Type theory unchained: Extending type theory with user-defined rewrite rules (draft of 12th November 2019)

Jesper Cockx
Department of Computer Science and Engineering, Göteborg University, Sweden
https://jesper.sikanda.be
jesper@sikanda.be

Abstract

Dependently typed languages such as Coq and Agda can statically guarantee the correctness of our proofs and programs. To provide this guarantee, they restrict users to certain schemes – such as strictly positive datatypes, complete case analysis, and well-founded induction – that are known to be safe. However, these restrictions can be too strict, making programs and proofs harder to write than necessary. On a higher level, they also prevent us from imagining the different ways the language could be extended.

In this paper I show how to extend a dependently typed language with user-defined higher-order non-linear rewrite rules. Rewrite rules are a form of equality reflection that is applied automatically by the typechecker. I have implemented rewrite rules as an extension to Agda, and I give six examples how to use them both to make proofs easier and to experiment with extensions of type theory. I also show how to make rewrite rules interact well with other features of Agda such as η-equality, implicit arguments, data and record types, irrelevance, and universe level polymorphism. Thus rewrite rules break the chains on computation and put its power back into the hands of its rightful owner: yours.

2012 ACM Subject Classification Theory of computation → Rewrite systems; Theory of computation → Equational logic and rewriting; Theory of computation → Type theory

Keywords and phrases Dependent types, Proof assistants, Rewrite rules, Higher-order rewriting, Agda

1 Introduction

As proclaimed by our prophet Per Martin-Löf [14], each type former in type theory is declared by four sets of rules:

- The **formation rule**, e.g. Bool : Set
- The **introduction rules**, e.g. true : Bool and false : Bool
- The **elimination rules**, e.g. if P : Bool → Set, b : Bool, pt : P true, and pf : P false, then if b then pt else pf : $P\ b$
- The **computation rules**, e.g. if true then pt else $pf = pt$ and if false then pt else $pf = pf$

When working in a proof assistant or dependently typed programming language, we usually do not introduce new types directly by giving these rules. That would be very unsafe, as there is no easy way to check that the given rules make sense. Instead, we introduce new rules through schemes that are well-known to be safe, such as strictly positive datatypes, complete case analysis, and well-founded induction.

However, users of dependently typed languages or researchers who are experimenting with adding new features to them might find working within these schemes too restrictive. They might be tempted to use postulate to simulate the formation, introduction, and elimination rules of new type formers. Yet there is one thing that cannot be added by using postulate: the computation rules.

This paper shows how to extend a dependently typed language with user-defined *rewrite rules*, allowing the user to extend the definitional equality of the language with new computation rules. Concretely, I extend the Agda language with a new option `--rewriting`. When this option is enabled, you can register a proof (or a postulate) $p : \forall x_1 \dots x_n \to f\ u_1\ \dots\ u_n \equiv v$ as a rewrite rule with a pragma `{-# REWRITE p #-}`. From this point on, Agda will automatically reduce instances of the left-hand side $f\ u_1\ \dots\ u_n$ (i.e. for specific values of $x_1\ \dots\ x_n$) to the corresponding instance of v. As a silly example, if $f : A \to A$ and $p : \forall x \to f\ x \equiv x$, then the rewrite rule will replace any application $f\ u$ with u, effectively turning f into the identity function $\lambda x \to x$.

The main goal of this paper is to specify in detail one possible way to add rewrite rules to a general-purpose dependently typed language. This is meant to serve at the same time as a specification of how rewrite rules are implemented in Agda and also as a guideline how they could be added to other languages.

Contributions

- I define a core type theory based on Martin-Löf's intentional type theory extended with user-defined higher-order non-linear rewrite rules.
- I describe how rewrite rules interact with several common features of dependently typed languages, such as η-equality, data and record types, parametrized modules, proof irrelevance, universe level polymorphism, and constraint solving for metavariables.
- I implement rewrite rules as an extension to Agda and show in six examples how to use them to make writing programs and proofs easier and to experiment with new extensions to Agda.

Sect. 2 consists of examples of how to use rewrite rules to go beyond the usual boundaries set by Agda and define your own computation rules. After these examples, Sect. 3 shows more generally how to add rewrite rules to a dependently typed language, and Sect. 4 shows how rewrite rules interact with other features of Agda. Related work and future work are discussed in Sect. 5 and Sect. 6, and Sect. 7 concludes.

2 Using rewrite rules

With the introduction out of the way, let us start with some examples of things you can do with rewrite rules. I hope at least one example gives you the itch to try rewrite rules for yourself. There are some restrictions on what kind of equality proofs can be turned into rewrite rules, which will be explained later in general. Until then, the examples should give an idea of the kind of things that are possible.

