Programming and Reasoning with Infinite Structures Using Copatterns and Sized Types

Andreas Abel

Department of Computer Science and Engineering
Chalmers and Gothenburg University, Sweden

7. Arbeitstagung Programmiersprachen
Kiel, Deutschland
26. Februar 2014

This is joint work with Brigitte Pientka.

Agda

- Purely functional programming language (Haskell family) with dependent types
- Types may mention values and programs
- Propositions-as-types: Types can include specifications
- Proofs-as-programs: Properties can be proven by terminating functions
- Agda 2 developed since 2006 (mostly at Chalmers)
- Formalization of algorithms, logics, lambda-caculus, function reactive programming etc.

Productivity Checking

- Coinductive structures: streams, processes, servers, continuous computation...
- Productivity: each request returns an answer after some time.
- Request on stream: *give me the next element*.
- Dependently typed languages have a productivity checker:

$$\mathsf{nats} = 0 :: \mathsf{map}\ (1 + \_)\ \mathsf{nats}$$

- Rejected by Coq and Agda's syntactic guardedness check.

Fibonacci Stream

- Recurrence for Fibonacci numbers:

			0	1	1	2	3	5	8	...
adds		0	1	1	2	3	5	8	13	...
	0	1	1	2	3	5	8	13	21	...

- Elegant implementation:

$$\text{fib} = 0 :: 1 :: \text{adds fib (fib.tail)}$$

- Rejected by guardedness check.

Coinduction and Dependent Types

- Consider the corecursively defined stream $a :: a :: a :: \ldots$

$$\text{repeat}\ a = a :: \text{repeat}\ a$$

- A dilemma:
  - Checking dependent types needs strong reduction.
  - Corecursion needs lazy evaluation.
- The current compromise (Coq, Agda):

  Corecursive definitions are unfolded only under elimination.

$$\begin{array}{lclcl} \text{repeat}\ a & \not\longrightarrow & & & \\ (\text{repeat}\ a).\text{tail} & \longrightarrow & (a :: \text{repeat}\ a).\text{tail} & \longrightarrow & \text{repeat}\ a \end{array}$$

- Reduction is context-sensitive.

Issues with Context-Sensitive Reduction

- Subject reduction is lost (Giménez 1996, Oury 2008).
- The Fibonacci stream is still diverging:

$$\mathsf{fib} = 0 :: 1 :: \mathsf{adds\ fib\ (fib.tail)}$$

$$\begin{array}{lll} \mathsf{fib.tail} & \longrightarrow & 1 :: \mathsf{adds\ fib\ (fib.tail)} \\ & \longrightarrow & 1 :: \mathsf{adds\ fib\ (}1 :: \mathsf{adds\ fib\ (fib.tail))} \\ & \longrightarrow & \dots \end{array}$$

- At POPL 2013, we presented a solution:

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer.
Copatterns: Programming infinite structures by observations.
In *POPL'13*, pages 27–38. ACM, 2013.

Copatterns — The Principle

- Define infinite objects (streams, functions) by observations.
- A function is defined by its applications.
- A stream by its head and tail.

$$\begin{aligned} \text{repeat}\ a\ .\text{head} &= a \\ \text{repeat}\ a\ .\text{tail} &= \text{repeat}\ a \end{aligned}$$

- These equations are taken as reduction rules.
- repeat a does not reduce by itself.
- No extra laziness required.

Deep Observations

- Any covering set of observations allowed for definition:

$$\begin{aligned} \text{fib.head} &= 0 \\ \text{fib.tail.head} &= 1 \\ \text{fib.tail.tail} &= \text{adds fib (fib.tail)} \end{aligned}$$

- Now fib.tail is stuck. Good!

Depth	0	1	2	...
Observations	id	.head	.tail.head	...
		.tail	.tail.tail	...

Stream Productivity

Definition (Productive Stream)

A stream is productive if all observations on it converge.

- Example of non-productiveness:

$$\text{bla} = 0 :: \text{bla}.\text{tail}$$

- Observation bla.tail diverges.
- This is apparent in copattern style...

$$\begin{array}{lcl} \text{bla}\ .\text{head} & = & 0 \\ \text{bla}\ .\text{tail} & = & \text{bla}\ .\text{tail} \end{array}$$

Proving Productivity

Theorem (repeat is productive)

repeat a .tail^n *converges for all* $n \geq 0$.

Proof.

By induction on n.

Base (repeat a).tail^0 = repeat a does not reduce.

Step (repeat a).tail^{n+1} = (repeat a).tail.tail^n $\longrightarrow$ (repeat a).tail^n which converges by induction hypothesis.

