Partiality and Co-Recursion

Ekaterina Komendantskaya

St Andrews

FP Lunch,
22 February 2010

Outline

1. Recursion and co-recursion in Coq (and other ITPs)
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Syntactic Approach to Termination: Structural Recursion and Guardedness

Outline

Outline

Workshop PAR at ITP'10

http://www.cs.st-andrews.ac.uk/ ek/PAR-10/

Important dates:

29 March 2010: Submission deadline
29 April 2010: Notification of acceptance
24 May 2010: Final version of accepted papers
15 July 2010: the workshop

Invited Speakers:

Conor McBride (University of Strathclyde)
tba

We plan EPTCS post-proceedings.

Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

- Choice of Type Theory: Type theory presents a powerful formal system that captures both the notion of *computation* (via the inclusion of functional programs written in typed λ-calculus), and *proof* (via the "formulas as types embedding", where types are viewed as propositions and terms as proofs).

Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

- Choice of Type Theory: Type theory presents a powerful formal system that captures both the notion of *computation* (via the inclusion of functional programs written in typed λ-calculus), and *proof* (via the "formulas as types embedding", where types are viewed as propositions and terms as proofs).
- Dependent products $\Longrightarrow$ additional expressiveness makes possible to consider propositions about programs/proofs; or to construct certified programs that satisfy a given property. (E.g. `prime_divisor`, `binary_world`,...)

Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

- Choice of Type Theory: Type theory presents a powerful formal system that captures both the notion of *computation* (via the inclusion of functional programs written in typed λ-calculus), and *proof* (via the "formulas as types embedding", where types are viewed as propositions and terms as proofs).
- Dependent products $\Longrightarrow$ additional expressiveness makes possible to consider propositions about programs/proofs; or to construct certified programs that satisfy a given property. (E.g. `prime_divisor`, `binary_world`,...)
- Proof assistant $=$ *proof checker* $+$ *proof-development system*. (*not theorem prover*)

The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

- Complexity: the problem is very big, the number of cases very large, etc.
  $\Longrightarrow$ computer assistance is needed.

The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

- Complexity: the problem is very big, the number of cases very large, etc.
  $\Longrightarrow$ computer assistance is needed.
- Depth: the problem is very deep, complicated, complex methods from different disciplines are needed (eg, Fermat theorem).
  $\Longrightarrow$ machine assistance for doing mathematical research.

The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

- Complexity: the problem is very big, the number of cases very large, etc.
  $\Longrightarrow$ computer assistance is needed.
- Depth: the problem is very deep, complicated, complex methods from different disciplines are needed (eg, Fermat theorem).
  $\Longrightarrow$ machine assistance for doing mathematical research.

The former is the reality, the latter is a challenge.

Type-theoretic approach to proof-checking

Decidability of type checking = core of the type-theoretic theorem proving

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{In situation } \Gamma \text{ we have A.}\\ \text{Proof. p.}\end{array}} \arrow[r, two heads] & \boxed{\Gamma \vdash_T \; p : A} \arrow[r, equal] & \boxed{\text{Type}_\Gamma(p) = A}
\end{tikzcd}$$

Type-theoretic approach to proof-checking

Decidability of type checking = core of the type-theoretic theorem proving

$$\begin{tikzcd} \boxed{\begin{array}{c}\text{In situation } \Gamma \text{ we have A.} \\ \text{Proof. p.}\end{array}} \arrow[r, two heads] & \boxed{\Gamma \vdash_T \; p : A} \arrow[r, equal] & \boxed{\text{Type}_\Gamma(p) = A} \end{tikzcd}$$

$\text{Type}_\_(-)$ is a function that finds for p a type in the given context Γ. The decidability of type-checking follows from:

- $\text{Type}_\Gamma(p)$ generates a type of p in context Γ or returns "false".
- The equality $=$ is decidable.

Three main decidability problems:

Three main decidability problems:

TCP $\Gamma \vdash_T M : A$?

“Does proof M indeed proves A?”

Three main decidability problems:

TCP $\Gamma \vdash_T M : A?$
“Does proof M indeed proves A?”

TSP $\Gamma \vdash_T M :?$
“Is proof M a proof?”

Three main decidability problems:

TCP $\Gamma \vdash_T M : A?$
"Does proof M indeed proves A?"

TSP $\Gamma \vdash_T M :?$
"Is proof M a proof?"

TIP $\Gamma \vdash_T ? : A?$
"Is A provable?"

Three main decidability problems:

TCP $\Gamma \vdash_T M : A?$
"Does proof M indeed proves A?"

TSP $\Gamma \vdash_T M :?$
"Is proof M a proof?"

TIP $\Gamma \vdash_T ? : A?$
"Is A provable?"

Both TCP and TSP are decidable.

provability of formula A	=	"inhabitation" of type A
proof checking	=	type checking
interactive theorem proving	=	i. construction of a term/given a type

Coq is the leading proof-assistant

Coq wins comparison with *Agda, Lego, Nurpl, HOL, Isabelle, Mizar, ACL2, PVS* in [Barendregt,Geuvers01].