All examples in this section are accepted by Agda 2.6.0.1 [2]. We start with some basic options and imports. For the purpose of this paper, the two most important ones are the `--rewriting` flag and the import of Agda.Builtin.Equality.Rewrite, which are both required to make rewrite rules work. Meanwhile, the `--prop` flag enables Agda's Prop universe[1] [12], which will be used in some of the examples.

```
{−# OPTIONS --rewriting --prop #−}

open import Agda.Primitive
```

[1] `https://agda.readthedocs.io/en/v2.6.0.1/language/prop.html`

```
open import Agda.Builtin.Bool
open import Agda.Builtin.Nat
open import Agda.Builtin.List
open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
```

The examples in this paper make use of generalizable variables[2] to avoid writing many quantifiers and make the code more readable.

```
variable
  ℓ ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level
  A B C          : Set ℓ
  P Q            : A → Set ℓ
  x y z          : A
  f g h          : (x : A) → P x
  b b₁ b₂ b₃     : Bool
  k l m n        : Nat
  xs ys zs       : List A
  R              : A → A → Prop
```

We use the following helper function to annotate terms with their types:

```
infix 5 El
El : (A : Set ℓ) → A → A
El A x = x

syntax El A x = x ∈ A
```

To avoid reliance on external libraries, we also need two basic properties of equality:

```
cong : (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl

transport : (P : A → Set ℓ) → x ≡ y → P x → P y
transport P refl p = p
```

2.1 Overlapping pattern matching

To start, let us look at a question that is asked by almost every newcomer to Agda: why does $0 + m$ compute to m, but $m + 0$ does not? Similarly, why does $(\mathsf{suc}\ m) + n$ compute to $\mathsf{suc}\ (m + n)$ but $m + (\mathsf{suc}\ n)$ does not? This problem manifests for example when trying to prove commutativity of _+_ (the lack of highlighting is a sign that the code is not accepted by Agda):

```
+comm : m + n ≡ n + m
+comm {m = zero}  = refl
+comm {m = suc m} = cong suc (+comm {m = m})
```

[2] https://agda.readthedocs.io/en/v2.6.0.1/language/generalization-of-declared-variables.html

Here Agda complains that $n \neq n + \mathsf{zero}$. The problem is usually solved by proving the equations $m + 0 \equiv m$ and $m + (\mathsf{suc}\ n) \equiv \mathsf{suc}\ (m + n)$ and using an explicit rewrite[3] statement in the proof of +comm.

Despite solving the problem, this solution is rather disappointing: if Agda can tell that $0 + m$ computes to m, why not $m + 0$? During my master thesis, I worked on overlapping computation rules [10] to make this problem go away without adding any explicit rewrite statements. By using rewrite rules, we can simulate this solution in Agda. First, we need to prove that the equations we want hold as propositional equalities:

```
+zero : m + zero ≡ m
+zero {m = zero}    = refl
+zero {m = suc m} = cong suc +zero

+suc : m + (suc n) ≡ suc (m + n)
+suc {m = zero}    = refl
+suc {m = suc m} = cong suc +suc
```

Then we mark the equalities as rewrite rules with a REWRITE pragma:

```
{-# REWRITE +zero +suc #-}
```

Now the proof of commutativity works exactly as we wrote before:

```
+comm : m + n ≡ n + m
+comm {m = zero}    = refl
+comm {m = suc m} = cong suc (+comm {m = m})
```

Without rewrite rules there is **no** way to make this proof go through unchanged: it is essential that _+_ computes both on its first and second arguments, but there is no way to define _+_ in such a way using Agda's regular pattern matching.

2.2 New equations for neutral terms

The idea of extending existing functions with new computation rules has been taken much further by Allais, McBride, and Boutillier [3]. They extend classic functions on lists such as map, _++_ (concatenation), and fold with new equational rules for neutral expressions. In Agda, we can prove these rules and then add them as rewrite rules. For example, here are their rules for map and _++_:

```
map : (A → B) → List A → List B
map f []       = []
map f (x :: xs) = (f x) :: (map f xs)

infixr 5 _++_
_++_ : List A → List A → List A
[] ++ ys       = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

[3] Agda's rewrite keyword should not be confused with rewrite rules, which are added by a REWRITE pragma.

```
map-id : map (λ x → x) xs ≡ xs
map-id {xs = []}     = refl
map-id {xs = x :: xs} = cong (x ::_) map-id

map-fuse : map f (map g xs) ≡ map (λ x → f (g x)) xs
map-fuse {xs = []}     = refl
map-fuse {xs = x :: xs} = cong (_ ::_) map-fuse

map-++ : map f (xs ++ ys) ≡ (map f xs) ++ (map f ys)
map-++ {xs = []}     = refl
map-++ {xs = x :: xs} = cong (_ ::_) (map-++ {xs = xs})

{-# REWRITE map-id map-fuse map-++ #-}
```

These rules look simple, but can be quite powerful. For example, below we show that the expression map swap (map swap xs ++ map swap ys) reduces to xs ++ ys, without requiring any induction on lists.

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
open _×_

swap : A × B → B × A
swap (x , y) = y , x

test : map swap (map swap xs ++ map swap ys) ≡ xs ++ ys
test = refl
```

To compute the left-hand side of the equation to the right-hand side, Agda makes use of map-++ (step_1), map-fuse (step_2), built-in η-equality of _ × _ (step_3), the definition of swap (step_4), and finally the map-id rewrite rule (step_5).

```
step₁ : map swap (map swap xs ++ map swap ys)
      ≡ map swap (map swap xs) ++ map swap (map swap ys)
step₁ = refl

step₂ : map swap (map swap xs) ≡ map (λ x → swap (swap x)) xs
step₂ = refl

step₃ : map (λ x → swap (swap x)) xs ≡ map (λ x → swap (swap (fst x , snd x))) xs
step₃ = refl

step₄ : map (λ x → swap (swap (fst x , snd x))) xs ≡ map (λ x → (fst x , snd x)) xs
step₄ = refl

step₅ : map (λ x → (fst x , snd x)) xs ≡ xs
step₅ = refl
```

