Using the Size-Change Principle for checking totality of recursive definitions

Pierre Hyvernat*

* Laboratoire de mathématiques, université Savoie Mont Blanc

chocola, Lyon, September 2016

Laboratoire de Mathématiques
Université Savoie Mont-Blanc

The “Size-Change Principle”

- Relevant instance of the SCP:

```
{ Head = x1::x2; Tail = { Head = x3 ; Tail = x4 } }
     =>
{  Head = Ω::<∞,-2>x4 ; Tail = <∞>x4  }, .<<-2>>
```

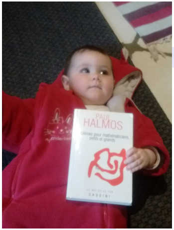
The "Size-Change Principle"

- Relevant instance of the SCP:

```
{ Head = x1::x2; Tail = { Head = x3 ; Tail = x4 } }
     =>
{  Head = Ω::<∞,-2>x4 ; Tail = <∞>x4  }, .<<-2>>
```

- Non relevant instance of the SCP:

=>



Plan

1. "size-change principle" and inductive types
2. "size-change principle" et productivity
3. "size-change principle" and totality

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
- call-by-value (?)

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
- call-by-value (?)
- arbitrary recursive definitions via equations (cf Haskell, Caml)

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
- call-by-value (?)
- arbitrary recursive definitions via equations (cf Haskell, Caml)
- termination checker to validate definitions

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
- call-by-value (?)
- arbitrary recursive definitions via equations (cf Haskell, Caml)
- termination checker to validate definitions

Termination checker: adaptation of the "size-change principle"
(Lee, Jones et Ben-Amram 2001, P.H. 2014)

Examples

```
val add m (n+1) = (add n m) + 1
  | add m 0 = m
```

Examples

```
val add m (n+1) = (add n m) + 1
  | add m 0 = m
val sum [] = 0
  | sum [n] = n
  | sum m::n::l = sum ((add m n)::l)
```

Examples

```
val add m (n+1) = (add n m) + 1
  | add m 0 = m
val sum [] = 0
  | sum [n] = n
  | sum m::n::l = sum ((add m n)::l)
```

Both functions terminate (on appropriate types)

- `add (m+1) (n+1)` ⇒ `add n (m+1)` ⇒ `add m n`: arguments decrease
- `sum _::(_::l)` ⇒ `sum ?::l`: tail of the argument decreases

Examples

```
val add m (n+1) = (add n m) + 1
  | add m 0 = m
val sum [] = 0
  | sum [n] = n
  | sum m::n::l = sum ((add m n)::l)
```

Both functions terminate (on appropriate types)

- `add (m+1) (n+1)` ⇒ `add n (m+1)` ⇒ `add m n`: arguments decrease
- `sum _::(_::l)` ⇒ `sum ?::l`: tail of the argument decreases

however

- `add m (n+1)` ⇒ `add n m`: no decrease with single call
- `sum n::m::l` ⇒ `sum ((add m n)::l)`: no decrease in whole argument

SCP: idea

Abstract interpretation of recursive call, keeping only

- first order arguments
- constants (constructors and structures)

SCP: idea

Abstract interpretation of recursive call, keeping only

- first order arguments
- constants (constructors and structures)

Example: for `add` et `sum`:

```
add m (n+1)  ⇒  add n m

sum n::m::l  ⇒  sum Ω::l
```

SCP: idea

Abstract interpretation of recursive call, keeping only

- first order arguments
- constants (constructors and structures)

Example: for add et sum:

```
add m (n+1)  ⇒  add n m

sum n::m::l  ⇒  sum Ω::l
```

We get in this way a call graph.

(vertices: mutually defined functions)

SCP: idea – 2

A bunch of mutually defined functions terminate if:

there are no infinite sequence of recursive calls to them.

SCP: idea – 2

A bunch of mutually defined functions terminate if:

> *there are no infinite sequence of recursive calls to them.*

The call graph contains cycles and thus, infinite path,

SCP: idea – 2

A bunch of mutually defined functions terminate if:

there are no infinite sequence of recursive calls to them.

