

Types and Topology

A workshop in honour of
Martín Escardó's 60th birthday
17–18 December 2025

Non-strictly positive induction
for breadth-first search

Ulrich Berger
Swansea University

Selection functions and quantifiers á la Martín Escardó

Type of selection functions of type x w.r.t. result type r :

```
type J r x = (x -> r) -> x
```

Product of selection functions:

```
bigotimes :: [J r x] -> J r [x]
```

```
bigotimes [] p = []
```

```
bigotimes (e : es) p = x0 : bigotimes es (p . (x0 :))  
  where x0 = e(\ x-> p(x : bigotimes es (p . (x :))))
```

Type of quantifiers, a.k.a. continuations:

```
type K r x = (x -> r) -> r
```

Turning a selection function into a quantifier:

```
overline :: J r x -> K r x
```

```
overline e = \ p-> p(e p)
```

M. Escardó, P. Oliva: *What Sequential Games, the Tychonoff Theorem and the Double-Negation Shift have in Common*. MFPS 2010.

Searching with monads

The function `bigotimes`, which uses the selection monad `J`, can be used to search for an 'optimal' path in an infinite binary tree.

By translating the selection monad to the continuation monad `K`, one can compute quantification over the (uncountable!) space of paths in an infinite tree.

We discuss another 'patently non self-explanatory' algorithm, due to Martin Hofmann, that uses the continuation monad for breadth-first search.

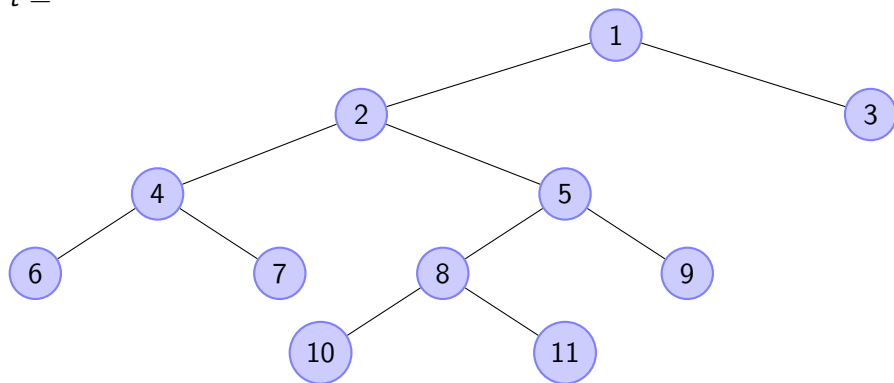
Martin Hofmann. Non strictly positive datatypes in system F, 15 Feb, 14:40:03 GMT 1993. Email on types mailing list:

<http://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00027.html>

Ralph Matthes, Anton Setzer, B: "Martin Hofmann's case for non-strictly positive data types", TYPES 2018 post-proceedings, LIPIcs 130).

Breadth-first search

$t =$



$\text{niv } t = [[1], [2, 3], [4, 5], [6, 7, 8, 9], [10, 11]]$

$\text{bfspec } t = \text{concat}(\text{niv } t) = [1, \dots, 11]$

Breadth-first search in Haskell

```
data Tree = Leaf Int | Node Tree Int Tree
```

```
bfspec :: Tree -> [Int]  
bfspec t = concat (niv t)
```

```
niv :: Tree -> [[Int]]  
niv (Leaf n) = [[n]]  
niv (Node t1 n t2) = [n] : zip (niv t1) (niv t2)
```

```
zip :: [[a]] -> [[a]] -> [[a]]  
zip [] l1 = l1  
zip l1 [] = l1  
zip (l:l1) (l':l1) = (l ++ l') : zip l1 l1
```

Martin Hofmann's breadth-first search algorithm

```
data Rou = Over | Next ((Rou -> [Int]) -> [Int])
```

```
bfMH :: Tree -> [Int]
```

```
bfMH t = extract(br t Over)
```

```
br :: Tree -> Rou -> Rou
```

```
br (Leaf n) c = Next (\ k -> n : unfold c k)
```

```
br (Node t0 n t1) c =
```

```
  Next(\ k -> n : unfold c (k . br t0 . br t1))
```

```
unfold :: Rou -> (Rou -> [Int]) -> [Int]
```

```
unfold Over k = k Over
```

```
unfold (Next f) k = f k
```

```
extract :: Rou -> [Int]
```

```
extract Over = []
```

```
extract (Next f) = f extract
```

Verification strategy

We extract the algorithm from a proof that $\text{bfMH } t$ yields the correct result for every finite tree t .

The extracted program will not use Haskell types or programs and the correctness does not rely on the correctness of the Haskell compiler.