2.3 Higher inductive types

The original motivation for adding rewrite rules to Agda had little to do with adding new computation rules to existing functions as in the previous examples. Instead, its purpose was to experiment with defining higher inductive types [1]. In particular, it was meant as an alternative for people using clever (but horrible) hacks to make higher inductive types compute.[4]

A higher inductive type is similar to a regular inductive type D with some additional path constructors, which construct an element of the identity type $a \equiv b$ where $a : D$ and $b : D$. A classic example is the Circle type, which has one regular constructor base and one path constructor loop:

```
postulate
  Circle : Set
  base   : Circle
  loop   : base ≡ base

postulate
  Circle-elim : (P : Circle → Set ℓ) (base* : P base) (loop* : transport P loop base* ≡ base*)
              → (x : Circle) → P x
  elim-base : ∀ (P : Circle → Set ℓ) base* loop* → Circle-elim P base* loop* base ≡ base*
{-# REWRITE elim-base #-}
```

To specify the computation rule for Circle−elim applied to loop, we need the dependent version of cong, which is called apd in the book [1].

```
apd : (f : (x : A) → P x) (p : x ≡ y) → transport P p (f x) ≡ f y
apd f refl = refl

postulate
  elim-loop : ∀ (P : Circle → Set ℓ) base* loop* → apd (Circle-elim P base* loop*) loop ≡ loop*
{-# REWRITE elim-loop #-}
```

Without the rewrite rule elim−base, the type of elim−loop is not well-formed. So without rewrite rules, it is impossible to even state the computation rule of Circle−elim on the path constructor loop.

2.4 Quotient types

One of the well-known weak spots of intentional type theory is its poor handling of quotient types. One of the more promising attempts at adding quotients to Agda is by Guillaume Brunerie in the initiality project[5], which uses a combination of rewrite rules and Agda's new (strict) Prop universe.

Before I can show this definition of the quotient type, we first need to define the Prop-valued equality type _≐_. We also define its corresponding notion of transport, which has to be postulated due to current limitations in the implementation of Prop. To make transportR compute in the expected way, we add it as a rewrite rule transportR−refl.

---

[4] https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/
[5] https://github.com/guillaumebrunerie/initiality/blob/reflection/quotients.agda

```
data _≐_ {A : Set ℓ} (x : A) : A → Prop ℓ where
  refl : x ≐ x

postulate
  transportR : (P : A → Set ℓ) → x ≐ y → P x → P y
  transportR−refl : transportR P refl x ≡ x
{−# REWRITE transportR−refl #−}
```

Now we are ready to define the quotient type _//_. Given a type A and a Prop-valued relation $R : A \to A \to \mathsf{Prop}$, the type $A \mathbin{//} R$ consists of elements proj x where $x : A$, and proj x is equal to proj y if and only if $R\ x\ y$ holds.

```
postulate
  _//_ : (A : Set ℓ) (R : A → A → Prop) → Set ℓ
  proj  : A → A // R
  quot  : R x y → proj {R = R} x ≐ proj {R = R} y
```

The elimination principle //−elim allows us to define functions that extract an element of A from a given element of $A \mathbin{//} R$, provided a proof quot$*$ that the function respects the equality on $A \mathbin{//} R$. The computation rule //−beta allows //−elim to compute when it is applied to a proj x.

```
//−elim : (P : A // R → Set ℓ) (proj* : (x : A) → P (proj x))
        → (quot* : {x y : A} (r : R x y) → transportR P (quot r) (proj* x) ≐ proj* y)
        → (x : A // R) → P x
//−beta : {R : A → A → Prop} (P : A // R → Set ℓ) (proj* : (x : A) → P (proj x))
        → (quot* : {x y : A} (r : R x y) → transportR P (quot r) (proj* x) ≐ proj* y)
        → {u : A} → //−elim P proj* quot* (proj u) ≡ proj* u
{−# REWRITE //−beta #−}
```

Compared to the more standard way of defining the quotient type as a higher inductive type, this definition behaves better with respect to definitional equality: the argument quot$*$ to the eliminator is definitionally irrelevant, so it does not matter what equality proof we give. Consequently, there is no need to add an additional constructor to truncate the quotient type.

2.5 Exceptional type theory

First-class exceptions are a common feature of object-oriented programming languages such as Java, but in the world of pure functional languages they are usually frowned upon. However, recently Pédrot and Tabareau have proposed an extension of Coq with first-class exceptions [16]. With the exceptional power of rewrite rules, we can also encode (part of) their system in Agda.

First, we postulate a type Exc with any kinds of exceptions we might want to use (here we just have a single runtimeException for simplicity). We then add the possibility to raise an exception, producing an element of an arbitrary type A.

```
postulate
  Exc : Set
  runtimeException : Exc
  raise : Exc → A
```

Note that raise makes the type theory inconsistent. In their paper, Pédrot and Tabareau show how to build a safe version of exceptions on top of this system, using parametricity to enforce that all exceptions are caught locally. Here that part is omitted for brevity.

