An ad-hoc and monolithic method for ensuring that corecursive definitions are productive

Nils Anders Danielsson

Nottingham

AIM 9, 2008-11-27

Hamming numbers (almost)

An ordered stream of all products of 2 and 3:

$$\begin{aligned} \mathit{hamming} \;=\; 1 : \mathit{merge}\ &(\mathit{map}\ (2\ *)\ \mathit{hamming}) \\ &(\mathit{map}\ (3\ *)\ \mathit{hamming}) \end{aligned}$$

- Productive?
- How can we get Agda to believe that it is?

Hamming numbers (almost)

An ordered stream of all products of 2 and 3:

$$\begin{aligned} \mathit{hamming} \; = \; 1 : \mathit{merge} \; & (\mathit{map} \; (2 \, *) \; \mathit{hamming}) \\ & (\mathit{map} \; (3 \, *) \; \mathit{hamming}) \end{aligned}$$

- Productive?
- How can we get Agda to believe that it is?

Hamming numbers (almost)

An ordered stream of all products of 2 and 3:

$$\begin{aligned} \mathit{hamming} \;=\; 1 : \mathit{merge}\ &(\mathit{map}\ (2\ *)\ \mathit{hamming}) \\ &(\mathit{map}\ (3\ *)\ \mathit{hamming}) \end{aligned}$$

- Productive?
- How can we get Agda to believe that it is?

One method

1. Define problem-specific language.
2. Implement provably productive interpreter.

The implementation can take advantage of the host language's productivity checker.

Disclaimer:

Hopefully this method will soon become obsolete.

…because it is awkward to use in practice.

However:

- Interesting to see what can be done without adding new features.
- Flexible.

How does it work?

Back to the example

$$\begin{array}{l}\textbf{codata}\ \mathit{Stream}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\ \quad \_\!\prec\!\_ : A \rightarrow \mathit{Stream}\ A \rightarrow \mathit{Stream}\ A\end{array}$$

Hamming numbers again

$$
\begin{aligned}
&\mathit{hamming} \ : \ \mathit{Stream}\ \mathbb{N} \\
&\mathit{hamming} \sim 1 \prec \mathit{merge}\ (\mathit{map}\ (\_{*}\_\ 2)\ \mathit{hamming}) \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{map}\ (\_{*}\_\ 3)\ \mathit{hamming})
\end{aligned}
$$

- Not guarded by constructors.
- But what if *merge* and *map* were constructors?

Ad-hoc programming language

```
mutual
  codata WHNF : Set → Set1 where
    _≺_ : forall {A} → A → Prog (Stream A) →
          WHNF (Stream A)

  data Prog : Set → Set1 where
    ↓_    : forall {A} → WHNF A → Prog A
    map   : forall {A B} → (A → B) →
            Prog (Stream A) → Prog (Stream B)
    merge : Prog (Stream ℕ) →
            Prog (Stream ℕ) →
            Prog (Stream ℕ)
```

Guarded definition

$$
\begin{aligned}
\mathit{hamming} &: \mathit{Prog}\ (\mathit{Stream}\ \mathbb{N}) \\
\mathit{hamming} &\sim \downarrow 1 \prec \mathsf{merge}\ (\mathsf{map}\ (\_*\_\ 2)\ \mathit{hamming}) \\
&\qquad\qquad\qquad\qquad (\mathsf{map}\ (\_*\_\ 3)\ \mathit{hamming})
\end{aligned}
$$

- Guarded by constructors.
- $\_\prec\_$ is a coconstructor.
- Note: Corecursive definition of inductive value.

Interpreter

1. One-step evaluator:

   $$\mathit{whnf} \ : \ \textbf{forall} \ \{A\} \ \rightarrow \ \mathit{Prog} \ A \ \rightarrow \ \mathit{WHNF} \ A$$

   Recursive: WHNF always reached in finite time.

2. Full evaluation:

   $$\begin{aligned}&\mathit{value} \ : \ \textbf{forall} \ \{A\} \ \rightarrow \ \mathit{WHNF} \ A \ \rightarrow \ A\\&\mathit{value} \ (x \prec \mathit{prog}) \sim x \prec \mathit{value} \ (\mathit{whnf} \ \mathit{prog})\end{aligned}$$

   $$\begin{aligned}&[\![\_]\!] \ : \ \textbf{forall} \ \{A\} \ \rightarrow \ \mathit{Prog} \ A \ \rightarrow \ A\\&[\![\ \mathit{prog}\ ]\!] \ = \ \mathit{value} \ (\mathit{whnf} \ \mathit{prog})\end{aligned}$$

   Uses guarded corecursion.