Parameters:

Presence of Proof Objects: the script generates and stores a term that is isomorphic to a proof that can be checked on independent/simple proof checker. $\Longrightarrow$ high reliability. (!)

Reliability. (!)

Poincaré Principle There is a distinction between *computations* and *proofs*; computations do not require a proof. (E.g. $1+0 = 1$ does not require a proof.) The principle is useful to deal with Conversion Rule. (!)

Logic - Intuitionistic

Dependent Types (!)

Inductive Types (!)

Limits of Coq?

Marelle Team, INRIA, April 2008:

Majour:

- = limits of pure functional programming: no computational effects (side effects, interactive input/output, exceptions,..);
- Proof checker and not prover (2 researchers);
- Syntactic restrictions: difficult to have different views/representations of one object;
- Constructive logic (?);
- Too much of expressiveness: Coq Art.
- Structural recursion, Guardedeness...;

Limits of Coq?

Marelle Team, INRIA, April 2008:

Majour:

- = limits of pure functional programming: no computational effects (side effects, interactive input/output, exceptions,..);
- Proof checker and not prover (2 researchers);
- Syntactic restrictions: difficult to have different views/representations of one object;
- Constructive logic (?);
- Too much of expressiveness: Coq Art.
- Structural recursion, Guardedeness...;

Minor, technical hurdles:

- higher-order unification;
- deciding guardedness;
- need for a better organised documentation.

What is the one best thing about Coq?

Marelle Team, INRIA, April 2008:

- Mathematics and programming together; compute and prove simultaneously; $\Longrightarrow$ Research in Coq (3 researchers);
- Dependent types;
- Type theory $\Longrightarrow$ formal rigour;
- Implicit arguments, type inference.
- Extraction;
- Replication of proofs;
- Simple, uniform notation.

Successful applications of Coq (http://coq.inria.fr/)

Mathematics

- Geometry,
- Set Theory,
- Algebra,
- Number theory,
- Category Theory,
- Domain theory,
- Real analysis and Topology,
- Probabilities.

Successful applications of Coq (http://coq.inria.fr/)

Mathematics

- Geometry,
- Set Theory,
- Algebra,
- Number theory,
- Category Theory,
- Domain theory,
- Real analysis and Topology,
- Probabilities.

CS

- Infinite Structures,
- Pr. Lang.: Data Types and Data Structures;
- Pr. Lang.: Semantics and Compilation;
- Formal Languages Theory and Automata;
- Decision Procedures and Certified Algorithms;
- Concurrent Systems and Protocols;
- Operating Systems;
- Biology and Bio-CS.

Inductive Types and Recursive Functions

Coq = COC [Coquand,Huet'88] + CIC [Coquand,Paulin'93]

```
Inductive nat :  Set :=
| O : nat
| S : nat -> nat.
```

```
Fixpoint div2 n :  nat :=
 match n with
| O => 0
| S O => 0
| S (S n') => S (div2 n')
end.
```

Coinductive Types and Corecursive Functions

Coq = COC ['88] + CIC ['93] + CCC [Gimenez'96]

```
CoInductive str (A: Set) :  Set :=
SCons:  A -> str A -> str A.
```

```
CoFixpoint repeat (a:  A): str A :=
SCons a (repeat a).
```

```
CoInductive ETrees (A B: Set) :  Set :=
 A_node :  A -> ETrees A B -> ETrees A B
 B_node :  B -> ETrees A B -> ETrees A B -> ETrees A B.
```

But note that in Isabelle/HOL coiductive definitions are given through greatest fixed-points of monotone operators.

Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow$ types; proofs $\rightarrow$ programs): non-terminating proofs can lead to inconsistency.

Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow$ types; proofs $\rightarrow$ programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking of dependent types, we need to reduce expressions to normal form.

Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\longrightarrow$ types; proofs $\longrightarrow$ programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking of dependent types, we need to reduce expressions to normal form.

Example

The function `div2` is terminative.

Productive Values

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

Productive Values

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.
The element of the stream at position *n* can be found by:

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

A given stream s is productive if we can be sure that the computation of the list `nth n s` is guaranteed to terminate, whatever the value of `n` is.

Productive Values

Values in co-inductive types are productive when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

The element of the stream at position n can be found by:

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

A given stream s is productive if we can be sure that the computation of the list `nth n s` is guaranteed to terminate, whatever the value of `n` is.

Example

For any `n`, the value `repeat n` is productive.