By adding the appropriate rewrite rules for each type former, we can ensure that exceptions are propagated appropriately. For positive types such as Nat, exceptions are propagated outwards, while for negative types such as function types, exceptions are propagated inwards.

```
postulate
  raise−suc : {e : Exc} → suc (raise e) ≡ raise e
  raise−fun : {e : Exc} → raise {A = (x : A) → P x} e ≡ λ x → raise {A = P x} e
{−# REWRITE raise−suc raise−fun #−}
```

To complete the system, we add the ability to catch exceptions at specific types. This takes the shape of an eliminator with one additional method for handling the case where the element under scrutiny is of the form raise e.

```
postulate
  catch−Bool : (P : Bool → Set ℓ) (pt : P true) (pf : P false)
               → (h : ∀ e → P (raise e)) → (b : Bool) → P b

  catch−true : ∀ (P : Bool → Set ℓ) pt pf h   → catch−Bool P pt pf h true     ≡ pt
  catch−false : ∀ (P : Bool → Set ℓ) pt pf h  → catch−Bool P pt pf h false    ≡ pf
  catch−exc  : ∀ (P : Bool → Set ℓ) pt pf h e → catch−Bool P pt pf h (raise e) ≡ h e
{−# REWRITE catch−true catch−false catch−exc #−}
```

As shown by this example, rewrite rules can be used to extend Agda with new primitive operations, including ones that compute according to the type of their arguments.

2.6 Observational equality

Rewrite rules also allow us to define type constructors that compute according to the type they are applied to. This is a core part of observational type theory (OTT) [4]. OTT replaces the usual identity type with an observational equality type (here called _≅_) that computes acoording to the type of the elements being compared. For example, an equality proof between pairs of type $(a\ ,\ b) \cong (c\ ,\ d)$ **is** a pair of proofs, one of type $a \cong c$ and one of type $b \cong d$.

Below, I show how to extend Agda with a fragment of OTT. Since OTT has a proof-irrelevant equality type, I use Agda's Prop to get the same effect. First, we need some basic types in Prop:

```
record ⊤ {ℓ} : Prop ℓ where constructor tt

data ⊥ {ℓ} : Prop ℓ where

record _∧_ (X : Prop ℓ₁) (Y : Prop ℓ₂) : Prop (ℓ₁ ⊔ ℓ₂) where
  constructor _,_
  field
    fst : X
    snd : Y
open _∧_
```

The central type of OTT is observational equality _≅_, which should compute according to the types of the elements being compared. Here I give the computation rules for Bool and for function types:

```
infix 6 _≅_
postulate
  _≅_ : {A : Set ℓ₁} {B : Set ℓ₂} → A → B → Prop (ℓ₁ ⊔ ℓ₂)

postulate
  refl-Bool  : (Bool ≅ Bool) ≡ ⊤
  refl-true  : (true ≅ true) ≡ ⊤
  refl-false : (false ≅ false) ≡ ⊤
  conflict-tf : (true ≅ false) ≡ ⊥
  conflict-ft : (false ≅ true) ≡ ⊥
{-# REWRITE refl-Bool refl-true refl-false conflict-tf conflict-ft #-}

postulate
  cong-Π : ((x : A) → P x) ≅ ((y : B) → Q y)
         ≡ (B ≅ A) ∧ ((x : A)(y : B) → y ≅ x → P x ≅ Q y)
  cong-λ : {A : Set ℓ₁} {B : Set ℓ₂} {P : A → Set ℓ₃} {Q : B → Set ℓ₄}
         → (f : (x : A) → P x) (g : (y : B) → Q y)
         → ((λ x → f x) ≅ (λ y → g y)) ≡ ((x : A) (y : B) (x≅y : x ≅ y) → f x ≅ g y)
{-# REWRITE cong-Π cong-λ #-}
```

According to cong-Π, an equality proof between function types computes to a pair of equality proofs between the domains and the codomains respectively. Though not necessary, it is convenient to swap the sides of the equality proofs in contravariant positions ($B \equiv A$ and $y \equiv x$). Meanwhile, an equality proof between two functions computes to an equality proof between the functions applied to heterogeneously equal variables $x : A$ and $y : B$.

To reason about equality proofs, OTT adds two more notions: **coercion** and **cohesion**. Coercion _[_⟩ transforms an element from one type to the other when both types are observationally equal, and cohesion _||_ states that coercion is computationally the identity.

```
infix 10 _[_⟩ _||_

postulate
  _[_⟩ : A → (A ≅ B) → B
  _||_ : (x : A) (Q : A ≅ B) → (x ∈ A) ≅ (x [ Q ⟩ ∈ B)
```

Again, we need rewrite rules to make sure coercion computes in the right way when applied to specific type constructors. On the other hand, We do not need rewrite rules for coherence since the result is of type _ ≅ _ which is a Prop, so the proof is anyway irrelevant.