The call graph contains cycles and thus, infinite path, but not all these path correspond to actual computations:

- the transition `f (A x)` $\Rightarrow$ `f (B x)` cannot be taken twice in a row (incompatibility),

SCP: idea – 2

A bunch of mutually defined functions terminate if:

there are no infinite sequence of recursive calls to them.

The call graph contains cycles and thus, infinite path, but not all these path correspond to actual computations:

- the transition `f (A x)` $\Rightarrow$ `f (B x)` cannot be taken twice in a row (incompatibility),
- the transition `f (B x)` $\Rightarrow$ `f x` cannot be taken infinitely many times in a row (decrease).

SCP: idea – 2

A bunch of mutually defined functions terminate if:

there are no infinite sequence of recursive calls to them.

The call graph contains cycles and thus, infinite path, but not all these path correspond to actual computations:

- the transition `f (A x)` $\Rightarrow$ `f (B x)` cannot be taken twice in a row (incompatibility),
- the transition `f (B x)` $\Rightarrow$ `f x` cannot be taken infinitely many times in a row (decrease).

"Size-change principle": sufficient condition for

no infinite path in the call graph corresponds to an actual computation path.

SCP: idea – 2

A bunch of mutually defined functions terminate if:

there are no infinite sequence of recursive calls to them.

The call graph contains cycles and thus, infinite path, but not all these path correspond to actual computations:

- the transition `f (A x)` $\Rightarrow$ `f (B x)` cannot be taken twice in a row (incompatibility),
- the transition `f (B x)` $\Rightarrow$ `f x` cannot be taken infinitely many times in a row (decrease).

"Size-change principle": sufficient condition for

no infinite path in the call graph corresponds to an actual computation path.

(all infinite path deconstruct an infinite branch in an argument)

SCP: more details

We compute a faithful approximation of the set of path:

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

with `g (S x)` $\Rightarrow$ `g x`:

- `g (S (S x))` $\Rightarrow$ `g x`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

with `g (S x)` $\Rightarrow$ `g x`:

- `g (S (S x))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g x`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

with `g (S x)` $\Rightarrow$ `g x`:

- `g (S (S x))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -1 \rangle$`x`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

with `g (S x)` $\Rightarrow$ `g x`:

- `g (S (S x))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -1 \rangle$`x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -2 \rangle$`x`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

with `g (S x)` $\Rightarrow$ `g x`:

- `g (S (S x))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -1 \rangle$`x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -2 \rangle$`x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -3 \rangle$`x`

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` $\Rightarrow$ `f (S x)`:

- `f x` $\Rightarrow$ `f (S (S x))`
- `f x` $\Rightarrow$ `f (S (S (S x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 1 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle 2 \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- `f x` $\Rightarrow$ `f (S (S (S` $\langle \infty \rangle$`x)))`
- ...

with `g (S x)` $\Rightarrow$ `g x`:

- `g (S (S x))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -1 \rangle$`x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -2 \rangle$`x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -3 \rangle$`x`
- `g (S (S (S x)))` $\Rightarrow$ `g` $\langle -3 \rangle$`x`
- ...

SCP: more details

We compute a faithful approximation of the set of path:

with `f x` ⇒ `f (S x)`:

- `f x ⇒ f (S (S x))`
- `f x ⇒ f (S (S (S x)))`
- `f x ⇒ f (S (S (S ⟨1⟩x)))`
- `f x ⇒ f (S (S (S ⟨2⟩x)))`
- `f x ⇒ f (S (S (S ⟨∞⟩x)))`
- `f x ⇒ f (S (S (S ⟨∞⟩x)))`
- ...

with `g (S x)` ⇒ `g x`:

- `g (S (S x)) ⇒ g x`
- `g (S (S (S x))) ⇒ g x`
- `g (S (S (S x))) ⇒ g ⟨−1⟩x`
- `g (S (S (S x))) ⇒ g ⟨−2⟩x`
- `g (S (S (S x))) ⇒ g ⟨−3⟩x`
- `g (S (S (S x))) ⇒ g ⟨−3⟩x`
- ...

