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

Pierre Hyvernat*

* Laboratoire de mathématiques, université Savoie Mont Blanc

chocola, Lyon, September 2016
The “Size-Change Principle”

Relevant instance of the SCP:

\[
\{ \text{Head} = x_1 : x_2 ; \text{Tail} = \{ \text{Head} = x_3 ; \text{Tail} = x_4 \} \} \\
\Rightarrow \\
\{ \text{Head} = \Omega : <\infty, -2>x_4 ; \text{Tail} = <\infty>x_4 \}, \langle<-2>\rangle
\]
The “Size-Change Principle”

Relevant instance of the SCP:
\[
\{ \text{Head} = x_1::x_2; \text{Tail} = \{ \text{Head} = x_3; \text{Tail} = x_4 \} \} \\
\Rightarrow \\
\{ \text{Head} = \Omega::<\infty,-2>x_4; \text{Tail} = <\infty>x_4 \}, <<2>>
\]

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

(termination checker: adaptation of the "size-change principle"
(P.H. 2014))
Goal

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

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

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

Termination checker: adaptation of the “size-change principle”

(Lee, Jones et Ben-Amram 2001, P.H. 2014)
Examples

\[
\begin{align*}
\text{val} & \quad \text{add m (n+1)} = (\text{add n m}) + 1 \\
| & \quad \text{add m 0} = m
\end{align*}
\]
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)

unicorn add (m+1) (n+1) ⇒ add n (m+1) ⇒ add m n: arguments decrease
llama 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:

\[
\begin{align*}
\text{add } m \ (n+1) & \Rightarrow \text{add } n \ m \\
\text{sum } n::m::l & \Rightarrow \text{sum } \Omega::l
\end{align*}
\]
SCP: idea

Abstract interpretation of recursive call, keeping only
- first order arguments
- constants (constructors and structures)

Example: for add et sum:

\[
\begin{align*}
\text{add } m \ (n+1) & \Rightarrow \text{add } n \ m \\
\text{sum } n::m::l & \Rightarrow \text{sum } \Omega::l
\end{align*}
\]

We get in this way a call graph.

(vertues: 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:

\textit{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

\textit{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:

\[
\text{with } f \ x \Rightarrow f \ (S \ x):
\]
SCP: more details

We compute a faithful approximation of the set of path:

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

Composition of path: unification + truncation with 2 parameters $a$ depth $\leq 0$ for terms (here 3) and $a$ bound $\leq 0$ on coefficients (here 3)
SCP: more details

We compute a faithful approximation of the set of path:

\[ \text{with } f \ x \Rightarrow f \ (S \ x) : \]
\[ - f \ x \Rightarrow f \ (S \ (S \ x)) \]
\[ - f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \]

Composition of path: unification + truncation
with 2 parameters
a depth \( \varepsilon \geq 0 \) for terms (here 3)
and a bound \( \alpha \geq 0 \) on coefficients (here 3)
SCP: more details

We compute a faithful approximation of the set of path:

\[ \text{with } f \ x \Rightarrow f \ (S \ x) : \]

- \( f \ x \Rightarrow f \ (S \ (S \ x)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \)
- \ldots \]

Composition of path: unification + truncation with 2 parameters:

- a depth \( \varepsilon \geq 0 \) for terms (here 3)
- a bound \( \alpha \leq 0 \) on coefficients (here 3)
SCP: more details

We compute a faithful approximation of the set of path:

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

Composition of path: unification + truncation with 2 parameters a depth $\epsilon \geq 0$ for terms (here 3) a bound $\lambda \geq 0$ on coefficients (here 3)
SCP: more details

We compute a faithful approximation of the set of path:

with \( f \ x \Rightarrow f \ (S \ x) \):
- \( f \ x \Rightarrow f \ (S \ (S \ x)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x))))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))) \)
- . . .
SCP: more details

We compute a faithful approximation of the set of path:

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

- $f \ x \Rightarrow f \ (S \ (S \ x))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ x)))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x)))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))))$
- ...
We compute a faithful approximation of the set of path:

with \( f \ x \Rightarrow f \ (S \ x) \):
- \( f \ x \Rightarrow f \ (S \ (S \ x)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x)))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))) \)
- ...
SCP: more details