Coercing an element from Bool to Bool is easy.

```
postulate
  coerce-Bool : (Bool≅Bool : Bool ≅ Bool) → b [ Bool≅Bool ⟩ ≡ b
{-# REWRITE coerce-Bool #-}
```

To coerce a function from $(x : A) \to P\ x$ to $(y : B) \to Q\ y$ we need to:

1. Coerce the input from $y : B$ to $x : A$

2. Apply the function to get an element of type $P\ x$
3. Coerce the output back to an element of $Q\ y$

In the last step, we need to use coherence to show that x and y are (heterogeneously) equal.

```
postulate
  coerce−Π : {A : Set ℓ₁} {B : Set ℓ₂} {P : A → Set ℓ₃} {Q : B → Set ℓ₄} {f : (x : A) → P x}
    → (ΠAP≅ΠBQ : ((x : A) → P x) ≅ ((y : B) → Q y))
    → (f [ ΠAP≅ΠBQ ⟩ ∈ ((y : B) → Q y))
    ≡ (λ (y : B) →
        let B≅A    = fst ΠAP≅ΠBQ
            x      = y [ B≅A ⟩
            Px≅Qy = snd ΠAP≅ΠBQ x y (_||_ {B = A} y B≅A)
        in f x [ Px≅Qy ⟩)
{−# REWRITE coerce−Π #−}
```

Of course this is just a fragment of the whole system, but implementing all of OTT would go beyond the scope of this paper. In principle, observational equality can be used as a full replacement for Agda's built-in equality type. So rewrite rules are even powerful enough to experiment with replacements for core parts of Agda.

3 Type theory with user-defined rewrite rules

In the previous section, I gave several examples of how to use rewrite rules in Agda to make programming and proving easier and to experiment with new extensions to type theory. The next two sections go into the details of how to rewrite rules work in general.

Instead of starting with a complex language like Agda, I start with a small core language and gradually extend it by adding more features to the rewriting machinery step by step. In the next section, I will extend this language with other features that you are used to from Agda. The full rules of the language can be found in Appendix A.

3.1 Syntax

The syntax has five constructors: variables, function symbols, lambdas, pi types, and universes.

$$
\begin{array}{lrll}
\boxed{\text{u, v, A, B}} & ::= & x\ \bar{u} & \text{(variable applied to arguments)} \\
& \mid & \mathsf{f}\ \bar{u} & \text{(function symbol applied to arguments)} \\
& \mid & \lambda x.\, u & \text{(lambda abstraction)} \\
& \mid & (x : A) \to B & \text{(dependent function type)} \\
& \mid & \mathsf{Set}_i & (i\text{th universe})
\end{array}
\tag{1}
$$

As in the internal syntax of Agda, there is no way to represent a β-redex in this syntax. Instead, substitution $\boxed{u\sigma}$ is defined to eagerly reduce β-redexes on the fly.

Contexts are right-growing lists of variables annotated with their types.

$$
\begin{array}{lrll}
\Gamma, \Delta & ::= & \cdot & \text{(empty context)} \\
& \mid & \Gamma(x : A) & \text{(context extension)}
\end{array}
\tag{2}
$$

Patterns $\boxed{p, q}$ share their syntax with regular terms, but must satisfy some additional restrictions. To start with, the only allowed patterns are unapplied variables x and applications of function symbols to other patterns $\mathsf{f}\ \bar{p}$. This allows us for example to declare rewrite rules like $\mathsf{plus}\ x\ \mathsf{zero} \longrightarrow x$ and $\mathsf{plus}\ x\ (\mathsf{suc}\ y) \longrightarrow \mathsf{suc}\ (x + y)$.

3.2 Declarations

There are two kinds of declarations: function symbols (corresponding to a postulate in Agda) and rewrite rules (corresponding to a postulate + a `{-# REWRITE #-}` pragma).

$$\begin{array}{rcll} \boxed{\mathrm{d}} & ::= & \mathsf{f} : A & \text{(function symbol)} \\ & | & \forall \Delta.\, \mathsf{f}\ \bar{p} : A \longrightarrow v & \text{(rewrite rule)} \end{array} \tag{3}$$

When the user declares a new rewrite rule, the following properties are checked:

Linearity Each variable in Δ must occur exactly once in the pattern $\bar{p}$ (this will later be relaxed to 'at least once').

Well-typedness The left- and right-hand side of the rewrite rule must be well-typed, i.e. $\Delta \vdash \mathsf{f}\ \bar{p} : A$ and $\Delta \vdash v : A$.

Neutrality The left-hand side of the rewrite rule should be neutral, i.e. it should not reduce.

The first restriction is necessary because otherwise reduction would introduce variables that are not in scope, breaking well-scopedness of expressions. Without the second restriction, it would be easy to define rewrite rules that break type preservation.[6] It is possible to go without the third restriction, but in practice this would mean that the rewrite rule would never be applied.

3.3 Reduction and matching

To reduce a term $\mathsf{f}\ \bar{u}$, we look at the rewrite rules with head symbol f to see if any of them apply. In the rule below and all rules in the future, we assume a fixed global signature Σ containing all (preceding) declarations.

$$\frac{(\forall \Delta.\mathsf{f}\ \bar{p} : A \longrightarrow v) \in \Sigma \qquad [\bar{u} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma}{\mathsf{f}\ \bar{u} \longrightarrow v\sigma} \tag{4}$$

Matching a term u against a pattern p $\boxed{[u \mathbin{/\!\!/} p] \Rightarrow \sigma}$ (or $\boxed{[\bar{u} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma}$ for matching a list of terms against a list of patterns) produces — if it succeeds — a substitution σ. In contrast to the first-match semantics of clauses of a regular definition by pattern matching, all rewrite rules are considered in parallel, so there is no need for separate notion of a failing match.