Composition of path: unification + truncation

SCP: more details

We compute a faithful approximation of the set of path:

with `f x ⇒ f (S x)`:

- `f x ⇒ f (S (S x))`
- `f x ⇒ f (S (S (S x)))`
- `f x ⇒ f (S (S (S ⟨1⟩x)))`
- `f x ⇒ f (S (S (S ⟨2⟩x)))`
- `f x ⇒ f (S (S (S ⟨∞⟩x)))`
- `f x ⇒ f (S (S (S ⟨∞⟩x)))`
- ...

with `g (S x) ⇒ g x`:

- `g (S (S x)) ⇒ g x`
- `g (S (S (S x))) ⇒ g x`
- `g (S (S (S x))) ⇒ g ⟨−1⟩x`
- `g (S (S (S x))) ⇒ g ⟨−2⟩x`
- `g (S (S (S x))) ⇒ g ⟨−3⟩x`
- `g (S (S (S x))) ⇒ g ⟨−3⟩x`
- ...

Composition of path: unification + truncation with 2 parameters

- a depth $\geqslant 0$ for terms (here 3)
- a bound > 0 on coefficients (here 3)

SCP: details – 2

Note: `{ Fst = x ; Snd = y }` is approximated by $\langle 1 \rangle$`x` + $\langle 1 \rangle$`y`

(+ is commutative, associative and idempotent)

SCP: details – 2

Note: `{ Fst = x ; Snd = y }` is approximated by $\langle 1\rangle$`x` + $\langle 1\rangle$`y`

(+ is commutative, associative and idempotent)

Theorem (Ramsey, Lee, Jones, Ben-Amram, P.H.)

All infinite path in the call graph end with an infinity of loops "c" satisfying $c \supset cc$.

$\supset$: *equal up to approximating coefficients*

SCP: details – 2

Note: `{ Fst = x ; Snd = y }` is approximated by $\langle 1\rangle$`x` + $\langle 1\rangle$`y`

(+ is commutative, associative and idempotent)

Theorem (Ramsey, Lee, Jones, Ben-Amram, P.H.)

All infinite path in the call graph end with an infinity of loops "c" satisfying $c \supset cc$.

$\supset$: *equal up to approximating coefficients*

We just need to check that all the loops $c \supset cc$ have a decreasing argument.

SCP: details – 2

Note: `{ Fst = x ; Snd = y }` is approximated by $\langle 1\rangle$`x` + $\langle 1\rangle$`y`
(+ is commutative, associative and idempotent)

Theorem (Ramsey, Lee, Jones, Ben-Amram, P.H.)

All infinite path in the call graph end with an infinity of loops "c" satisfying $c \supset cc$.

$\supset$: equal up to approximating coefficients

We just need to check that all the loops $c \supset cc$ have a decreasing argument.

We get structural recursion in subterms, lexicographic combinations, argument permutations, locale size increase, ...

Plan

1. "size-change principle" and inductive types
2. "size-change principle" et productivity
3. "size-change principle" and totality

Example

```
val sums : stream(list(nat)) -> stream(nat)
```

Example

```
val sums : stream(list(nat)) -> stream(nat)
  | sums { Head=[]; Tail=s }  = { Head=0; Tail=sums s }
```

Example

```
val sums : stream(list(nat)) -> stream(nat)
  | sums { Head=[]; Tail=s }  = { Head=0; Tail=sums s }
  | sums { Head=[n]; Tail=s } = { Head=n; Tail=sums s }
```

Example

```
val sums : stream(list(nat)) -> stream(nat)
  | sums { Head=[]; Tail=s }  = { Head=0; Tail=sums s }
  | sums { Head=[n]; Tail=s } = { Head=n; Tail=sums s }
  | sums { Head=n::m::l; Tail=s } =
          sums { Head=(add n m)::l ; Tail=s }
```

Example

```
val sums : stream(list(nat)) -> stream(nat)
  | sums { Head=[]; Tail=s }  = { Head=0; Tail=sums s }
  | sums { Head=[n]; Tail=s } = { Head=n; Tail=sums s }
  | sums { Head=n::m::l; Tail=s } =
          sums { Head=(add n m)::l ; Tail=s }
```

- structures are lazy
- the third recursive call isn't guarded (Coquand 1993)
- but the definition is productive

SCP and productivity

In addition to arguments, we also keep track of the result.