Productivity for tree-like structures

```
Inductive direct : Type :=
L | R
```

fetch of the type `forall A B:Set, list direct -> ETrees A B -> A+B`

$$
\begin{cases}
\texttt{fetch nil (A\_node a t)} = \texttt{inl a} \\
\texttt{fetch (\_::tl) (A\_node a t)} = \texttt{fetch tl t} \\
\texttt{fetch nil (B\_node b t1 t2)} = \texttt{inr b} \\
\texttt{fetch (L::tl) (B\_node b t1 t2)} = \texttt{fetch tl t1} \\
\texttt{fetch (R::tl) (B\_node b t1 t2)} = \texttt{fetch tl t2}
\end{cases}
$$

Productive Functions

We call a function *productive at the input value i*, if it outputs a productive value at i.

Example

(Filter for streams). Filter is productive only on certain inputs. For a given predicate P,

$$\text{filter (SCons x tl)} = \begin{cases} \text{SCons x (filter tl)} & \text{if } P(x) \\ \text{filter tl} & \text{otherwise.} \end{cases}$$

A more general example

Definition

Let A, B be of type `Set`. For a predicate $P:\ B \longrightarrow$ `bool` and functions $h:\ B \longrightarrow A$, $g,\ g':\ B \longrightarrow B$, we define the function *dyn*:

$$\mathrm{dyn}\ (\mathrm{x})\ = \begin{cases} \mathrm{SCons}\ h(x)\ (\mathrm{dyn}\ (g(x))) & \text{if } P(x) \\ \mathrm{dyn}\ (g'(x)) & \text{otherwise.} \end{cases}$$

A more general example

Definition

Let A, B be of type `Set`. For a predicate $P: B \to$ `bool` and functions $h: B \to A$, $g, g': B \to B$, we define the function *dyn*:

$$\mathrm{dyn}\ (\mathrm{x}) = \begin{cases} \mathrm{SCons}\ h(x)\ (\mathrm{dyn}\ (g(x))) & \text{if } P(x) \\ \mathrm{dyn}\ (g'(x)) & \text{otherwise.} \end{cases}$$

Example

Suppose B is the set of natural numbers, $h = id$, $g = +1$; $g' = *2$; $P =$ "even". If we take $x = 1$, `dyn` will compute the infinite list:
2, 6, 14, 30, 62, 126, ...

A more general example

Definition

Let A, B be of type `Set`. For a predicate $P: B \to$ `bool` and functions $h: B \to A$, $g, g': B \to B$, we define the function *dyn*:

$$\mathrm{dyn}\ (\mathrm{x}) = \begin{cases} \mathrm{SCons}\ h(x)\ (\mathrm{dyn}\ (g(x))) & \text{if } P(x) \\ \mathrm{dyn}\ (g'(x)) & \text{otherwise.} \end{cases}$$

Example

Suppose B is the set of natural numbers, $h = id$, $g = +1$; $g' = *2$; $P =$ "even". If we take $x = 1$, `dyn` will compute the infinite list:
2, 6, 14, 30, 62, 126, ...
If B is a set of streams, we can have `dyn` = filter.

Totally-, Partially-, Non- Productive Functions

- Totally Productive
  (Function `repeat`)
- Partially Productive
  (Filters on streams and trees; `dyn`).
- Non-Productive
  Computing `nth 0 (filter even (repeat 1))` provokes the following computation:
  `filter even (repeat 1) repeat 1` ⇝ `filter even (1::repeat 1)` ⇝ `filter even (repeat 1)`...

Halting Problem

Before we start the survey of how recursion/co-recursion is formalised in Coq and other ITPs, it is good to remember that:

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

Design of new methods of implementation of recursion and co-recursion in ITPs is a small but ever-growing area. Of course, no new method is capable of giving us a tool that will allow precisely terminating/productive functions in the ITP, no less and no more. So, there is always a trade off between consistency/correctness and expressiveness.

Structural Recursion

Most of ITPs such as AGDA, Coq (but Isabelle/HOL also uses fixed point approach)

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

In this way we can be sure that the recursion terminates.

Structural Recursion

Most of ITPs such as AGDA, Coq (but Isabelle/HOL also uses fixed point approach)

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

In this way we can be sure that the recursion terminates.

Example

```
Fixpoint div2 n :  nat :=
match n with
| O => 0
| S O => 0
| S (S n') => S (div2 n')
 end.
```

Note also that there are additional termination checkers - `foetus` in AGDA; `Function` in Coq.

General Recursion

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument, are called *general recursive* arguments.

Example

$$\begin{cases} \log(\mathrm{S}\ 0) = 0 \\ \log(\mathrm{S}(\mathrm{S}\ n)) = \mathrm{S}(\log \mathrm{S}(\mathrm{div2}\ n)). \end{cases}$$

General Recursion

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument, are called *general recursive* arguments.