We compute a faithful approximation of the set of path:

with \( f \ x \Rightarrow f \ (S \ x) \):
- \( f \ x \Rightarrow f \ (S \ (S \ x)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x)))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))) \)
- ... 

with \( g \ (S \ x) \Rightarrow g \ x \):
- \( g \ (S \ (S \ x)) \Rightarrow g \ x \)
- \( g \ (S \ (S \ (S \ x))) \Rightarrow g \ x \)

Composition of path: unification + truncation with 2 parameters a depth \( \delta \geq 0 \) for terms (here 3) a bound \( \alpha \geq 0 \) on coefficients (here 3)
SCP: more details

We compute a faithful approximation of the set of path:

with $f \ x \Rightarrow f \ (S \ x)$:
- $f \ x \Rightarrow f \ (S \ (S \ x))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ x)))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x)))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x)))$)
- ...
SCP: more details

We compute a faithful approximation of the set of path:

with \( f \ x \Rightarrow f \ (S \ x) \):
- \( f \ x \Rightarrow f \ (S \ (S \ x)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x()))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x))))) \)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x))))) \))) \)
- \( \ldots \)

with \( g \ (S \ x) \Rightarrow g \ x \):
- \( g \ (S \ (S \ x)) \Rightarrow g \ x \)
- \( g \ (S \ (S \ (S \ x))) \Rightarrow g \ x \)
- \( g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -1 \rangle x) \)
- \( g \ (S \ (S \ (S \ (S \ (S \ x)))) \) \Rightarrow g \ (\langle -2 \rangle x) \)
SCP: more details

We compute a faithful approximation of the set of path:

with $f \ x \Rightarrow f \ (S \ x)$:
- $f \ x \Rightarrow f \ (S \ (S \ x))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ x)))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x)))))$
- $f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))))$
- $\ldots$

with $g \ (S \ x) \Rightarrow g \ x$:
- $g \ (S \ (S \ x)) \Rightarrow g \ x$
- $g \ (S \ (S \ (S \ x))) \Rightarrow g \ x$
- $g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -1 \rangle x)$
- $g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -2 \rangle x)$
- $g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -3 \rangle x)$
- $\ldots$
SCP: more details

We compute a faithful approximation of the set of path:

with \( f \ x \Rightarrow f \ (S \ x) \):
- \( f \ x \Rightarrow f \ (S \ (S \ x)) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x)))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))) \)
- \( f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))) \)
- \( \ldots \)

with \( g \ (S \ x) \Rightarrow g \ x \):
- \( g \ (S \ (S \ x)) \Rightarrow g \ x \)
- \( g \ (S \ (S \ (S \ x))) \Rightarrow g \ x \)
- \( g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -1 \rangle x) \)
- \( g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -2 \rangle x) \)
- \( g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -3 \rangle x) \)
- \( g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -3 \rangle x) \)
- \( \ldots \)
SCP: more details

We compute a faithful approximation of the set of path:

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

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

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

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

Composition of path: unification + truncation
SCP: more details

We compute a faithful approximation of the set of path:

\[
\text{with } f \ x \Rightarrow f \ (S \ x) : \\
\quad - f \ x \Rightarrow f \ (S \ (S \ x)) \\
\quad - f \ x \Rightarrow f \ (S \ (S \ (S \ x))) \\
\quad - f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ x)))) \\
\quad - f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ x)))))) \\
\quad - f \ x \Rightarrow f \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))) \\
\quad - \ldots
\]

\[
\text{with } g \ (S \ x) \Rightarrow g \ x : \\
\quad - g \ (S \ (S \ x)) \Rightarrow g \ x \\
\quad - g \ (S \ (S \ (S \ x))) \Rightarrow g \ x \\
\quad - g \ (S \ (S \ (S \ (S \ x)))) \Rightarrow g \ (\langle -1 \rangle x) \\
\quad - g \ (S \ (S \ (S \ (S \ (S \ x)))))) \Rightarrow g \ (\langle -2 \rangle x) \\
\quad - g \ (S \ (S \ (S \ (S \ (S \ (S \ x))))))) \Rightarrow g \ (\langle -3 \rangle x) \\
\quad - \ldots
\]

Composition of path: unification + truncation with 2 parameters

- a depth $\geq 0$ 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)
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 \bowtie cc.

