence.

Don't throw away the Matching

```
snd :: (_,b) -> b
snd = (unit * unknown / id) * join

snd' :: (_,b) -> b
snd' = (unknown / id) * join
```

Matching on more than one Argument

```
(&&) :: Bool -> Bool -> Bool
True  && y = y
False && _ = False
```

```
(&&) :: (Bool,Bool) -> Bool
(&&) = invert (true / id) * snd'
      ? invert (false / id) * fst * false
```

Rule combination

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

(?) has least precedence.

Don't throw away the Matching

```
snd :: (_,b) -> b
snd = (unit / id) * snd'

snd' :: (_,b) -> b
snd' = (unknown / id) * join
```

The General Method

Given Expression

$$\llbracket f \ e_1 \dots e_k \rrbracket = (\llbracket e_1 \rrbracket / \dots / \llbracket e_k \rrbracket) * f$$
$$\llbracket x \rrbracket = \text{id}$$

Given Function

$$f :: a_1 \rightarrow \dots \rightarrow a_n \rightarrow b$$
$$f \ p_{11} \ \dots \ p_{1n} = e_1$$
$$\vdots$$
$$f \ p_{m1} \ \dots \ p_{mn} = e_m$$

Adaption / Simplification

Transform tuple structures, e.g.,
 $(x, (y, z)) \mapsto ((x, y), (x, z))$
employing
id, fork, join, unit, unknown

The Result

$$f :: (\dots (a_1, a_2) \dots, a_n) \rightarrow b$$
$$f = \text{invert} (\llbracket p_{11} \rrbracket / \dots / \llbracket p_{1n} \rrbracket) * \{\leftarrow \text{adaption}_1 \rightarrow\} * \llbracket e_1 \rrbracket$$
$$\vdots$$
$$? \text{invert} (\llbracket p_{m1} \rrbracket / \dots / \llbracket p_{mn} \rrbracket) * \{\leftarrow \text{adaption}_m \rightarrow\} * \llbracket e_m \rrbracket$$

The General Method

Given Expression

$$\llbracket f \ e_1 \dots e_k \rrbracket = (\llbracket e_1 \rrbracket / \dots / \llbracket e_k \rrbracket) * f$$
$$\llbracket x \rrbracket = \text{id}$$

Given Function

$$f :: a_1 \rightarrow \dots \rightarrow a_n \rightarrow b$$
$$f \ p_{11} \ \dots \ p_{1n} = e_1$$
$$\vdots$$
$$f \ p_{m1} \ \dots \ p_{mn} = e_m$$

Adaption (+Simplification)

Transform tuple structures, e.g.,
 $(x, (y, z)) \mapsto ((x, y), (x, z))$
employing
id, fork, join, unit, unknown

The Result

$$f :: (\dots (a_1, a_2) \dots, a_n) \rightarrow b$$
$$f = \text{invert} (\llbracket p_{11} \rrbracket / \dots / \llbracket p_{1n} \rrbracket) * \{\leftarrow \text{adaption}_1 \rightarrow\} * \llbracket e_1 \rrbracket$$
$$\vdots$$
$$? \text{invert} (\llbracket p_{m1} \rrbracket / \dots / \llbracket p_{mn} \rrbracket) * \{\leftarrow \text{adaption}_m \rightarrow\} * \llbracket e_m \rrbracket$$

The General Method

Given Expression

$$\llbracket f \ e_1 \dots e_k \rrbracket = (\llbracket e_1 \rrbracket / \dots / \llbracket e_k \rrbracket) * f$$
$$\llbracket x \rrbracket = \text{id}$$

Given Function

$$f :: a_1 \rightarrow \dots \rightarrow a_n \rightarrow b$$
$$f \ p_{11} \ \dots \ p_{1n} = e_1$$
$$\vdots$$
$$f \ p_{m1} \ \dots \ p_{mn} = e_m$$

Adaption (+Simplification)

Transform tuple structures, e.g.,
 $(x, (y, z)) \mapsto ((x, y), (x, z))$
employing
id, fork, join, unit, unknown

The Result

$$f :: (\dots (a_1, a_2) \dots, a_n) \rightarrow b$$
$$f = \text{invert} (\llbracket p_{11} \rrbracket / \dots / \llbracket p_{1n} \rrbracket) * \{\leftarrow \text{adaption}_1 \rightarrow\} * \llbracket e_1 \rrbracket$$
$$\vdots$$
$$? \text{invert} (\llbracket p_{m1} \rrbracket / \dots / \llbracket p_{mn} \rrbracket) * \{\leftarrow \text{adaption}_m \rightarrow\} * \llbracket e_m \rrbracket$$

The General Method – Main Goals

Evaluating main goals

- transformed goals have *closed argument type*, i.e.,
 - $()$ is closed
 - if t_1 and t_2 are closed then also (t_1, t_2)
- for a given goal e , adapt $\llbracket e \rrbracket$ to $()$, i.e., fork to match closed type
- evaluate the result applied to $()$.