The basic matching algorithm is defined by the rules in Fig. 1. Matching a term against a pattern variable produces a substitution that assigns the given value to the variable. Matching an expression against a pattern $\mathsf{f}\ \bar{p}$ evaluates the expression until it becomes of the form $\mathsf{f}\ \bar{v}$. It then recursively matches the arguments $\bar{v}$ against the patterns $\bar{p}$, combining the results of each match by taking the disjoint union $\sigma_1 \uplus \sigma_2$. Since matching can reduce the term being matched, matching and reduction are mutually recursive.

[6] To prove type preservation we also need confluence of reduction, see the future work section for more details.

$$\frac{}{[u \mathbin{/\!\!/} x] \Rightarrow [u \,/\, x]} \qquad \frac{u \longrightarrow^* \mathsf{f}\ \bar{v} \qquad [\bar{v} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma}{[u \mathbin{/\!\!/} \mathsf{f}\ \bar{p}] \Rightarrow \sigma}$$

$$\frac{}{[\cdot \mathbin{/\!\!/} \cdot] \Rightarrow []} \qquad \frac{[u \mathbin{/\!\!/} p] \Rightarrow \sigma_1 \qquad [\bar{u} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma_2}{[u; \bar{u} \mathbin{/\!\!/} p; \bar{p}] \Rightarrow \sigma_1 \uplus \sigma_2}$$

Figure 1 Basic rules for the matching algorithm used for rewriting.

$$\frac{u \longrightarrow^* \lambda x.\, v \qquad \Phi, x \vdash [v \mathbin{/\!\!/} p] \Rightarrow \sigma}{\Phi \vdash [u \mathbin{/\!\!/} \lambda x.\, p] \Rightarrow \sigma}$$

$$\frac{A \longrightarrow^* (x : B) \to C \qquad \Phi \vdash [B \mathbin{/\!\!/} p] \Rightarrow \sigma_1 \qquad \Phi, x \vdash [C \mathbin{/\!\!/} q] \Rightarrow \sigma_2}{\Phi \vdash [A \mathbin{/\!\!/} (x : p) \to q] \Rightarrow \sigma_1 \uplus \sigma_2}$$

$$\frac{u \longrightarrow^* x\ \bar{v} \qquad x \in \Phi \qquad \Phi \vdash [\bar{v} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma}{\Phi \vdash [u \mathbin{/\!\!/} x\ \bar{p}] \Rightarrow \sigma} \qquad \frac{x \notin \Phi \qquad FV(v) \cap \Phi \subseteq \bar{y}}{\Phi \vdash [v \mathbin{/\!\!/} x\ \bar{y}] \Rightarrow [(\lambda \bar{y}.\, v) \,/\, x]}$$

Figure 2 Rules for higher-order pattern matching.

3.4 Higher-order rewriting

With the basic set of rewrite rules introduced in the previous section, we can already declare a surprisingly large number of rewrite rules for first-order algebraic structures. From the examples in Sect. 2, it handles all of Sect. 2.1, rules map−fuse and map−++ from Sect. 2.2, all of Sect. 2.3, rule //−beta from Sect. 2.4, rules catch−true, catch−false, and catch−exc from Sect. 2.5, and the rules dealing with Bool in Sect. 2.6.

Most of the examples that are not yet handled use λ and/or function types in the pattern of a rewrite rule. This brings us to the issue of *higher-order rewriting*.[7] To support higher-order rewriting, we extend the pattern syntax with the following patterns:

- A lambda pattern $\boxed{\lambda x.\, p}$
- A function type pattern $\boxed{(x : p) \to q}$
- A bound variable pattern $\boxed{y\ \bar{p}}$, where x is a variable bound locally in the pattern by a lambda or function type
- A pattern variable $\boxed{x\ \bar{y}}$ applied to locally bound variables

During matching we must keep the (rigid) bound variables separate from the (flexible) pattern variables. For this purpose, the algorithm keeps a list Φ of all rigid variables. This list is not touched by any of the rules of Fig. 1, but any variables bound by a λ or a function type are added to it.

The extended matching rules for higher-order patterns are given in Fig. 2. Note the strong similarity between the third rule and the rule for matching a function symbol f. This is not a coincidence: both function symbols and bound variables act as rigid symbols that

[7] See also `https://github.com/agda/agda/issues/1563` for more examples where higher-order rewrite rules are needed.

can be matched against. The first three rules in Fig. 2 extend the pattern syntax to allow for bound variables in patterns, and allow for rules such as map − id : map (λ x → x) xs ≡ xs. However, alone they do not yet constitute true higher-order rewriting (such as used in rules raise−fun, cong−Π, and cong−λ). For this we also consider *pattern variables* applied to arguments. Allowing arbitrary patterns as arguments to pattern variables is well known to make matching undecidable, so we restrict patterns to Miller's pattern fragment [15] by requiring pattern variables to be applied to distinct bound variables. Matching against a pattern variable in the Miller fragment is implemented by the fourth rule in Fig. 2. Since all the arguments of x are variables, we can construct the lambda term $\lambda\bar{y}.\, v$. To avoid having out-of-scope variables in the resulting substitution, the free variables in v are checked to be included in $\bar{y}$, otherwise matching fails.

