Termination Checking in the Presence of Nested Inductive and Coinductive Types

Thorsten Altenkirch
(joint work with Nils Anders Danielsson)

School of Computer Science
University of Nottingham

July 16, 2010
Dependently typed programming, e.g. Agda, Epigram, Coq, ... 

Totality?
- Soundness as a logic
- Efficient code (don’t run proofs)

Two approaches:
1. Reduce to a total core language
   Epigram?, Coq?
2. Use a partial language and a termination checker
   Agda, Coq?
This talk

- Adding coinductive types to Agda
- Mixed inductive-coinductive definitions
- Simple but powerful extension of the termination checker (due to Andreas Abel).
- Easy to define inductive types nested inside coinductive types ($\nu\mu$).
- Impossible to define coinductive types nested inside inductive types ($\mu\nu$) directly.
- Is this a (serious) issue?
- If so, how can we fix it?
Foetus

- Andreas Abel’s master thesis
- Closely related to size change termination (N. Jones et al)

mutual

\[
\begin{align*}
f : \mathbb{N} \rightarrow \mathbb{N} \\
f(m \text{ zero}) &= m \\
f(m \text{ (suc } n)) &= f(m \text{ } n + g(m)} \\
g : \mathbb{N} \rightarrow \mathbb{N} \\
g \text{ zero} &= \text{ zero} \\
g \text{ (suc } n) &= f(n \text{ } n)
\end{align*}
\]

\[
\begin{align*}
f \rightarrow f : (\frac{=}{} <) & \quad f \rightarrow g : (\frac{=}{} ?) \\
g \rightarrow f : (< <)
\end{align*}
\]
Coinductive Definitions in Agda

Streams:

\[
\text{data } \text{Stream } (A : \text{Set}) : \text{Set} \ \text{where} \\
_ :: _ : A \rightarrow \infty (\text{Stream } A) \rightarrow \text{Stream } A
\]

Categorically: \( \text{Stream } A = \nu X . A \times X \)

Force and Delay:

\[
\begin{align*}
\♭ & : \{ A : \text{Set} \} \rightarrow \infty A \rightarrow A \\
\♯_ & : \{ A : \text{Set} \} \rightarrow A \rightarrow \infty A
\end{align*}
\]

Corecursive programs

\[
\begin{align*}
\text{from} & : \mathbb{N} \rightarrow \text{Stream } \mathbb{N} \\
\text{from } n & = n :: \#\text{from} (\text{suc } n) \\
\text{mapStream} & : \forall \{ A B \} \rightarrow (A \rightarrow B) \rightarrow \text{Stream } A \rightarrow \text{Stream } B \\
\text{mapStream } f (a :: as) & = f a :: \#(\text{mapStream } f (\♭ as))
\end{align*}
\]
Functional representation of streams

Stream' : Set → Set
Stream' A = N → A

from' : N → (Stream' N)
from' n 0 = n
from' n (suc m) = from' (suc n) m

mapStream' : { A B : Set } → (A → B) → (Stream' A) → (Stream' B)
mapStream' f as 0 = f (as 0)
mapStream' f as (suc n) = mapStream' f (λ i → as (suc i)) n

Using subsets (Σ) such a representation (as an ω-limit) exists for all coinductive types.
Extending the termination checker

The translation suggests:

- Coinductive types introduce an additional (invisible) argument.
- Any use of ♯ reduces this argument.
- ♭ does not preserve the structural order.

\[
\begin{align*}
\text{from} &: \mathbb{N} \rightarrow \text{Stream } \mathbb{N} \\
\text{from } n &= n :: \#\text{from} (\text{suc } n)
\end{align*}
\]

\[
\begin{align*}
\text{from} \rightarrow \text{from} &: \mathbb{N} \\
\text{mapStream} &: \forall \{A, B\} \rightarrow (A \rightarrow B) \rightarrow \text{Stream } A \rightarrow \text{Stream } B \\
\text{mapStream } f (a :: as) &= f \ a :: \#(\text{mapStream } f (\text{♭} as))
\end{align*}
\]

\[
\begin{align*}
\text{mapStream} \rightarrow \text{mapStream} &: \mathbb{N} \\
\end{align*}
\]
Mixed induction/coinduction