One-step evaluator

Structurally recursive:

$$
\begin{array}{l}
\mathit{whnf} \;:\; \textbf{forall}\,\{A\} \to \mathit{Prog}\ A \to \mathit{WHNF}\ A \\
\mathit{whnf}\ ({\downarrow}\, w) \;=\; w \\[4pt]
\mathit{whnf}\ (\mathsf{map}\ f\ \mathit{xs})\ \textbf{with}\ \mathit{whnf}\ \mathit{xs} \\
\ldots \mid x \prec \mathit{xs}' \;=\; f\ x \prec \mathsf{map}\ f\ \mathit{xs}' \\[4pt]
\mathit{whnf}\ (\mathsf{merge}\ \mathit{xs}\ \mathit{ys})\ \textbf{with}\ \mathit{whnf}\ \mathit{xs} \mid \mathit{whnf}\ \mathit{ys} \\
\ldots \mid x \prec \mathit{xs}' \mid y \prec \mathit{ys}'\ \textbf{with}\ \mathit{cmp}\ x\ y \\
\ldots \mid \mathsf{lt} \;=\; x \prec \mathsf{merge}\ \mathit{xs}'\ \mathit{ys} \\
\ldots \mid \mathsf{eq} \;=\; x \prec \mathsf{merge}\ \mathit{xs}'\ \mathit{ys}' \\
\ldots \mid \mathsf{gt} \;=\; y \prec \mathsf{merge}\ \mathit{xs}\ \mathit{ys}'
\end{array}
$$

Wrapping up

$$\begin{aligned} \mathit{ham} &: \mathit{Stream}\ \mathbb{N} \\ \mathit{ham} &= [\![\ \mathit{hamming}\ ]\!] \end{aligned}$$

Perhaps one should also prove that *ham* satisfies its intended defining equation.

What happens with unproductive code?

Productivity ⇒ termination

Productivity problems are sometimes turned into termination problems:

$$
\begin{aligned}
&\mathit{map}_2 \;:\; \textbf{forall}\ \{A\ B\} \;\rightarrow\; (A \;\rightarrow\; B) \;\rightarrow \\
&\qquad\qquad \mathit{Prog}\ (\mathit{Stream}\ A) \;\rightarrow\; \mathit{Prog}\ (\mathit{Stream}\ B) \\
&\mathit{map}_2\ f\ (x \prec x' \prec xs'') \sim f\ x \prec f\ x' \prec \mathit{map}_2\ f\ xs''
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hamming} \;:\; \mathit{Stream}\ \mathbb{N} \\
&\mathit{hamming} \sim 1 \prec \mathit{merge}\ (\mathit{map}_2\ (\_{*}\_\ 2)\ \mathit{hamming}) \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{map}_2\ (\_{*}\_\ 3)\ \mathit{hamming})
\end{aligned}
$$

Productivity $\Rightarrow$ termination

Productivity problems are sometimes turned into termination problems:

$$\textbf{data}\ \mathit{Prog} : \mathit{Set} \to \mathit{Set1}\ \textbf{where}$$
$$\quad \mathsf{map}_2 : \textbf{forall}\ \{A\ B\} \to (A \to B) \to$$
$$\qquad \mathit{Prog}\ (\mathit{Stream}\ A) \to \mathit{Prog}\ (\mathit{Stream}\ B)$$

$$\mathit{whnf}\ (\mathsf{map}_2\ f\ xs)\ \textbf{with}\ \mathit{whnf}\ xs$$
$$\ldots \mid x \prec xs'\ \textbf{with}\ \mathit{whnf}\ xs'$$
$$\ldots \mid x' \prec xs'' = f\ x \prec (\downarrow f\ x' \prec \mathsf{map}_2\ f\ xs'')$$

How far can this be taken?

Flexibility

It is possible to handle map_2:

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \textbf{data}\ \mathit{WHNF}_2\ :\ \mathit{Set}\ \rightarrow\ \mathit{Set1}\ \textbf{where} \\
\quad\quad \langle\_,\_\rangle\!\prec\!\_\ :\ \textbf{forall}\ \{A\}\ \rightarrow \\
\quad\quad\quad A\ \rightarrow\ A\ \rightarrow\ \mathit{Prog}_2\ (\mathit{Stream}\ A)\ \rightarrow \\
\quad\quad\quad \mathit{WHNF}_2\ (\mathit{Stream}\ A)
\end{array}
$$

