Sprinkles of extensionality for your vanilla type theory

Adding custom rewrite rules to Agda

Jesper Cockx Andreas Abel

DistriNet – KU Leuven

24 May 2016

What are we doing?

Take some vanilla Agda . . .

What are we doing?

Take some vanilla Agda . . .

. . . and sprinkle some rewrite rules on top

Goals of adding rewrite rules

1. Turn propositional equalities into definitional ones
2. Add new primitives with custom computation rules

Disclaimer

This is basically one big hack. We are not responsible for any unintended side-effects such as unsoundness, non-termination, lack of subject reduction, shark attacks, or zombie outbreaks.

Acknowledgements

A big thanks to

- Guillaume Brunerie
- Nils Anders Danielsson
- Martin Escardo

and other brave early adopters to point out bugs and limitations of the rewriting mechanism!

Sprinkles of extensionality for your vanilla type theory

1. What are rewrite rules?
2. What can you do with them?
3. How do they work?

Sprinkles of extensionality for your vanilla type theory

1. What are rewrite rules?
2. What can you do with them?
3. How do they work?

What are rewrite rules?

A way to plug new computation rules
into Agda's typechecker

```
plus0 : (n : ℕ) → n + 0 ≡ n
plus0 n = ...
{-# REWRITE plus0 #-}
```

This adds a computation rule $n + 0 \rightsquigarrow n$

What are rewrite rules?

A way to plug new computation rules into Agda's typechecker

$\texttt{plus0} : (n : \mathbb{N}) \to n + 0 \equiv n$

$\texttt{plus0}\ n = \ldots$

```
{-# REWRITE plus0 #-}
```

This adds a computation rule $n + 0 \rightsquigarrow n$

What are rewrite rules?

Applications the reflection rule ...

$$\frac{\Gamma \vdash e : u \equiv v}{u = v}$$

... but only for specific equality proofs $e \in \mathsf{Rew}$:

$$\frac{\Gamma \vdash e : f\ \bar{p} \equiv v \qquad \sigma : \Delta \Rightarrow \Gamma}{f\ \bar{p}\sigma \rightsquigarrow v\sigma}\,(e \in \mathsf{Rew})$$

What are rewrite rules not?

Not a conservative extension

- Can destroy termination
  e.g. $x + y \rightsquigarrow y + x$
- Can destroy confluence
  e.g. $\texttt{true} \rightsquigarrow \texttt{false}$
- Can destroy subject reduction
  e.g. $\texttt{subst}\ P\ e\ x \rightsquigarrow x$

Sprinkles of extensionality for your vanilla type theory

1. What are rewrite rules?
2. What can you do with them?
3. How do they work?

Make neutral terms reduce[1]

$$
\begin{aligned}
xs \mathbin{+\!\!+} [\,] &\rightsquigarrow xs \\
(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs &\rightsquigarrow xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) \\[1em]
\texttt{map}\ f\ (xs \mathbin{+\!\!+} ys) &\rightsquigarrow \texttt{map}\ f\ xs \mathbin{+\!\!+} \texttt{map}\ f\ ys \\
\texttt{map}\ (\lambda x.\, x)\ xs &\rightsquigarrow xs \\[1em]
\texttt{subst}\ (\lambda \_.\, B)\ p\ x &\rightsquigarrow x \\
\texttt{cong}\ (\lambda x.\, x)\ p &\rightsquigarrow p
\end{aligned}
$$

[1] See *New equations for neutral terms* by Guillaume Allais, Conor McBride, and Pierre Boutillier.

Implement higher inductive types

$$
\begin{aligned}
&\texttt{Circle} &&: \texttt{Set}\\
&\texttt{base} &&: \texttt{Circle}\\
&\texttt{loop} &&: \texttt{base} \equiv \texttt{base}\\
&\texttt{elim}_{\texttt{Circle}} &&: (P : \texttt{Circle} \to \texttt{Set})(b : P\ \texttt{base})\\
& && \quad (l : \texttt{subst}\ P\ \texttt{loop}\ b \equiv b)\\
& && \quad (x : \texttt{Circle}) \to P\ x
\end{aligned}
$$

$$
\begin{aligned}
\texttt{elim}_{\texttt{Circle}}\ P\ b\ l\ \ \texttt{base} &\rightsquigarrow b\\
\texttt{cong}\ (\texttt{elim}_{\texttt{Circle}}\ P\ b\ l)\ \texttt{loop} &\rightsquigarrow l
\end{aligned}
$$

Add custom resizing rules[2]

$\texttt{resize} : \texttt{Set}_i \to \texttt{Set}_j$

$\texttt{Prop}' : \texttt{Set}_1$

$\texttt{Prop}' = \Sigma[X : \texttt{Set}]\ ((x\ y : X) \to x \equiv y)$

$\texttt{Prop} : \texttt{Set}_0$

$\texttt{Prop} = \texttt{resize}\ \texttt{Prop}'$

[2]Based on code by Martin Escardo, see
`cs.bham.ac.uk/~mhe/impredicativity-via-rewriting/`