Stating the correctness of Hofmann's algorithm

Variables: $n : \text{Int}$, $l : [\text{Int}]$, $t : \text{Tree}$.

$$\text{N}(n) \stackrel{\mu}{=} n = 0 \vee \text{N}(n - 1)$$

$$\text{List}(l) \stackrel{\mu}{=} l = [] \vee \exists n, l'. l = n : l' \wedge \text{N}(n) \wedge \text{List}(l')$$

$$\begin{aligned} \text{T}(t) \stackrel{\mu}{=} & (\exists n. t = \text{Leaf } n \wedge \text{N}(n)) \vee \\ & \exists t_0, t_1, n. t = \text{Node } t_0 \ n \ t_1 \wedge \text{T}(t_0) \wedge \text{N}(n) \wedge \text{T}(t_1) \end{aligned}$$

Theorem

$\forall t. \text{T}(t) \rightarrow \text{List}(\text{bfMH } t) \wedge \text{bfMH } t = \text{bfspec } t$.

i.e.

$\forall t. \text{T}(t) \rightarrow \text{List}(\text{extract}(\text{br } t \text{ Over})) \wedge$

$\text{extract}(\text{br } t \text{ Over}) = \text{concat}(\text{niv } t)$.

Formalization

The types `Int`, `[Int]`, `[[Int]]`, `Tree`, `Row` are treated as base types.

Functions are modelled as constants, specified by their defining equations.

Proof of $T(t) \rightarrow \text{List}(\text{extract}(\text{br } t \text{ Over})) \wedge \text{extract}(\text{br } t \text{ Over}) = \text{concat}(\text{niv } t)$

$\vec{l}: [[\text{Int}]], c : \text{Rou}, k : \text{Rou} \rightarrow [\text{Int}], f : (\text{Rou} \rightarrow [\text{Int}]) \rightarrow [\text{Int}],$
 $R : \text{Rou} \times [[\text{Int}]] \rightarrow \text{Prop}.$

$\text{isextractor}(R, \vec{l}, k) := \forall c, \vec{l}. R(c, \vec{l}) \rightarrow \text{List}(k \ c) \wedge k \ c = \text{concat}(\text{zip } \vec{l} \ \vec{l})$

$\text{rep}(c, \vec{l}_0) \stackrel{\mu}{=} (c = \text{Over} \wedge \vec{l}_0 = []) \vee$
 $\exists f, l, \vec{l}. c = \text{Next}(f) \wedge \vec{l}_0 = l : \vec{l} \wedge$
 $\forall k, \vec{l}'. \text{isextractor}(\text{rep}, \vec{l}', k) \rightarrow$
 $\text{List}(f \ k) \wedge f \ k = l ++ \text{concat}(\text{zip } \vec{l}' \ \vec{l}')$

Lemma 1 $\text{isextractor}(\text{rep}, [], \text{extract})$, i.e.

$\forall c, \vec{l}. \text{rep}(c, \vec{l}) \rightarrow \text{List}(\text{extract } c) \wedge \text{extract } c = \text{concat } \vec{l}$

Proof of Lemma 1: $R_0(c, \vec{l}) := \text{List}(\text{extract } c) \wedge \text{extract } c = \text{concat } \vec{l}$.

Our goal is $\text{rep} \subseteq R_0$ which can be proven by monotone induction.

Proof of $T(t) \rightarrow \text{List}(\text{extract}(\text{br } t \text{ Over})) \wedge \text{extract}(\text{br } t \text{ Over}) = \text{concat}(\text{niv } t)$

We already proved

Lemma 1. $\text{isextractor}(\text{rep}, [], \text{extract})$.

Lemma 2. $\forall t. T(t) \rightarrow \forall c, \vec{l}. \text{rep}(c, \vec{l}) \rightarrow \text{rep}(\text{br } t \text{ } c, \text{zip}(\text{niv } t) \vec{l})$.

Proof of Lemma 1: Induction on $T(t)$.

Finishing the proof of the theorem:

Assume $T(t)$.

Since $\text{rep}(\text{Over}, [])$, by Lemma 2, $\text{rep}(\text{br } t \text{ Over}, \text{niv } t)$.

Since $\text{isextractor}(\text{rep}, [], \text{extract})$, by Lemma 1, it follows, by the definition of isextractor , $\text{List}(\text{extract}(\text{br } t \text{ Over}))$ and $\text{extract}(\text{br } t \text{ Over}) = \text{concat}(\text{niv } t)$.