Main goal example

The expression $(\text{True} \ \&\& \ \text{False})$

- is transformed: $((\text{true} / \text{false}) * (\&\&)::((), ())) \rightarrow \text{Bool}$
- is adapted: $(\text{fork} * (\text{true} / \text{false}) * (\&\&)::() \rightarrow \text{Bool})$
- and applied: $(\text{fork} * (\text{true} / \text{false}) * (\&\&)) () \xrightarrow{*}$
 $(\&\&)((\text{true} / \text{false}) (\text{fork} ())) \Rightarrow$
 $(\&\&)((\text{true} / \text{false}) ((), ())) \Rightarrow$
 $(\&\&)(\text{true} (), \text{false} ()) \xrightarrow{*} (\&\&)(\text{True}, \text{False}) \xrightarrow{*} \text{False}$

The General Method – Main Goals

Evaluating main goals

- transformed goals have *closed argument type*, i.e.,
 - $()$ is closed
 - if t_1 and t_2 are closed then also (t_1, t_2)
- for a given goal e , adapt $\llbracket e \rrbracket$ to $()$, i.e., fork to match closed type
- evaluate the result applied to $()$.

Main goal example

The expression $(\text{True} \ \&\& \ \text{False})$

- is transformed: $((\text{true} / \text{false}) * (\&\&)::((), ())) \rightarrow \text{Bool}$
- is adapted: $(\text{fork} * (\text{true} / \text{false}) * (\&\&)::() \rightarrow \text{Bool})$
- and applied: $(\text{fork} * (\text{true} / \text{false}) * (\&\&)) () \xRightarrow{*}$
 $(\&\&)((\text{true} / \text{false}) (\text{fork} ())) \Rightarrow$
 $(\&\&)((\text{true} / \text{false}) ((), ())) \Rightarrow$
 $(\&\&)(\text{true} (), \text{false} ()) \xRightarrow{*} (\&\&)(\text{True}, \text{False}) \xRightarrow{*} \text{False}$

A standard example

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

The Unseen Servant

```
apply :: (a -> b, a) -> b  
apply (f,x) = f x
```

Recursion is: Recursion is Recursion

```
map = invert (id / nil) * fst' * nil  
    ? invert (id / cons) * adapt * (apply / map) * cons  
adapt (f,(x,xs)) = ((f,x),(f,xs))
```

The Partner of Apply

```
curry :: (() -> (a,b) -> c) -> () -> (a -> b -> c)
curry f () x y = f () (x,y)
```

Tupling, another useful abbreviation

```
(><) :: (a -> b) -> (a -> c) -> a -> (b,c)
f >< g = fork * (f / g)
```

An example mapping

```
curryAnd () x = (&&) x

listFalseTrue = (false >< (true >< nil) * cons) * cons
andTrue = (curry curryAnd >< true) * apply
main = (andTrue >< listFalseTrue) * map

main ()  $\xRightarrow{*}$  [False,True]
```

Denotation by Transformation: Relation Algebra

Converting Expressions

$\text{id} :: a \rightarrow a$

$(*) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c$

$(/) :: (a \rightarrow c, b \rightarrow d) \rightarrow (a, c) \rightarrow (b, d)$

Recursion is: Recursion is Recursion

Interface Adaption

$\text{fork} :: a \rightarrow (a, a)$

$\text{unit} :: a \rightarrow ()$

Pattern Matching

$\text{invert} :: (a \rightarrow b) \rightarrow (b \rightarrow a)$

$(?) :: a \rightarrow a \rightarrow a$

Relation Algebra

Identity I

Multiplication $*$

Parallel Comp. \parallel

Least Fixpoint

Relation Algebra

Tupling $[I, I]$

All-Relation L

Relation Algebra

Transposition \top

Union \cup

Transformation to point-free style:

- transform functional logic language to point-free style
- keep close to the original to make equational reasoning possible
 - keep the given recursive structure and existing data declarations
- logic features strongly reduce the set of point-wise primitives needed
 - seven primitives suffice for core language, two more for higher order

Denotation by Transformation:

- point-wise primitives direct correspond to relation algebraic constructs
- algebraic reasoning about programs in reach (*half*-automatic)

Future Work:

- short term: correctness of transformation
- middle term: device relation algebraic model to reflect laziness
- long term: many fruitful applications are waiting. . .