Flexibility

It is possible to handle map_2:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Prog}_2\ :\ \mathit{Set}\ \rightarrow\ \mathit{Set1}\ \textbf{where}} \\
\quad \downarrow\_ & : & \textbf{forall}\ \{A\}\ \rightarrow \\
 & & \mathit{WHNF}_2\ A\ \rightarrow\ \mathit{Prog}_2\ A \\
\quad \mathsf{map}_2 & : & \textbf{forall}\ \{A\ B\}\ \rightarrow \\
 & & (A\ \rightarrow\ B)\ \rightarrow \\
 & & \mathit{Prog}_2\ (\mathit{Stream}\ A)\ \rightarrow\ \mathit{Prog}_2\ (\mathit{Stream}\ B)
\end{array}
$$

Flexibility

It is possible to handle map_2:

$$
\begin{aligned}
&\mathit{whnf}_2 \ : \ \textbf{forall} \ \{A\} \ \rightarrow \ \mathit{Prog}_2 \ A \ \rightarrow \ \mathit{WHNF}_2 \ A \\
&\mathit{whnf}_2 \ (\downarrow w) \ = \ w \\
&\mathit{whnf}_2 \ (\mathsf{map}_2 \ f \ \mathit{xs}) \ \textbf{with} \ \mathit{whnf}_2 \ \mathit{xs} \\
&\ldots \ | \ \langle \, x \ , \ x' \, \rangle\!\prec \mathit{xs}'' \ = \ \langle \, f \ x \ , \ f \ x' \, \rangle\!\prec \mathsf{map}_2 \ f \ \mathit{xs}''
\end{aligned}
$$

Flexibility

- Can be generalised from 2 to larger depths.
- Functions like *tail* can be handled.
  (But a coercion constructor may be necessary.)
- Can handle other types as well.
  - Breadth-first labelling of
    potentially infinite trees.

Equality proofs also possible

Unique fixed-points $\Rightarrow$ guarded coinduction:

$$
\begin{aligned}
&\mathit{iterate\text{-}fusion}\ h\ f_1\ f_2\ \mathit{hyp}\ x \sim \\
&\quad \mathrm{map}\ h\ (\mathrm{iterate}\ f_1\ x) \\
&\qquad \equiv\langle\ \equiv\text{-refl}\ \rangle \\
&\quad \downarrow h\ x \prec \mathrm{map}\ h\ (\mathrm{iterate}\ f_1\ (f_1\ x)) \\
&\qquad \approxeq\langle\ \downarrow \equiv\text{-refl} \prec \mathit{iterate\text{-}fusion}\ h\ f_1\ f_2\ \mathit{hyp}\ (f_1\ x)\ \rangle \\
&\quad \downarrow h\ x \prec \mathrm{iterate}\ f_2\ (h\ (f_1\ x)) \\
&\qquad \equiv\langle\ \equiv\text{-}\mathit{cong}\ (\backslash y\ \rightarrow\ [\![\ \downarrow h\ x \prec \mathrm{iterate}\ f_2\ y\ ]\!]) \\
&\qquad\qquad\qquad\qquad (\mathit{hyp}\ x)\ \rangle \\
&\quad \downarrow h\ x \prec \mathrm{iterate}\ f_2\ (f_2\ (h\ x)) \\
&\qquad \equiv\langle\ \equiv\text{-refl}\ \rangle \\
&\quad \mathrm{iterate}\ f_2\ (h\ x) \\
&\qquad \Box
\end{aligned}
$$

What about the drawbacks?

Drawbacks

- Ad-hoc.
- Monolithic.
- Awkward.
- Limited support for higher-order functions: $(Prog\ A \ \rightarrow\ Prog\ B) \ \rightarrow\ \ldots$ is negative.
- Inefficient: sharing lost.

Sharing lost

$$\_,\_ \; : \; \textbf{forall} \; \{ A \; B \} \; \rightarrow$$
$$\quad WHNF \; A \; \rightarrow \; WHNF \; B \; \rightarrow \; WHNF \; (A \times B)$$

$$\mathsf{fst} \; : \; \textbf{forall} \; \{ A \; B \} \; \rightarrow \; Prog \; (A \times B) \; \rightarrow \; Prog \; A$$

$$whnf \; (\mathsf{fst} \; prog) \; \textbf{with} \; whnf \; prog$$
$$\ldots \; | \; (x , y) \; = \; x$$

- Can perhaps be worked around by implementing a call-by-need interpreter. . .

Conclusion

- Fun to play around with. . .
- . . . but for real work we need something more convenient.
- What? (Andreas Abel might add to the discussion tomorrow.)