Example

$$\begin{cases} \log(\mathrm{S}\ 0) = 0 \\ \log(\mathrm{S}(\mathrm{S}\ n)) = \mathrm{S}(\log\ \mathrm{S}(\mathrm{div2}\ n)). \end{cases}$$

Syntactic and semantic methods to encode general recursion in ITPs

Synthactic methods

McBride: "Every total program is structurally recursive"

- Use of dummy arguments.
- Accessibility predicates [Bove];
- Lexicographic orders [Krauss, Nipkow];
- McKinna (M.Sozau's package);
- Ultrametric spaces and Banach's Fixed point theorem [Buchholz05, GianantonioMiculan03, Matthews99]

Syntactic and semantic methods to encode general recursion in ITPs

Synthactic methods

McBride: "Every total program is structurally recursive"

- Use of dummy arguments.
- Accessibility predicates [Bove];
- Lexicographic orders [Krauss, Nipkow];
- McKinna (M.Sozau's package);
- Ultrametric spaces and Banach's Fixed point theorem [Buchholz05, GianantonioMiculan03, Matthews99]

Semantic methods: Sized types

Could really bring "change". But encoding this approach in ITPs seems unlikely: this would require implementation of ordinal notation systems in type theory, which is non-trivial. Moreover, the closure ordinal ∞ of all inductive definitions should be inaccessible within the theory. [Hughes, Pareto, Sabry, 1996], [Barthe et al. , 2004], [Blanqui, 2005, 2005], (cost inference algorithm) [Hammond, Loidl, Vasconcelos 2002, 2004]

A very shallow account of type-based termination

General recursion can be implemented by adding a f-p combinator `fix` to a functional language based on λ-calculus.

$$\frac{f \in A \to A}{\mathit{fix}\ f \in A} \qquad \text{fix f} = \text{f (fix f)}$$

That means: "If f is an endofunctor on A, then $\mathit{fix}\ f$ inhabits A, and $\mathit{fix}\ f$ behaves as $f(\mathit{fix}\ f)$". General recursion makes language inconsistent as a logic, since every type is inhabited; and introduces non-termination if the equation is read as a computation rule. To maintain termination, *fix* should be restricted.

A very shallow account of type-based termination

General recursion can be implemented by adding a f-p combinator `fix` to a functional language based on λ-calculus.

$$\frac{f \in A \to A}{\mathit{fix}\ f \in A} \qquad\qquad \mathrm{fix\ f = f\ (fix\ f)}$$

That means: "If f is an endofunctor on A, then *fix* f inhabits A, and *fix* f behaves as $f(\mathit{fix}\ f)$". General recursion makes language inconsistent as a logic, since every type is inhabited; and introduces non-termination if the equation is read as a computation rule. To maintain termination, *fix* should be restricted.

The following typing rule for *fix* is admissible, that is, provable using just the typing rules of λ-calculus: $\dfrac{A^0 = T \quad f \in A_i \to A^{i+1}, \text{for all } i}{\mathit{fix}\ f \in A^n}$

If $\bigcap_{n \in N} A^n$ is "interesting", then *fix* f has interesting properties, like termination and productivity.

"Well-founded recursion": where does the notion come from?

Let $\prec$ be a binary relation on a set A. The well-founded part of $\prec$ is the set $W(\prec)$ of $a \in A$ such that there is no infinite descending sequence $a \succ a_0 \succ a_1 \ldots$. The relation $\prec$ is a well-founded relation if $A = W(\prec)$. $W(\prec)$ can be inductively defined as follows. Let $\Phi_\prec$ be the set of rules $(\prec a) \rightarrow a$ for $a \in A$, where $(\prec a) = \{x \in A | x \prec a\}$. See Aczel "An introduction to inductive definitions", 1977.
Already in early 70s [e.g. Manna 74], the well-founded induction (or also *Noetherian induction*) was used as the fundamental method for proving termination.
[Paulson 1984] lists a number of methods, e.g., the *ordering on natural* numbers, a *subrelation* of a w.f. relation, the *inverse image* of w.f. relation, the *transitive closure* of a w.f. relation, the *disjoint sum* over two w.f. relations; the *lexocographic order* of two w.f. relations; the *lexicographic power* of a w.f. relation; the *immediate subtree relation* of a wellordering type.

Accessibility and well-founded induction [Aczel77, Huet88]

The method of recursion over an inductive predicate is provided by *well-founded induction* that relies on an inductive notion of accessibility or adjoint.

```
Inductive Acc (A: Set) (R: A -> A -> Prop):  A -> Prop
:=
Acc_intro:  forall x:A, (forall y:  A, R x y -> Acc R y) ->
Acc R x.
```

If R is an arbitrary relation, we say that a sequence $a_i, (i \in N)$ is R-decreasing if R$a_i a_{i+1}$ holds for every i. Taking Φ to be the predicate "does not belong to an infinite decreasing chain", the following holds:

$$\forall x (\forall y . Ryx \rightarrow \Phi x) \leftrightarrow \Phi x$$

The accessibility predicate gives a good constructive description of the elements that do not belong to infinite decreasing chains. We can use this to express that function computations do not involve infinite sequences of recursive calls.