Stream Processors:

```haskell
data SP (A B : Set) : Set where
    get : (A → SP A B) → SP A B
    put : B → ∞(SP A B) → SP A B
```

Categorical interpretation:

\[
SP A B = \nu X. \mu Y. A \rightarrow Y + B \times X
\]

In general:

```haskell
data D = F (∞D) D corresponds to
D = \nu X. \mu Y. F X Y.
```
**Semantics of SP**

\[
data \ SP \ (A \ B : \ Set) : \ Set \ where \\
get : (A \rightarrow SP \ A \ B) \rightarrow SP \ A \ B \\
put : B \rightarrow \infty (SP \ A \ B) \rightarrow SP \ A \ B
\]

Semantics of stream processors:

\[
[\_] : \{ A \ B : \ Set \} \rightarrow SP \ A \ B \rightarrow Stream \ A \rightarrow Stream \ B \\
[get \ f ] (a :: as) = [f a] (b \ as) \\
[put \ b \ sp \] as = b :: \# [b sp] as
\]

Extended Call graph

\[
[\_] \rightarrow [\_] : \\
\begin{pmatrix}
= & = & < & ? \\
< & = & ? & = \\
< & = & ? & ?
\end{pmatrix}
\]
Composition of SPs

Data driven:

\[ _{>\>>>\?\_ : \forall \{ A B C \} \rightarrow SP A B \rightarrow SP B C \rightarrow SP A C \] 
\[ \text{get } f \ _{>\>>>\? \_ \text{ tq } = \text{get } (\lambda a \rightarrow f \ a \ _{>\>>>\? \_ \text{ tq})} \] 
\[ \text{put } a \ sp \ _{>\>>>\? \_ \text{ get } f = \flat \ sp \ _{>\>>>\? \_ \text{ f a}} \] 
\[ \text{put } a \ sp \ _{>\>>>\? \_ \text{ put } b \ \text{tq } = \text{put } b \ (\sharp \text{put } a \ sp \ _{>\>>>\? \_ \text{ b tq)})} \]

Demand Driven

\[ _{>\>>>!\_ : \forall \{ A B C \} \rightarrow SP A B \rightarrow SP B C \rightarrow SP A C \] 
\[ \text{get } g \ _{>\>>>! \_ \text{ get } f = \text{get } (\lambda a \rightarrow g \ a \ _{>\>>>! \_ \text{ get } f})} \] 
\[ \text{put } b \ sp \ _{>\>>>! \_ \text{ get } f = \flat \ sp \ _{>\>>>! \_ \text{ f b}} \] 
\[ \text{sp} \ _{>\>>>! \_ \text{ put } c \ \text{tq } = \text{put } c \ (\sharp (sp \ _{>\>>>! \_ \text{ b tq})}) \]

- Both are accepted by the extended termination checker.
- Try to implement them using the categorical combinators.
From $\nu \mu$ to $\mu \nu$?

**data** $ZO : \text{Set}$ **where**

- $0 : ZO \to ZO$
- $1 : \infty ZO \to ZO$

$$ZO = \nu X. \mu Y. (0 : Y) + (1 : X)$$

$01^\omega : ZO$

$01^\omega = 0, (1, (\#01^\omega))$
From $\nu\mu$ to $\mu\nu$?

$$ZO' = \mu Y.\nu X.(0 : Y) + (1 : X)$$
$$= \mu Y.O X$$
with $O X = \nu X.0 : Y + 1 : X$

**data** $O (X : Set) : Set$ where
- $0, : X \rightarrow O X$
- $1, : \infty (O X) \rightarrow O X$

**data** $ZO' : Set$ where
- $emb : O ZO' \rightarrow ZO'$

But we can still define:

$$01^\omega : ZO'$$
$$01^\omega = emb (1, (\#0, 01^\omega))$$
No fold!