□

Productive Functions

Definition (Productive Function)

A function on streams is productive if it maps productive streams to productive streams.

$$\begin{aligned}(\text{adds}\ s\ t).\text{head} &= s.\text{head} + t.\text{head} \\ (\text{adds}\ s\ t).\text{tail} &= \text{adds}\ (s.\text{tail})\ (t.\text{tail})\end{aligned}$$

- *Productivity* of adds not sufficient for fib!
- Malicious adds:

$$\begin{aligned}\text{adds}'\ s\ t &= t.\text{tail} \\ \text{fib.tail.tail} &\longrightarrow \text{adds}'\ \text{fib}\ (\text{fib.tail}) \\ &\longrightarrow \text{fib.tail.tail} \longrightarrow \dots\end{aligned}$$

i-Productivity

Definition (Productive Stream)

A stream s is i-productive if all observations of depth $< i$ converge.
Notation: $s : \mathsf{Stream}^i$.

Lemma

$\mathsf{adds} : \mathsf{Stream}^i \to \mathsf{Stream}^i \to \mathsf{Stream}^i$ *for all* i*.*

Theorem

fib *is* i*-productive for all* i*.*

Proof, case $i + 2$: Show fib is $(i + 2)$-productive.

Show fib.tail.tail is i-productive.
IH: fib is $(i + 1)$-productive, so fib is i-productive. (Subtyping!)
IH: fib is $(i + 1)$-productive, so fib.tail is i-productive.
By Lemma, adds fib (fib.tail) is i-productive. □

Type System for Productivity

- "Church F^ω with inflationary and deflationary fixed-point types".
- Coinductive types = deflationary iteration:

$$\mathsf{Stream}^i A = \bigcap_{j<i} (A \times \mathsf{Stream}^j A)$$

- Bidirectional type-checking:
- Type inference $\boxed{\Gamma \vdash r \rightrightarrows A}$ and checking $\boxed{\Gamma \vdash t \leftleftarrows A}$.

$$\dfrac{\dfrac{\Gamma \vdash r \rightrightarrows \mathsf{Stream}^i A}{\Gamma \vdash r\,.\mathsf{tail} \rightrightarrows \forall j{<}i.\, \mathsf{Stream}^j A} \qquad \Gamma \vdash a < i}{\Gamma \vdash r\,.\mathsf{tail}\ a : \mathsf{Stream}^a A}$$

Conclusions

- A unified approach to termination and productivity: Induction.
  - Recursion as induction on data size.
  - Corecursion as induction on observation depth.
- Adaption of sized types to deep (co)patterns:
  - Shift to in-/deflationary fixed-point types.
  - Bounded size quantification.
- Implementations:
  - MiniAgda: ready to play with!
  - Agda (with James Chapman): in development version, planned for next release (2.3.4).

Andreas Abel and Brigitte Pientka.
Wellfounded recursion with copatterns:
A unified approach to termination and productivity.
International Conference on Functional Programming (ICFP 2013).

Some Related Work

- Sized types: many authors (1996–)
- Inflationary fixed-points: Dam & Sprenger (2003)
- Observation-centric coinduction and coalgebras: Hagino (1987), Cockett & Fukushima (Charity, 1992)
- Focusing sequent calculus: Zeilberger & Licata & Harper (2008)
- Form of termination measures taken from Xi (2002)

Copattern typing

- Fibonacci again (official syntax with explicit sizes).

$$
\begin{array}{llll}
\mathsf{fib} & : \forall i.\ |i| \Rightarrow \mathsf{Stream}^i \mathbb{N} & & \\
\mathsf{fib}\ i\ .\mathsf{head}\ j & & = & 0 \\
\mathsf{fib}\ i\ .\mathsf{tail}\ \ j\ .\mathsf{head}\ k & & = & 1 \\
\mathsf{fib}\ i\ .\mathsf{tail}\ \ j\ .\mathsf{tail}\ \ k & & = & \mathsf{adds}\ k\ (\mathsf{fib}\ k)\ (\mathsf{fib}\ j\ .\mathsf{tail}\ k)
\end{array}
$$

- Copattern inference $\boxed{\Delta \mid A \vdash \vec{q} \rightrightarrows C}$ (linear).

$$
\dfrac{\dfrac{\dfrac{\dfrac{\cdot \mid \mathsf{Stream}^k \mathbb{N} \vdash \cdot \rightrightarrows \mathsf{Stream}^k \mathbb{N}}{k{<}j \mid \forall k{<}j.\, \mathsf{Stream}^k \mathbb{N} \vdash k \rightrightarrows \mathsf{Stream}^k \mathbb{N}}}{k{<}j \mid \mathsf{Stream}^j \mathbb{N} \vdash .\mathsf{tail}\ k \rightrightarrows \mathsf{Stream}^k \mathbb{N}}}{j{<}i, k{<}j \mid \forall j{<}i.\, \mathsf{Stream}^j \mathbb{N} \vdash j\ .\mathsf{tail}\ k \rightrightarrows \mathsf{Stream}^k \mathbb{N}}}{j{<}i, k{<}j \mid \mathsf{Stream}^i \mathbb{N} \vdash .\mathsf{tail}\ j\ .\mathsf{tail}\ k \rightrightarrows \mathsf{Stream}^k \mathbb{N}}
$$