Accessibility and well-founded induction [Aczel77, Huet88], ctd

On the technical side, when h_x is a proof that x is accessible, and there is a proof h_r of type Ryx, we can easily build a proof (by pattern-matching) that y is accessible. This new proof is structurally smaller than h_x; so it can be used as an argument in a recursive call for a function whose principal argument is h_x. See theorem `Acc_inv` given in the Coq library:

```
Lemma Acc_inv :  forall x:A, Acc x -> forall y:A, R y x
-> Acc y.
```

Well-founded recursive functions

In the Coq library, there are defined "wellordering types", and then one uses the fact that *Wellfounded relations are the inverse image of wellordering types*.
The well-founded relation in Coq library:

```
Definition well_founded := forall a:A, Acc a.
```

When a relation is well-founded, we can define recursive functions where the relation is used to control which recursive calls are correct:

```
Definition well_founded_induction (A : Set) (R: A -> A
-> Prop) (H: well_founded R) (P: A -> Set) (f:  forall
x:A, R y x -> P y) -> P x) (x:A): P x :=
Acc_iter P f (H x).
```

Method of Ad-hoc Predicates [Aczel77], [Bove02]

Bove's contribution in particular was to use the method as a way of defining functions, (as opposed to using it separately for "proving" termination)

```
Inductive log_domain :  nat -> Prop :=
| log_domain_1 :  log_domain 1
| log_domain_2 :
 forall p:  nat, log_domain (S (div2 p)) -> log_domain (S (S
p)).
```

```
Lemma log_domain_inv :
forall x p :  nat, log_domain x -> x = S(S p) ->
log_domain (S (div2 p)).
```

```
Lemma log_domain_non_0:  forall x :nat,
log_domain x -> x ≠ 0.
```

Method of Ad-hoc Predicates [Aczel77], [Bove02].

```
Fixpoint log (x:nat)(h:log_domain x){struct h} :  nat
:=
match x as y return x = y -> nat with
| 0 => fun h' => False_rec nat (log_domain_non_0 x h h')
| S 0 => fun h' => 0
| S (S p) =>
fun h' => S (log (S (div2 p)) (log_domain_inv x p h h'))
end (refl_equal x).
```

This idea is now implemented in Isabelle/HOL, AGDA, Coq

Method of Ad-hoc Predicates [Aczel77], [Bove02].

```
Fixpoint log (x:nat)(h:log_domain x){struct h} :  nat
:=
match x as y return x = y -> nat with
| 0 => fun h' => False_rec nat (log_domain_non_0 x h h')
| S 0 => fun h' => 0
| S (S p) =>
fun h' => S (log (S (div2 p)) (log_domain_inv x p h h'))
end (refl_equal x).
```

This idea is now implemented in Isabelle/HOL, AGDA, Coq

Briefly about size-change principle of termination

Lee, Jones, Ben-Amram 2001

A program terminates on all inputs if every infinite call sequence would cause an infinite descent in soma data values. Note its relation to the well-founded recursion. Generally, checking the SCT is PSPACE complete, but with restrictions becomes polynomial.

SCT abstracts from the actual program by viewing it as a set of *control points* and transitions between them, forming a directed graph (the *control graph*). Each control point has a finite set of abstract *data positions* associated to it. Each transition is labeled by a *size-change graph*, which contains information about how the values in the data positions are related: e.g., decreases ($p \downarrow q$, if $q < p$) or remains equal ($p \updownarrow q$ if $q \leq p$ after transition. By connecting the size-change graphs along a control flow path, the data flow becomes visible. Chains of such connected edges are called *threads*. A thread has infinite descent if it contains infinitely many $\downarrow$ edges.

Briefly about size-change principle of termination, ctd

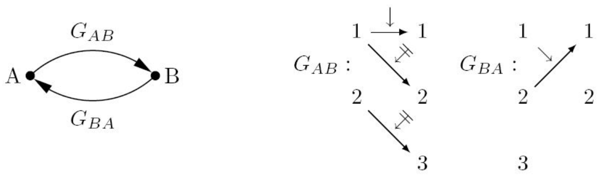
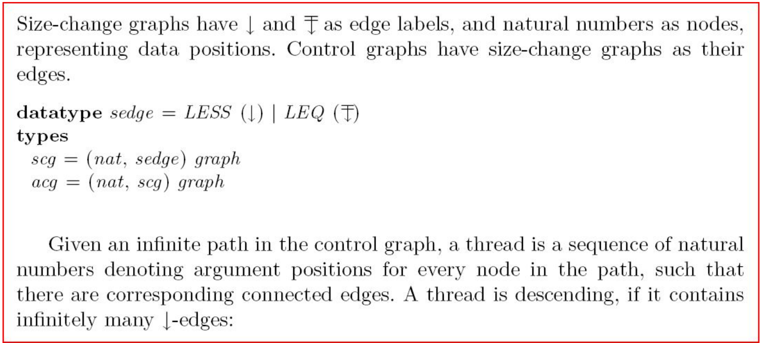