**mutual**

\[ fold : \forall \{ A \} \rightarrow (O A \rightarrow A) \rightarrow ZO' \rightarrow A \]
\[ fold f (\text{emb} \ x) = f (\text{mapfold} \ f \ x) \]

\[ \text{mapfold} : \forall \{ A \} \rightarrow (O A \rightarrow A) \rightarrow O ZO' \rightarrow O A \]
\[ \text{mapfold} f (0, x) = 0, (\text{fold} \ f \ x) \]
\[ \text{mapfold} f (1, x) = 1, (\#\text{mapfold} \ f \ (\flat x)) \]

is not accepted by the termination checker.
The problem is that \( \flat \) doesn’t preserve the structural order.
Otherwise we could derive a diverging program:

\[ \text{foo} : O ZO' \rightarrow ZO' \]
\[ \text{foo} (0, x) = x \]
\[ \text{foo} (1, x) = \text{emb} (\flat x) \]

\[ \text{bar} : ZO' \]
\[ \text{bar} = \text{fold} \ \text{foo} \ 01^\omega \]
Our attempt to define a $\mu\nu$-type by parametrisation fails.
We can define infinite elements which shouldn’t be there.
We cannot define fold (or induction) for the $\mu$-type.
What is going on?
Domain-theoretic explanation?

- $\infty A$ is interpreted as $A_\bot$ (lifting).
- Recursive datatypes as solutions to (strictly positive) domain equations.
- The termination checker identifies the total elements in the domain.
- $ZO = \nu X.\mu Y.0 : Y + 1 : X$ is interpreted as $\text{rec} X.\text{rec} Y.0 : Y + 1 : X_\bot$.
- $ZO' = \mu Y.\nu X.0 : Y + 1 : X$ is interpreted as $\text{rec} Y.\text{rec} X.0 : Y + 1 : X_\bot$.
- In general we have $\text{rec} X.\text{rec} Y.T X Y \simeq \text{rec} Y.\text{rec} X.T X Y$.
- Since the domains are isomorphic, they have the same total elements.
- How to define the total elements for a (strictly positive) domain equation in general?
We can explain parametrized types by mutual types.

```haskell
data O (X : Set) : Set where
  0, : X → O X
  1, : ∞ (O X) → O X

data ZO' : Set where
  emb : O ZO' → ZO'
```

becomes

```haskell
mutual
data O_ZO'' : Set where
  0, : ZO'' → O_ZO''
  1, : ∞ (O_ZO'') → O_ZO''

data ZO'' : Set where
  emb : O_ZO'' → ZO''
```

It is easy to see that ZO and ZO'' are isomorphic.
So what?

- We cannot easily define $\mu\nu$ types.
- There are extensionally isomorphic functional encodings.

```haskell
data Tree : Set where
  leaf : Tree
  node : Stream Tree -> Tree
```

can be encoded as

```haskell
data Tree' : Set where
  leaf : Tree'
  node : (Nat -> Tree') -> Tree'
```

- This also shows that data types may not preserve extensional isomorphism.
- Maybe $Tree$ should be forbidden by saying that $Tree$ doesn’t appear strictly positive in $Stream Tree$. 
Keiko and Tarmo’s encoding

- Coq doesn’t permit nested datatypes at all. (Not even $\mu\mu$).
- To represent $\nu\mu$ they use left Kan extensions. I.e. $FD$ is replaced by $\Sigma Y.(Y \to D) \times FY$.
- Can we use the same trick to encode $\mu\nu$ in Agda? (Switching off the universe checker).

```
data O (X : Set) : Set where
  0, : X → O X
  1, : ∞ (O X) → O X

data ZO : Set where
  emb : ∀{X} → (X → ZO) → O X → ZO
```

- Indeed, $fold$ is definable for this encoding!
- But so is $01\omega$.
- Indeed, Agda’s termination checker is unsound if we allow impredicativity (unlike Coq’s).
Nested $\nu$-types are not treated properly by Agda’s termination checker.

One solution is to outlaw them (we would be still better than Coq).

One can still use an extensionally isomorphic functional encoding.

Or can we fix the termination checker?

One idea is to combine parity games with size change termination.