Types of predicates and lemmas

$$\begin{aligned}\mathcal{T}(\mathbf{N}) &= \mathbf{nat} := \mathbf{fix}(\lambda\alpha : *. \mathbf{1} \oplus \alpha) \\ \mathcal{T}(\mathbf{List}) &= \mathbf{list} := \mathbf{fix}(\lambda\alpha : *. \mathbf{1} \oplus \mathbf{nat} \otimes \alpha) \\ \mathcal{T}(\mathbf{T}) &= \mathbf{tree} := \mathbf{fix}(\lambda\alpha : *. \mathbf{nat} \oplus (\alpha \otimes \mathbf{nat} \otimes \alpha)) \\ \mathcal{T}(\mathbf{isextractor}) &= \lambda\alpha : *. \alpha \Rightarrow \mathbf{list} \\ \mathcal{T}(\mathbf{rep}) &= \mathbf{fix}(\lambda\alpha : *. \mathbf{1} \oplus (\mathcal{T}(\mathbf{isextractor}) \alpha \Rightarrow \mathbf{list})) \\ &= \mathbf{fix}(\lambda\alpha : *. \mathbf{1} \oplus ((\alpha \Rightarrow \mathbf{list}) \Rightarrow \mathbf{list})) \\ &\stackrel{\wedge}{=} \mathbf{Rou} \\ \mathcal{T}(\mathbf{Lemma1}) &= \mathcal{T}(\mathbf{isextractor}) \mathcal{T}(\mathbf{rep}) \\ &= \mathcal{T}(\mathbf{rep}) \Rightarrow \mathbf{list} \\ &\stackrel{\wedge}{=} \mathbf{Rou} \rightarrow [\mathbf{Int}] \\ \mathcal{T}(\mathbf{Lemma2}) &= \mathbf{tree} \Rightarrow \mathcal{T}(\mathbf{rep}) \Rightarrow \mathcal{T}(\mathbf{rep}) \\ &\stackrel{\wedge}{=} \mathbf{Tree} \rightarrow \mathbf{Rou} \rightarrow \mathbf{Rou} \\ \mathcal{T}(\mathbf{Theorem}) &= \mathbf{tree} \Rightarrow \mathbf{list} \\ &\stackrel{\wedge}{=} \mathbf{Tree} \rightarrow [\mathbf{Int}]\end{aligned}$$

Extracting Hofmann's algorithm

ep(Lemma1) : $\mathcal{T}(\text{rep}) \Rightarrow \mathbf{list}$

ep(Lemma1) $\stackrel{\wedge}{=} \text{extract}$

ep(Lemma2) : $\mathbf{tree} \Rightarrow \mathcal{T}(\text{rep}) \Rightarrow \mathcal{T}(\text{rep})$

ep(Lemma2) $\stackrel{\wedge}{=} \text{br}$

ep(Theorem) : $\mathbf{tree} \Rightarrow \mathbf{list}$

ep(Theorem) $\stackrel{\wedge}{=} \text{bfMH}$

Breadth-first search for infinite trees

Redefining the predicate T as greatest fixed point characterizes finite and infinite trees:

$$T(t) \stackrel{\nu}{=} (\exists n. t = \text{Leaf } n \wedge N(n)) \vee \\ \exists t_0, t_1, n. t = \text{Node } t_0 \ n \ t_1 \wedge T(t_0) \wedge N(n) \wedge T(t_1)$$

Finite nonempty lists of natural numbers:

$$NL(l) \stackrel{\mu}{=} \exists n, l_0. l = n : l_0 \wedge N(n) \wedge (l_0 = [] \vee NL(l_0))$$

Bisimulation for finite and infinite streams of natural numbers:

$$l \sim l' \stackrel{\nu}{=} l = l' = [] \vee \\ \exists l_0, h_1, l'_1. l = l_0 ++ h_1 \wedge l' = l_0 ++ l'_1 \wedge NL(l_0) \wedge h_1 \sim l'_1$$

Theorem

$\forall t. T(t) \rightarrow \text{extract}(\text{br } t \text{ Over}) \sim \text{concat}(\text{niv } t).$

Proving $\mathbb{T}(t) \rightarrow \text{extract}(\text{br } t \text{ Over}) \sim \text{concat}(\text{niv } t)$

Finite lists of infinite trees:

$$\text{LT}(\vec{t}) \stackrel{\mu}{=} \vec{t} = [] \vee \exists t, \vec{t}_0. \vec{t} = t : \vec{t}_0 \wedge \mathbb{T}(t) \wedge \text{LT}(\vec{t}_0)$$

```
brs :: [Tree] -> Rou -> Rou
```

```
brs = foldr br Over          -- brs [t] = br t Over
```

```
nivs :: [Tree] -> [[Int]]
```

```
nivs = foldr (zip . niv) []  -- nivs [t] = niv t
```

Suffices $\forall \vec{t}. \text{LT}(\vec{t}) \rightarrow \text{extract}(\text{brs } \vec{t}) \sim \text{concat}(\text{nivs } \vec{t})$.

Proof by coinduction:

The case $\vec{t} = []$ is easy. Hence, assume \vec{t} is nonempty:

$$\begin{aligned} & \text{extract}(\text{brs } \vec{t}) \\ &= \text{roots } \vec{t} ++ \text{extract}(\text{brs}(\text{subs } \vec{t})) \quad \text{ind. on } \text{LT}(\vec{t}) \\ &\sim \text{roots } \vec{t} ++ \text{concat}(\text{nivs}(\text{subs } \vec{t})) \quad \text{by coind. hyp.} \\ &= \text{concat}(\text{roots } \vec{t} : \text{nivs}(\text{subs } \vec{t})) = \text{concat}(\text{nivs } \vec{t}) \end{aligned}$$

Concluding remarks

Although the examples discussed in this talk cannot be directly carried out in Agda or Rocq, they can be interpreted in the theory of Scott domains, which is constructive and can be modelled in type theory (synthetic domain theory).

Monotone (but not necessarily strictly positive) induction and coinduction has a natural computational interpretation and may be considered as an addition to constructive type theory.

Question: Is there a proof whose extracted program is Hofmann's algorithm for infinite trees?