Fig. 1. A simple size-change problem


Size-change graphs have $\downarrow$ and $\overline{\downarrow}$ as edge labels, and natural numbers as nodes, representing data positions. Control graphs have size-change graphs as their edges.

datatype *sedge* = *LESS* ($\downarrow$) | *LEQ* ($\overline{\downarrow}$)
types
  scg = (*nat*, *sedge*) *graph*
  acg = (*nat*, *scg*) *graph*

Given an infinite path in the control graph, a thread is a sequence of natural numbers denoting argument positions for every node in the path, such that there are corresponding connected edges. A thread is descending, if it contains infinitely many $\downarrow$-edges:

Use of call descriptors, measure functions

A call descriptor is a triple (Γ, r, l), which describes a recursive call in a function definition:

r is the argument of the recursive call, l is the original argument (from the left hand side of the equation) and Γ is the condition under which the call occurs. All three values depend on variables (the pattern variables), which we replace by a single variable (possibly containing a tuple).

`types`

$(\alpha, \gamma) \mathit{cdesc} = (\gamma \rightarrow \mathit{bool}) \times (\gamma \rightarrow \alpha) \times (\gamma \rightarrow \alpha)$

Here, α is the argument type of the function and γ is the type of the pattern variable.

Measure functions capture the notion of size.

`types`

$\alpha \mathit{measure} = \alpha \rightarrow \mathit{nat}$

Isabelle provides a structural measure function for each inductive type.

Briefly about size-change principle of termination, ctd

It has been implemented in several ITPs:

- Agda: tool `foetus` detects lexicographic termination orderings for simply-typed functional programs and inductive types. [Abel, Altenkirch 2002]
- HOL/Isabelle: [Krauss 2007] Also related methods of lexicographic orders...
- Coq: McKinna, Sozeau - ?

Guardedness [Gimenez96: Calculus of Coinductive Constructions]

The guardedness condition insures that

- * each corecursive call is made under at least one constructor;
- ** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Guardedness [Gimenez96: Calculus of Coinductive Constructions]

The guardedness condition insures that

* each corecursive call is made under at least one constructor;

** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Example

```
CoFixpoint repeat (a:A): str A := SCons a (repeat a).
```

Non-guarded functions:

[$*$] is not satisfied:
Filters, `dyn`;
[$**$] is not satisfied:
Consider the following function computing lists of ordered natural numbers:
`nats = (SCons 1 (map (+ 1) nats)).`
where the function `map` above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```

Non-guarded functions:

[∗] is not satisfied:
Filters, `dyn`;
[∗∗] is not satisfied:
Consider the following function computing lists of ordered natural numbers:

```
nats = (SCons 1 (map (+ 1) nats)).
```

where the function `map` above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```

Non-guarded functions:

[$*$] is not satisfied:

Filters, `dyn`;

[$**$] is not satisfied:

Consider the following function computing lists of ordered natural numbers:

```
nats = (SCons 1 (map (+ 1) nats)).
```

where the function `map` above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```

Non-guarded functions:

[∗] is not satisfied:
Filters, `dyn`;
[∗∗] is not satisfied:
Consider the following function computing lists of ordered natural numbers:

```
nats = (SCons 1 (map (+ 1) nats)).
```

where the function `map` above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```

Other Productive Non-Guarded Functions

What other productive non-guarded functions do we know?

Other Productive Non-Guarded Functions

What other productive non-guarded functions do we know?

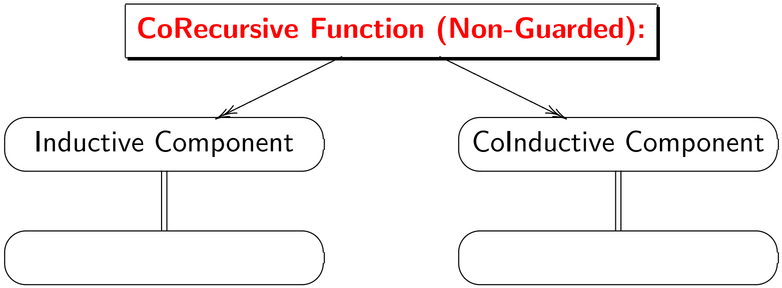
Every terminative function gives rise to a non-guarded totally productive function.

Example

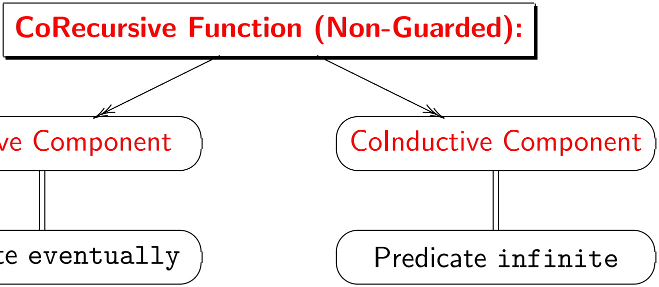
Terminative function $x - 1$ gives rise to the totally productive function `f: stream nat -> stream nat`:

$$\texttt{f (x::y::tl)} = \begin{cases} \texttt{x::f(y::tl)} & \text{if } x \leq y \\ \texttt{f((x-1)::y::tl)} & \text{otherwise.} \end{cases}$$