3.5 η-equality

The attentive reader may have noticed a flaw in the matching for λ-patterns: it does not respect η-equality. With η-equality for functions, any term $u : (x : A) \to B\ x$ can always be expanded to $\lambda x.\, u\ x$, so it should also match a pattern $\lambda x.\, p$. A naive attempt to add η-equality would be to η-expand on the fly whenever we match something against a λ-pattern:

$$\frac{\Phi, x \vdash [u\ x \mathbin{/\!\!/} p] \Rightarrow \sigma}{\Phi \vdash [u \mathbin{/\!\!/} \lambda x.\, p] \Rightarrow \sigma} \tag{5}$$

This is however not enough to deal with η-equality in general. It is possible that the pattern itself is underapplied as well, e.g. when we match a term of type $(x : A) \to B\ x$ against a pattern f $\bar{p}$ or $x\ \bar{p}$.

To respect eta equality for functions and record types, we need to make matching *type-directed*. We also need contexts with the types of the free and bound variables. Thus we extend the matching judgement to $\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} p] \Rightarrow \sigma$ where A is the type of u (note: not necessarily the same as the type of p) and Γ and Φ are now contexts of pattern variables and bound variables respectively.

The type information is used by the matching algorithm to do on-the-fly η-expansion of functions whenever the type is (or computes to) a function type:

$$\frac{A \longrightarrow^* (x : B) \to C \qquad \Gamma; \Phi(x : B) \vdash [u\ x : C \mathbin{/\!\!/} p\ x] \Rightarrow \sigma}{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} p] \Rightarrow \sigma} \tag{6}$$

Here $p\ x$ is only defined if the result is actually a pattern, otherwise the rule cannot be applied.

Having access to the type of the expression being matched is not only useful for η-equality of functions, but also for non-linear patterns (Sect. 3.6), η-equality for records Sect. 4.1), and irrelevance (Sect. 4.4).

3.6 Non-linearity and non-patterns

Sometimes it is desirable to declare rewrite rules with *non-linear* patterns, i.e. where a pattern variable occurs more than once. As an example, this allows us to postulate an equality proof trustMe : $(x\ y : A) \to x \equiv y$ with a rewrite rule trustMe $x\ x \equiv$ refl. This can be

used in a similar way to Agda's built-in primTrustMe[8]. Another example where non-linearity is used is the rule transportR−refl from example 4.[9]

Non-linear matching is actually an instance of a more general concept I call a *non-pattern*. A non-pattern is an arbitrary term that matches another term precisely when the terms are definitionally equal. Non-patterns allow embedding of arbitrary terms inside a pattern, but they cannot bind any variables. So even though we allow non-patterns, each pattern variable used in the rewrite rule still has to occur at least once in a pattern position.

Non-patterns are similar to inaccessible patterns (aka dot patterns in Agda) used in dependent pattern matching, with the important difference that inaccessible patterns are assumed to match whenever the rest of the pattern does, while non-patterns have to be *checked*.

For both non-linear patterns and non-patterns, the matching algorithm needs to decide whether two given terms are definitionally equal. This means reduction and matching are now mutually recursive with conversion checking.[10]

We make use of a type-directed conversion judgement $\Gamma \vdash u = v : A$ (see the appendix for the full conversion rules). The new judgement form of matching is now $\Gamma; \Phi \vdash [v : A \mathbin{/\!\!/} p] \Rightarrow \sigma; \Psi$, where Ψ is a set of constraints of the form $\Phi \vdash u \stackrel{?}{=} v$. We extend the matching algorithm with the ability to postpone a matching problem:

$$\frac{}{\Gamma; \Phi \vdash [v : A \mathbin{/\!\!/} p] \Rightarrow []; \{\Phi \vdash v \stackrel{?}{=} p : A\}} \tag{7}$$

All other rules just gather the set of constraints, taking the union whenever matching produces multiple sub-problems. When matching concludes, the constraints are checked before the rewrite rule is applied:

$$\frac{\mathsf{f} : \Gamma \to A \in \Sigma \qquad (\forall \Delta.\mathsf{f}\ \bar{p} : B \longrightarrow v) \in \Sigma \qquad [\bar{u} : \Gamma[\bar{u}] \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma; \Psi \qquad \forall (\Phi \vdash v \stackrel{?}{=} p : A) \in \Psi.\ \Phi \vdash v = p\sigma : A}{\mathsf{f}\ \bar{u} \longrightarrow v\sigma} \tag{8}$$

When checking a constraint we apply the final substitution σ to the pattern p but not to the term v or the type A. This makes sense because the term being matched does not contain any pattern variables in the first place (and neither does its type).

4 Interaction with other features

Adding rewrite rules to an existing language such as Agda is quite an undertaking. Rewrite rules often interact with other features in a non-trivial matter, and it takes work to resolve these interactions in a satisfactory way. In this section, I describe the interaction of rewrite rules with several other features of Agda: record types with eta equality, data-types, parametrized modules, definitional irrelevance, universe polymorphism, and constraint solving.

[8] `https://agda.readthedocs.io/en/v2.6.0.1/language/built-ins.html#primtrustme`

[9] It also needs irrelevance for Prop, see Sect. 4.4 for more details.