Add custom resizing rules[2]

$$\texttt{resize} : \texttt{Set}_i \to \texttt{Set}_j$$

$$\texttt{Prop}' : \texttt{Set}_1$$
$$\texttt{Prop}' = \Sigma[X : \texttt{Set}]\ ((x\ y : X) \to x \equiv y)$$

$$\texttt{Prop} : \texttt{Set}_0$$
$$\texttt{Prop} = \texttt{resize}\ \texttt{Prop}'$$

[2] Based on code by Martin Escardo, see `cs.bham.ac.uk/~mhe/impredicativity-via-rewriting/`

Do shallow embeddings: cubical[3]

$\mathtt{I} \quad : \mathtt{Set}$

$\mathtt{0}\ \mathtt{1} \quad : \mathtt{I}$

$\_ \longrightarrow \_ : A \to A \to \mathtt{Set}$

$\langle i \rangle\, t \quad : t[i \mapsto 0] \longrightarrow t[i \mapsto 1]$

$\_ \$ \_ \quad : (a \longrightarrow b) \to I \to A$

$\mathtt{funext} : ((x : A) \to f\ x \longrightarrow g\ x) \to (f \longrightarrow g)$

$\mathtt{funext}\ p = \langle i \rangle\, (\lambda x.\, p\ x \$ i)$

[3] Based on *A cubical crossroads* by Conor McBride at AIM XXIII, see `github.com/jespercockx/cubes` for the Agda code

Do shallow embeddings: cubical[3]

$$
\begin{aligned}
\texttt{I} &: \texttt{Set} \\
\texttt{0}\ \texttt{1} &: \texttt{I} \\
\_ — \_ &: A \to A \to \texttt{Set} \\
\langle i \rangle\ t &: t[i \mapsto 0] — t[i \mapsto 1] \\
\_ \$ \_ &: (a — b) \to I \to A
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{funext} : ((x : A) \to f\ x — g\ x) \to (f — g) \\
&\texttt{funext}\ p = \langle i \rangle\ (\lambda x.\ p\ x \$ i)
\end{aligned}
$$

---

[3]Based on *A cubical crossroads* by Conor McBride at AIM XXIII, see `github.com/jespercockx/cubes` for the Agda code

Sprinkles of extensionality for your vanilla type theory

1. What are rewrite rules?
2. What can you do with them?
3. How do they work?

Higher-order Miller matching

The LHS is compiled into a pattern $f\ p_1\ \ldots\ p_n$

- f should be a defined symbol or postulate
- patterns $p_1, \ldots, p_n$ should bind all variables

Patterns can be higher-order and non-linear

- $f\ p_1\ \ldots\ p_n$
- $\lambda x.p$
- $(x : P_1) \to P_2$
- `Set` p
- $x\ y_1 \ldots y_n$ (x free, y_i bound, $y_i \neq y_j$)
- $y\ p_1 \ldots p_n$ (y bound)
- Arbitrary terms t

Higher-order Miller matching

The LHS is compiled into a pattern $f\ p_1\ \ldots\ p_n$

- f should be a defined symbol or postulate
- patterns $p_1, \ldots, p_n$ should bind all variables

Patterns can be higher-order and non-linear

- $f\ p_1\ \ldots\ p_n$
- $\lambda x.p$
- $(x : P_1) \to P_2$
- `Set` p
- $x\ y_1 \ldots y_n$ (x free, y_i bound, $y_i \neq y_j$)
- $y\ p_1 \ldots p_n$ (y bound)
- Arbitrary terms t

Applying rewrite rules

How to rewrite $f\ t_1\ \ldots\ t_n$
with rewrite rule $f\ p_1\ \ldots\ p_n \leadsto r$?

1. $t_1 \ldots t_n$ are matched against linear part of $p_1 \ldots p_n$, producing a substitution σ
2. Non-linear parts are checked for equality after applying σ
3. $f\ t_1\ \ldots\ t_n$ is rewritten to $r\sigma$

Effects on constraint solving

- Previously inert terms can now reduce,
  so we have to postpone constraint solving
  E.g. $x + ?0 = x$
- Defined functions become matchable,
  so pruning has to be more conservative
  E.g. $?1\ (f\ x) =$ `true`

Effects on constraint solving

- Previously inert terms can now reduce,
  so we have to postpone constraint solving
  E.g. $x + ?\mathit{0} = x$

- Defined functions become matchable,
  so pruning has to be more conservative
  E.g. $?\mathit{1}\ (f\ x) = \texttt{true}$

Rewriting systems in type theory

Other systems based on rewrite rules:

- Dedukti (`dedukti.gforge.inria.fr`)
- CoqMT (`github.com/strub/coqmt`)
- ...

Future work

- Add proper import system
- Add confluence checking / completion
- Custom eta rules

Conclusion

You can use rewrite rules

- to simplify neutral terms such as $x + 0$
- to implement new primitives such as HIT's
- to embed other theories such as cubical

. . . but you should know what you are doing

Why don't you give it a try?

Conclusion

You can use rewrite rules

- to simplify neutral terms such as $x + 0$
- to implement new primitives such as HIT's
- to embed other theories such as cubical

…but you should know what you are doing

Why don't you give it a try?