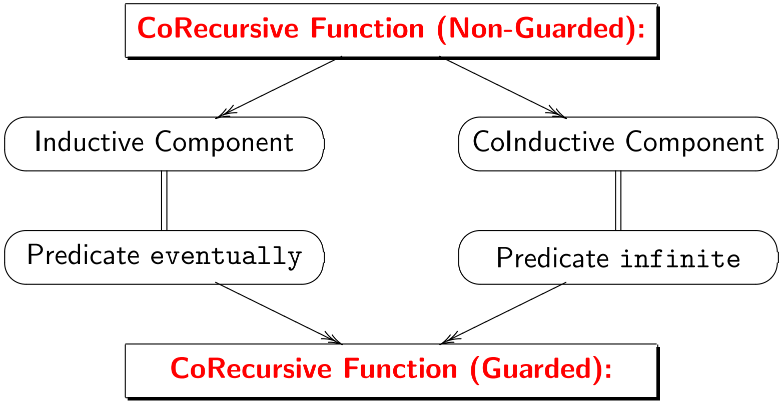
Our Method: Inductive and Coinductive Components



Our Method: Inductive and Coinductive Components



Our Method: Inductive and Coinductive Components



Predicate eventually: First-Step Productivity

Using eventually, we can describe the inductive component of a corecursive function. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call.

```
Inductive eventually_s:  str A -> Prop :=
| ev_b:  forall x s, P x -> eventually_s (SCons A x s)
| ev_r:  forall x s, not P x
-> eventually_s s -> eventually_s (SCons A x s).
```

Predicate eventually: First-Step Productivity

Using eventually, we can describe the inductive component of a corecursive function. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call.

```
Inductive eventually_s:  str A -> Prop :=
| ev_b:  forall x s, P x -> eventually_s (SCons A x s)
| ev_r:  forall x s, not P x
-> eventually_s s -> eventually_s (SCons A x s).
```

Predicate `eventually`: First-Step Productivity

Using `eventually`, we can describe the inductive component of a corecursive function. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call.

```
Inductive eventually_s:  str A -> Prop :=
| ev_b:  forall x s, P x -> eventually_s (SCons A x s)
| ev_r:  forall x s, not P x
-> eventually_s s -> eventually_s (SCons A x s).
```

eventually for dyn

```
Inductive eventually_dyn (x:  B) : Prop :=
| ev_dyn1 :  P x = true -> eventually_dyn x
| ev_dyn2 :  P x = false -> eventually_dyn (g' x) ->
eventually_dyn x.
```

Compare eventually with $\Diamond$ in [Pnueli81, Jacobs02].

Inversion Lemmas

Lemma eventually_s_inv:

```
forall (s :  str A),
eventually_s s -> forall x s', s = SCons A x s' ->
not P x -> eventually_s s'.
```

Inversion Lemmas

Lemma eventually_s_inv:

```
forall (s :  str A),
eventually_s s -> forall x s', s = SCons A x s' ->
not P x -> eventually_s s'.
```

Lemma eventually_dyn_inv :

```
forall x, eventually_dyn x -> P x = false ->
eventually_dyn (g' x).
```

Inductive Component of Filter

```
Fixpoint pre_filter_s (s : str A) (h : eventually_s s)
struct h : A * str A :=
match s as b return s = b -> A*str A with
SCons x s' =>
fun heq =>
match P_dec x with
| left _ => (x, s')
| right hn =>
pre_filter_s s' (eventually_s_inv s h x s' heq hn)
end
end (refl_equal s).
```

Inductive Component of Filter

```
Fixpoint pre_filter_s (s :  str A) (h :  eventually_s s)
struct h :  A * str A :=
match s as b return s = b -> A*str A with
SCons x s' =>
fun heq =>
match P_dec x with
| left _ => (x, s')
| right hn =>
pre_filter_s s' (eventually_s_inv s h x s' heq hn)
end
end (refl_equal s).
```

Inductive Component of Filter

```
Fixpoint pre_filter_s (s : str A) (h : eventually_s s)
struct h : A * str A :=
match s as b return s = b -> A*str A with
SCons x s' =>
fun heq =>
match P_dec x with
| left _ => (x, s')
| right hn =>
pre_filter_s s' (eventually_s_inv s h x s' heq hn)
end
end (refl_equal s).
```

Inductive Component of dyn

```
Fixpoint pre_dyn(x:B)(d:eventually_dyn x){struct d}:
A*B:=
 match P x as b return P x = b -> A*B with
| true => fun t => (h x, g x)
| false => fun t =>
pre_dyn (g' x) (eventually_dyn_inv x d t)
end (refl_equal (P x)).
```