\bowtie: equal up to approximating coefficients
SCP: details – 2

Note: \{ \text{Fst} = x \; ; \; \text{Snd} = y \} \text{ is approximated by } \langle 1 \rangle x + \langle 1 \rangle y \\
(+ \text{ 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 \( c \bowtie cc \).

\( \bowtie \): equal up to approximating coefficients

We just need to check that all the loops \( c \bowtie cc \) 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 \( c \bowtie cc \).

\( \bowtie \): equal up to approximating coefficients

We just need to check that all the loops \( c \bowtie cc \) 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

```ocaml
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 }
Example

```plaintext
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

```plaintext
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)

\[
\text{(sums \{ Head=[\ ]; Tail=s \}) \quad \text{.Tail} \quad \Rightarrow \quad \text{sums s}}\\
\text{(sums \{ Head=[n]; Tail=s \}) \quad \text{.Tail} \quad \Rightarrow \quad \text{sums s}}\\
\text{sums \{ Head=n::m::l; Tail=s \} \quad \Rightarrow}\\
\text{sums \{ Head=\Omega::l \ ; Tail=s \}}
\]
SCP and productivity

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

(Altenkirch & Danielsson 2010, Raffalli & Hyvernat 2014)

\[
\begin{align*}
&\text{(sums } \{ \text{Head=}[]; \text{Tail}=s \}) \quad .\text{Tail} \quad \Rightarrow \quad \text{sums } s \\
&\text{(sums } \{ \text{Head=}[[n]; \text{Tail}=s \}) \quad .\text{Tail} \quad \Rightarrow \quad \text{sums } s \\
&\text{sums } \{ \text{Head=}n::m::l; \text{Tail}=s \} \quad \Rightarrow \\
&\quad \text{sums } \{ \text{Head=}\Omega::l; \text{Tail}=s \}
\end{align*}
\]

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)

\[(\text{sums } \{ \text{Head}=[\ ]; \text{Tail}=s \}) \quad \text{.Tail} \Rightarrow \text{sums } s\]
\[(\text{sums } \{ \text{Head}=[n]; \text{Tail}=s \}) \quad \text{.Tail} \Rightarrow \text{sums } s\]
\[\text{sums } \{ \text{Head}=n::m::l; \text{Tail}=s \} \quad \Rightarrow \]
\[\text{sums } \{ \text{Head}=\Omega::l ; \text{Tail}=s \} \]

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
```

Pierre Hyvernat
(Counter) example

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

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

The definition is well-typed (Hindley-Milner) and productive. Evaluation of `bad_t` (and all its subterms) terminates. `bad_t` is not an element of the (empty) type `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 \}
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 (?)
- arbitrary recursive definitions via equations

(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 (?)

arbitrary recursive definitions via equations

totality checker to validate definitions

(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 (?)
- 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

\[ \text{Succ(Succ Zero)} \]

\[ \text{Succ Zero} \]

\[ \text{Zero} \]

\[ \bot \]

\[ \text{Succ(Succ } \bot \text{)} \]

\[ \vdots \]
**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 **total** 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 lose if I can't play
Totality and strategies

Rules of the game:

- I play on odd vertices
- I lose 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:

\[
\text{Cons}^1 \{ \text{Fst}^2 = \text{Succ}^1 x ; \text{Snd}^2 = y \}
\]

becomes

\[
\langle 2^1, 1^2 \rangle x + \langle 1^1, 1^2 \rangle y
\]
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:

\[
\text{Cons}^1 \left\{ \text{Fst}^2 = \text{Succ}^1 x ; \text{Snd}^2 = y \right\}
\]

becomes

\[
\langle 2^1, 1^2 \rangle x + \langle 1^1, 1^2 \rangle y
\]

algorithm: SCP, yet again
What’s missing

Some kind of definitions break the criterion:

```haskell
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:

\[
\text{val } \text{total } (\text{Fork } ts) = \text{sum } (\text{list_map } \text{total } ts)
\]

partially applied recursive function: the test always fails

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

\[
\text{val } f (x::xs) = f (\text{list_map } (\text{add } 1) \ x s)
\]

parameter under an application: unknown size (Ω)
What’s missing

Some kind of definitions break the criterion:

\[\text{val total (Fork ts) = sum (list_map total ts)}\]

partially applied recursive function: the test always fails

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

\[\text{val f (x::xs) = f (list_map (add 1) xs)}\]

parameter under an application: unknown size (\(\Omega\))

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