(Altenkirch & Danielsson 2010, Raffalli & Hyvernat 2014)

SCP and productivity

In addition to arguments, we also keep track of the result.

(Altenkirch & Danielsson 2010, Raffalli & Hyvernat 2014)

```
(sums { Head=[]; Tail=s })      .Tail  ⇒ sums s
(sums { Head=[n]; Tail=s })     .Tail  ⇒ sums s
 sums { Head=n::m::l; Tail=s }         ⇒
                          sums { Head=Ω::l ; Tail=s }
```

SCP and productivity

In addition to arguments, we also keep track of the result.

(Altenkirch & Danielsson 2010, Raffalli & Hyvernat 2014)

```
(sums { Head=[]; Tail=s })      .Tail  ⇒ sums s
(sums { Head=[n]; Tail=s })     .Tail  ⇒ sums s
 sums { Head=n::m::l; Tail=s }         ⇒
                     sums { Head=Ω::l ; Tail=s }
```

A recursive definition is productive if for all infinite path:

- an 'inductive" branch in an argument is infinite (cf. previous slides),
- the "coinductive" branch of the result is infinite.

SCP and productivity

In addition to arguments, we also keep track of the result.

(Altenkirch & Danielsson 2010, Raffalli & Hyvernat 2014)

```
(sums { Head=[]; Tail=s })      .Tail  ⇒ sums s
(sums { Head=[n]; Tail=s })     .Tail  ⇒ sums s
 sums { Head=n::m::l; Tail=s }         ⇒
                          sums { Head=Ω::l ; Tail=s }
```

A recursive definition is productive if for all infinite path:

- an 'inductive" branch in an argument is infinite (cf. previous slides),
- the "coinductive" branch of the result is infinite.

The test is very similar, the coinductive branch of the result is seen as an additional argument.

Plan

1. "size-change principle" and inductive types
2. "size-change principle" et productivity
3. "size-change principle" and totality

(Counter) example

```
data tree where    --  (empty) inductive type
  | Node : stream(tree) -> tree
```

(Counter) example

```
data tree where   --  (empty) inductive type
  | Node : stream(tree) -> tree

val bad_s : stream(tree)
  | bad_s = { Head=Node bad_s ; Tail=bad_s }
```

(Counter) example

```
data tree where    --  (empty) inductive type
  | Node : stream(tree) -> tree

val bad_s : stream(tree)
  | bad_s = { Head=Node bad_s ; Tail=bad_s }
val bad_t : tree
  | bad_t = Node bad_s
```

(Counter) example

```
data tree where    --  (empty) inductive type
  | Node : stream(tree) -> tree

val bad_s : stream(tree)
  | bad_s = { Head=Node bad_s ; Tail=bad_s }
val bad_t : tree
  | bad_t = Node bad_s
```

- the definition is well-typed (Hindley-Milner)
- the definition is productive
- evaluation of `bad_t` (and all its subterms) terminates
- `bad_t` is not an element of the (empty) type `tree`

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
  - terminal coalgebras (coinductive types)
- call-by-value and lazy structures (?)

(cf `charity` by R. Cockett)

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
  - terminal coalgebras (coinductive types)
- call-by-value and lazy structures (?) (cf `charity` by R. Cockett)
- arbitrary recursive definitions via equations

Goal

- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
  - terminal coalgebras (coinductive types)
- call-by-value and lazy structures (?) (cf `charity` by R. Cockett)
- arbitrary recursive definitions via equations
- totality checker to validate definitions

Goal

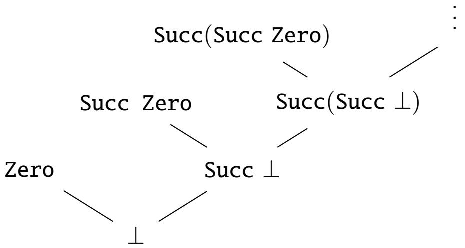
- typed functional language
- algebraic datatypes
  - sums (constructors)
  - products (structures)
  - initial algebras (inductive types)
  - terminal coalgebras (coinductive types)
- call-by-value and lazy structures (?) (cf `charity` by R. Cockett)
- arbitrary recursive definitions via equations
- totality checker to validate definitions

totality test: generalizes termination and productivity test
(SCP + "guard conditions" inspired by L. Santocanale's circular proofs)

Totality

Datatypes are interpreted by “lazy” domains.