Inductive Component of dyn

```
Fixpoint pre_dyn(x:B)(d:eventually_dyn x){struct d}:
A*B:=
 match P x as b return P x = b -> A*B with
| true => fun t => (h x, g x)
| false => fun t =>
pre_dyn (g' x) (eventually_dyn_inv x d t)
end (refl_equal (P x)).
```

Inductive Component of dyn

```
Fixpoint pre_dyn(x:B)(d:eventually_dyn x){struct d}:
A*B:=
 match P x as b return P x = b -> A*B with
| true => fun t => (h x, g x)
| false => fun t =>
pre_dyn (g' x) (eventually_dyn_inv x d t)
end (refl_equal (P x)).
```

Coinductive Predicate `infinite`

Corecursive computations are introduced by repeating computations performed by the inductive component. This can happen only if recursive calls satisfy the `eventually` predicate repeatedly. We need the predicate `infinite` to express this.

```
CoInductive infinite_s :  str -> Prop :=
al_cons: forall (s:str A) (h: eventually s),
 infinite_s (snd(pre_filter_s s h)) ->  infinite_s s.
```

The same predicate for dyn

```
CoInductive infinite_dyn (x :  B): Prop :=
  di : forall (d: eventually_dyn x),
   infinite_dyn (snd (pre_dyn x d)) -> infinite_dyn x.
```

The `infinite` predicate describes exactly those arguments to the function for which the function is guaranteed to be productive.

Relating eventually and infinite

```
Lemma infinite_eventually_dyn :
forall x, infinite_dyn x -> eventually_dyn x.
```

```
Lemma infinite_always_dyn :
forall x, infinite_dyn x ->
forall e:  eventually_dyn x,
infinite_dyn (snd (pre_dyn x e)).
```

Guarded Representation of a filter

```
CoFixpoint filter (s :  str A) : forall (h:  infinite_s
s), str A :=
match s return infinite_s s -> str A with
| SCons x s' =>
fun h' :  infinite_s (SCons A x s') =>
SCons A (fst
(pre_filter_s _ infinite_eventually_s (SCons A x s') h')))
(filter _ (infinite_always_s (SCons A x s') h'))
end.
```

Guarded Representation of a filter

```
CoFixpoint filter (s : str A) : forall (h: infinite_s
s), str A :=
match s return infinite_s s -> str A with
| SCons x s' =>
fun h' : infinite_s (SCons A x s') =>
SCons A (fst
(pre_filter_s _ infinite_eventually_s (SCons A x s') h'))
(filter _ (infinite_always_s (SCons A x s') h'))
end.
```

Guarded Representation of dyn

```
CoFixpoint dyn (x:B)(h:infinite_dyn x) :  str :=
SCons (fst (pre_dyn x (infinite_eventually_dyn ev x h)))
(dyn _ (infinite_always_dyn x h
(infinite_eventually_dyn x h))).
```

Guarded Representation of dyn

```
CoFixpoint dyn (x:B)(h:infinite_dyn x) :  str :=
SCons (fst (pre_dyn x (infinite_eventually_dyn ev x h)))
(dyn _ (infinite_always_dyn x h
(infinite_eventually_dyn x h))).
```

Recursive Equation Lemma for dyn

Theorem dyn_equation :

```
forall x i: infinite_dyn x, bisimilar_s (dyn x i)
(match Px as b return Px = b -> infinite_dyn x -> str
with
|true => fun t i =>
SCons(h x)(dyn(g x)(dyn_step1 x t i))
|false => fun t i => dyn (g' x) (dyn_step2 x t i)
end (refl_equal (P x)) i).
```

More Complicated Applications of Our Method:

Expression trees and dynamic filtering on expression trees.

The dynamic filter function on expression trees was used to establish a normalisation algorithm for an admissible representation of a closed interval of real numbers in [Geuvers1993], [Niqui2004].

More Complicated Applications of Our Method:

Expression trees and dynamic filtering on expression trees.

The dynamic filter function on expression trees was used to establish a normalisation algorithm for an admissible representation of a closed interval of real numbers in [Geuvers1993], [Niqui2004].
The function was not guarded.

We applied our method to give a Coq formalisation of the function.

Conclusions

1. We generalised the method of [Bertot05] to a wider class of functions:
    - various output data types including streams, expression and binary trees;
    - included dynamically changing functions;

   and thereby gave a general analysis of the method.
2. We work with partial productivity, and not just total productivity.
3. We establish the uniform Recursive Equation Lemmas for the functions we formalise, this was not achieved in [Bertot05].

Conclusions

1. We generalised the method of [Bertot05] to a wider class of functions:
   - various output data types including streams, expression and binary trees;
   - included dynamically changing functions;

   and thereby gave a general analysis of the method.
2. We work with partial productivity, and not just total productivity.
3. We establish the uniform Recursive Equation Lemmas for the functions we formalise, this was not achieved in [Bertot05].

Future work $\longrightarrow$ further automatisation.

Thank you!