- Type of recursive call $\mathsf{fib} : \forall i'{<}i.\ \mathit{Stream}^{i'} \mathbb{N}$

Pattern typing rules

$\boxed{\Delta; \Gamma \vdash_{\Delta_0} p \Leftarrow A}$ Pattern typing (linear).
In: Δ_0, p, A with $\Delta_0 \vdash A$. Out: Δ, Γ with $\Delta_0, \Delta; \Gamma \vdash p \Leftarrow A$.

$$\frac{}{\cdot; x{:}A \vdash_{\Delta_0} x \Leftarrow A} \qquad \frac{}{\cdot; \cdot \vdash_{\Delta_0} () \Leftarrow 1}$$

$$\frac{\Delta_1; \Gamma_1 \vdash_{\Delta_0} p_1 \Leftarrow A_1 \qquad \Delta_2; \Gamma_2 \vdash_{\Delta_0} p_2 \Leftarrow A_2}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash_{\Delta_0} (p_1, p_2) \Leftarrow A_1 \times A_2}$$

$$\frac{\Delta; \Gamma \vdash_{\Delta_0} p \Leftarrow \exists j{<}a^{\uparrow}.\, S_c\,(\mu^j S)}{\Delta; \Gamma \vdash_{\Delta_0} c\, p \Leftarrow \mu^a S} \qquad \frac{\Delta; \Gamma \vdash_{\Delta_0, X:\kappa} p \Leftarrow F \; @^{\kappa} \; X}{X{:}\kappa, \Delta; \Gamma \vdash_{\Delta_0} {}^{X}p \Leftarrow \exists_\kappa F}$$

Copattern typing rules

$\boxed{\Delta; \Gamma \mid A \vdash_{\Delta_0} \vec{q} \rightrightarrows C}$ Pattern spine typing. In: $\Delta_0, A, \vec{q}$ with $\Delta_0 \vdash A$.
Out: Δ, Γ, C with $\Delta_0, \Delta; \Gamma \vdash C$ and $\Delta_0, \Delta; \Gamma, z{:}A \vdash z\,\vec{q} \rightrightarrows C$.

$$\frac{}{\cdot; \cdot \mid A \vdash_{\Delta_0} \cdot \rightrightarrows A} \qquad \frac{\Delta_1; \Gamma_1 \vdash_{\Delta_0} p \Leftarrow A \qquad \Delta_2; \Gamma_2 \mid B \vdash_{\Delta_0} \vec{q} \rightrightarrows C}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \mid A \to B \vdash_{\Delta_0} p\,\vec{q} \rightrightarrows C}$$

$$\frac{\Delta; \Gamma \mid \forall j{<}a^{\uparrow}.\, R_d\,(\nu^j R) \vdash_{\Delta_0} \vec{q} \rightrightarrows C}{\Delta; \Gamma \mid \nu^a R \vdash_{\Delta_0} .d\,\vec{q} \rightrightarrows C} \qquad \frac{\Delta; \Gamma \mid F \mathbin{@^{\kappa}} X \vdash_{\Delta_0, X:\kappa} \vec{q} \rightrightarrows C}{X{:}\kappa, \Delta; \Gamma \mid \forall_\kappa F \vdash_{\Delta_0} X\,\vec{q} \rightrightarrows C}$$

Semantics

- Reduction:

$$\frac{\vec{e} \;/\; \vec{q} \searrow \sigma}{\lambda\{\vec{q} \to t\}\, \vec{e}\, \vec{e}\,' \mapsto t\sigma\, \vec{e}\,'} \qquad \frac{\lambda D_k\, \vec{e} \mapsto t}{f\, \vec{e} \mapsto t}\; (f{:}A = \vec{D}) \in \Sigma$$

- Types are reducibility candidates $\mathcal{A}$:
  - $\mathcal{A}$ is a set of strongly normalizing terms.
  - $\mathcal{A}$ is closed under reduction.
  - $\mathcal{A}$ is closed under addition of well-behaved neutrals (redexes and terminally stuck terms).
  - $\mathcal{A}$ is closed under simulation:
    r is simulated by $r_{1..n}$ if $r\, \vec{e} \mapsto t$ implies $r_k\, \vec{e} \mapsto t$ for some k.