Totality



Totality

Datatypes are interpreted by "lazy" domains.

`data` and `codata` are identical.

Totality

Datatypes are interpreted by "lazy" domains.

`data` and `codata` are identical.

Theorem

Every recursive definition induces a continuous function between the corresponding domains.

Totality

Datatypes are interpreted by "lazy" domains.

`data` and `codata` are identical.

Theorem

Every recursive definition induces a continuous function between the corresponding domains.

To distinguish inductive and coinductive types, we use the set theoretic interpretation (cf. Knaster Tarski theorem)

Definition

A (maximal) element of such a domain is <u>total</u> if it belongs to the corresponding set theoretic interpretation.

Totality

Datatypes are interpreted by "lazy" domains.

`data` and `codata` are identical.

Theorem

Every recursive definition induces a continuous function between the corresponding domains.

To distinguish inductive and coinductive types, we use the set theoretic interpretation (cf. Knaster Tarski theorem)

Definition

A (maximal) element of such a domain is total if it belongs to the corresponding set theoretic interpretation.

Goal: find a decidable totality criterion.

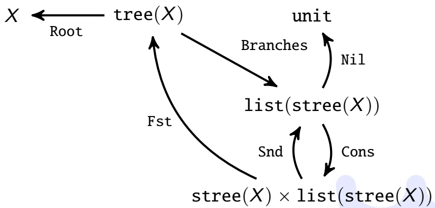
Parity games and totality

Coinductive "Rose trees":

```
codata stree('x) where
  | Root     : stree('x) -> 'x
  | Branches : stree('x) -> list(stree('x))
```

Parity games and totality

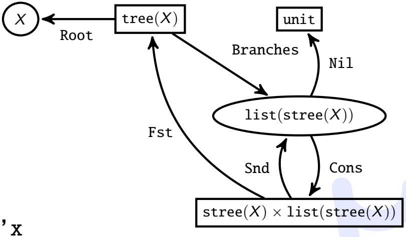
Coinductive "Rose trees":




```
codata stree('x) where
  | Root     : stree('x) -> 'x
  | Branches : stree('x) -> list(stree('x))
```

Parity games and totality

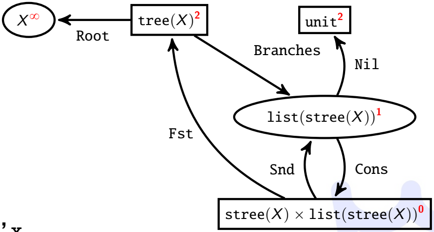
Coinductive "Rose trees":




```
codata stree('x) where
  | Root     : stree('x) -> 'x
  | Branches : stree('x) -> list(stree('x))
```

Parity games and totality

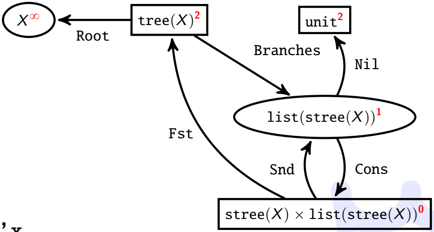
Coinductive "Rose trees":




```
codata stree('x) where
  | Root     : stree('x) -> 'x
  | Branches : stree('x) -> list(stree('x))
```

Parity games and totality

Coinductive "Rose trees":




```
codata stree('x) where
  | Root     : stree('x) -> 'x
  | Branches : stree('x) -> list(stree('x))
```

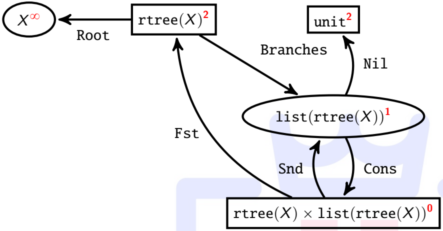
Theorem (L. Santocanale 2002)

Total elements of a type are exactly the winning strategies for the associated parity game.

