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.
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

\[ \text{plus0} : (n : \mathbb{N}) \rightarrow n + 0 \equiv n \]

\[ \text{plus0 } n = \ldots \]

\{-# REWRITE plus0 #-\}

This adds a computation rule \( n + 0 \sim n \)
What are rewrite rules?

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

\[
\text{plus0} : (n : \mathbb{N}) \mapsto n + 0 \equiv n
\]

\text{plus0} \; n = \ldots

\{-# \text{REWRITE plus0} \; #-\}

This adds a computation rule \( n + 0 \leadsto n \)
What are rewrite rules?

Applications the reflection rule . . .

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

. . . but only for specific equality proofs \(e \in \text{Rew}\):

\[
\frac{\Gamma \vdash e : f \bar{p} \equiv v \quad \sigma : \Delta \Rightarrow \Gamma}{f \bar{p}\sigma \leadsto v\sigma} (e \in \text{Rew})
\]
What are rewrite rules not?

Not a conservative extension

- Can destroy termination
  - e.g. $x + y \leadsto y + x$

- Can destroy confluence
  - e.g. $\text{true} \leadsto \text{false}$

- Can destroy subject reduction
  - e.g. $\text{subst } P \ e \ x \leadsto x$
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

\[ \begin{align*}
  & xs ++ [] \quad \leadsto \quad xs \\
  & (xs ++ ys) ++ zs \quad \leadsto \quad xs ++ (ys ++ zs) \\
  & \text{map } f \ (xs ++ ys) \quad \leadsto \quad \text{map } f \ xs ++ \text{map } f \ ys \\
  & \text{map } (\lambda x. x) \ xs \quad \leadsto \quad xs \\
  & \text{subst } (\lambda_. B) \ p \ x \quad \leadsto \quad x \\
  & \text{cong } (\lambda x. x) \ p \quad \leadsto \quad p
\end{align*} \]

---

Implement higher inductive types

\[
\begin{align*}
\text{Circle} & : \text{Set} \\
\text{base} & : \text{Circle} \\
\text{loop} & : \text{base} \equiv \text{base} \\
\text{elim}_\text{Circle} : (P : \text{Circle} \rightarrow \text{Set})(b : P \text{ base}) \\
& \quad (l : \text{subst } P \text{ loop } b \equiv b) \\
& \quad (x : \text{Circle}) \rightarrow P \ x \\
\text{elim}_\text{Circle} & P \ b \ l \ \text{base} \sim b \\
\text{cong} & (\text{elim}_\text{Circle} P \ b \ l) \ \text{loop} \sim l
\end{align*}
\]
Add custom resizing rules

\textbf{resize} : \textbf{Set}_i \rightarrow \textbf{Set}_j

\textbf{Prop}' : \textbf{Set}_1

\textbf{Prop}' = \Sigma[X : \textbf{Set}] \ ( (x \ y : X) \rightarrow x \equiv y )

\textbf{Prop} : \textbf{Set}_0

\textbf{Prop} = \text{resize Prop}'

\footnote{Based on code by Martin Escardo, see \texttt{cs.bham.ac.uk/~mhe/impredicativity-via-rewriting/}}
Add custom resizing rules

\[ \text{resize} : \text{Set}_i \rightarrow \text{Set}_j \]

\[ \text{Prop}' : \text{Set}_1 \]
\[ \text{Prop}' = \Sigma[X : \text{Set}] \ ((x \ y : X) \rightarrow x \equiv y) \]

\[ \text{Prop} : \text{Set}_0 \]
\[ \text{Prop} = \text{resize Prop}' \]

---

\(^2\)Based on code by Martin Escardo, see cs.bham.ac.uk/~mhe/impredicativity-via-rewriting/
Do shallow embeddings: cubical

\[
I : \text{Set}
\]

\[
0 1 : I
\]

\[
_\_ \_ : A \rightarrow A \rightarrow \text{Set}
\]

\[
\langle i \rangle t : t[i \mapsto 0] \rightarrow t[i \mapsto 1]
\]

\[
\_ \_ : (a \rightarrow b) \rightarrow I \rightarrow A
\]

\[
\text{funext} : ((x : A) \rightarrow f x \rightarrow g x) \rightarrow (f \rightarrow g)
\]

\[
\text{funext} p = \langle i \rangle (\lambda x. p x \_ \_ i)
\]

---

3Based on *A cubical crossroads* by Conor McBride at AIM XXIII, see github.com/jespercockx/cubes for the Agda code
Do shallow embeddings: cubical\(^3\)

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

\[
\text{funext} : ((x : A) \to f \ x \to g \ x) \to (f \to g)
\]

\[
\text{funext} \ p = \langle i \rangle \ (\lambda x. \ p \ x \ $ i)
\]

\(^3\)Based on \textit{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

- $f \ p_1 \ldots \ p_n$
- $\lambda x. p$
- $(x : P_1) \rightarrow P_2$
- $\text{Set} \ p$
- $x \ y_1 \ldots \ y_n$ ($x$ 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

- $f \ p_1 \ldots \ p_n$
- $\lambda x. p$
- $(x : P_1) \rightarrow P_2$
- $\text{Set } p$
- $x \ y_1 \ldots \ y_n$ ($x$ 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 $\sigma$

2. Non-linear parts are checked for equality after applying $\sigma$

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. \(?1 (f \ x) = \text{true}\)
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. $?1 \ (f \ x) = \text{true}$
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?