Totality and strategies

Rules of the game:

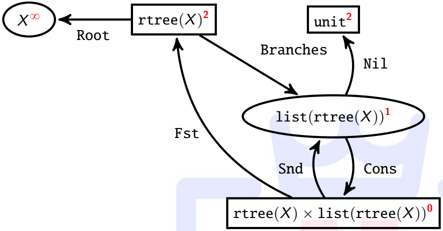
- I play on odd vertices



Totality and strategies

Rules of the game:

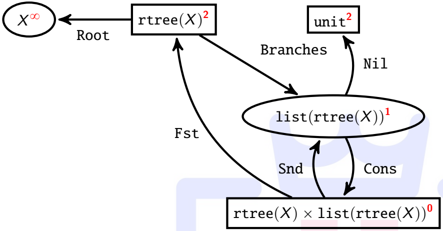
- I play on odd vertices
- I loose if I can't play



Totality and strategies

Rules of the game:

- I play on odd vertices
- I loose if I can't play
- if the play is infinite, I win if the maximum value that is visited infinitely often is even



SCP and strategies

- we keep track of the arguments and the result (like for productivity)

SCP and strategies

- we keep track of the arguments and the result (like for productivity)
- criterion: for all infinite path in the call graph,
  - either an argument contains an infinite branch where the maximal infinitely visited vertex is odd,
  - either the result contains an infinite branch where the maximal infinitely visited vertex is even

SCP and strategies

- we keep track of the arguments and the result (like for productivity)
- criterion: for all infinite path in the call graph,
  - either an argument contains an infinite branch where the maximal infinitely visited vertex is odd,
  - either the result contains an infinite branch where the maximal infinitely visited vertex is even

we need to keep a coefficient corresponding to the priority of a vertex during truncation:

$$\texttt{Cons}^{1}\ \{\ \texttt{Fst}^{2} = \texttt{Succ}^{1}\ \texttt{x}\ ;\ \texttt{Snd}^{2} = \texttt{y}\ \}$$

becomes

$$\langle 2^{1}, 1^{2} \rangle \texttt{x} + \langle 1^{1}, 1^{2} \rangle \texttt{y}$$

SCP and strategies

- we keep track of the arguments and the result (like for productivity)
- criterion: for all infinite path in the call graph,
  - either an argument contains an infinite branch where the maximal infinitely visited vertex is odd,
  - either the result contains an infinite branch where the maximal infinitely visited vertex is even

we need to keep a coefficient corresponding to the priority of a vertex during truncation:

$$\texttt{Cons}^{1}\ \{\ \texttt{Fst}^{2} = \texttt{Succ}^{1}\ \texttt{x}\ ;\ \texttt{Snd}^{2} = \texttt{y}\ \}$$

becomes

$$\langle 2^{1}, 1^{2} \rangle \texttt{x} + \langle 1^{1}, 1^{2} \rangle \texttt{y}$$

algorithm: SCP, yet again

What's missing

Some kind of definitions break the criterion:

- `val total (Fork ts) = sum (list_map total ts)`

  partially applied recursive function: the test always fails

What's missing

Some kind of definitions break the criterion:

- `val total (Fork ts) = sum (list_map total ts)`

  partially applied recursive function: the test always fails

  this can be solved by a smart static analysis (PML1)

- `val f (x::xs) = f (list_map (add 1) xs)`

  parameter under an application: unknown size (Ω)

What's missing

Some kind of definitions break the criterion:

- `val total (Fork ts) = sum (list_map total ts)`

  partially applied recursive function: the test always fails

  this can be solved by a smart static analysis (PML1)

- `val f (x::xs) = f (list_map (add 1) xs)`

  parameter under an application: unknown size (Ω)

  idea: complement the criterion with "sized types", as in Agda.