[10] To actually change the implementation of Agda to make the matching depend on conversion checking took quite some effort (see `https://github.com/agda/agda/pull/3589`). The reason for this difficulty was that reduction and matching are running in one monad `ReduceM`, while conversion was running in another monad `TCM` (short for 'type-checking monad'). The new version of the conversion checker is polymorphic in the monad it runs in. This means the same piece of code implements at the same time a pure, declarative conversion checker and a stateful constraint solver.

4.1 η-equality for records

Agda has η-equality not just for function types, but also for record types. For example, any term $u : A \times B$ is definitionally equal to $(\mathsf{fst}\ u, \mathsf{snd}\ u)$. Since η-equality of records is a core part of Agda, we extend the matching algorithm to deal with it.[11] As for η-equality of functions, we make use of the type of the expression to η-expand terms and patterns during matching.

Let $\mathsf{R} : \mathsf{Set}_i$ be a record type with fields $\pi_1 : A_1$, …, $\pi_n : A_n$. We have the following matching rule:

$$\frac{\Gamma; \Phi \vdash [\pi_i\ u : A_i[\pi_1\ u\,/\,\pi_1, \cdots, \pi_{i-1}\ u\,/\,\pi_{i-1}] \mathbin{/\!\!/} \pi_i\ p] \Rightarrow \sigma \quad (i = 1 \cdots n)}{\Gamma; \Phi \vdash [u : \mathsf{R} \mathbin{/\!\!/} p] \Rightarrow \sigma} \tag{9}$$

Since records can be dependent, each type A_i may depend on the previous fields $\pi_1, \ldots, \pi_{i-1}$, so we need to substitute the concrete values $\pi_j\ u$ for π_j in A_i for each $j < i$.

In the case where $n = 0$, this rule says that a term of the unit record type $\top$ (with no fields) matches any pattern. So the matching algorithm even handles the notorious η-unit types.

4.2 Datatypes and constructors

An important question is how rewrite rules interact with datatypes such as Nat, List, and $\_\equiv\_$. Can we simply add rewrite rules to (type and/or term) constructors? The answer is actually a bit more complicated.

If we allow rewriting of datatype constructors, we could (for example) postulate an equality proof of type $\mathsf{Nat} \equiv \mathsf{Bool}$ and register it as a rewrite rule. However, this would mean $\mathsf{zero} : \mathsf{Bool}$, violating an important internal invariant of Agda that any time we have $\mathsf{c}\ \bar{u} : \mathsf{D}$ for a constructor c and a datatype D, c is actually a constructor of D.[12] For this reason, it is not allowed to have rewrite rules on datatypes or record types.

For constructors of datatypes there is no a priori reason why they cannot have rewrite rules attached to them. This would actually be useful to define a 'definitional quotient type' where some of the constructors may compute. Unfortunately, there is another problem: internally, Agda does not store the constructor arguments corresponding to the parameters of the datatype. For example, the constructors [] and $\_::\_$ of the $\mathsf{List}\ A$ type do not store the type A as an argument. This is important for efficient representation of parametrized datatypes. However, this means that rewrite rules that match on a constructor on constructors cannot match against arguments in those positions, or bind pattern variables in them.

When a rewrite rule is added with a constructor as the head symbol, we have to take care that the rewrite rule is not applied too generally. For example, a rewrite rule for $[] : \mathsf{List\ Nat}$ should not be applied to $[] : \mathsf{List}\ A$ where $A \neq \mathsf{Nat}$[13]. To avoid unwanted reductions like these, it is only allowed to add a rewrite rule to a constructor if the parameters are *fully general*, i.e. they must be distinct variables. This ensures that rewrite rules are only applied to terms whose type matches the type of the rewrite rule.

[11] See `https://github.com/agda/agda/issues/2979` and `https://github.com/agda/agda/issues/3335`.
[12] See `https://github.com/agda/agda/issues/3846`.
[13] See `https://github.com/agda/agda/issues/3211`.

4.3 Parametrized modules and where blocks

A parametrized module is a collection of declarations parametrized over a common telescope Γ. In one sense, parametrized modules can be thought of as λ-lifting all the definitions inside the module: if a module with parameters Γ contains a definition of $\mathsf{f} : A$, then the real type of f is $\Gamma \to A$. But this does not quite capture the intuition that definitions inside a parametrized module should be *parametric* in the parameters. So module parameters should be treated as rigid symbols like postulates rather than as flexible variables.

For this reason, module parameters play a double role on the left-hand side of a rewrite rule:

- As long as the parameter is in scope (i.e. inside the module), it is treated as a non-pattern, so it has to match 'on the nose' (i.e. it cannot be instantiated by matching).
- Once the parameter goes out of scope (i.e. outside of the module), it is treated as a regular pattern variable that can be instantiated by matching.

For example, inside a module parametrized over $n : \mathsf{Nat}$, a rewrite rule $\mathsf{f}\ n \longrightarrow \mathsf{zero}$ only applies to terms definitionally equal to $\mathsf{f}\ n$. On the other hand, outside of the module the rewrite rule applies to any expression of the form $\mathsf{f}\ u$.

This intuition of module parameters as rigid symbols also applies to Agda's treatment of where blocks, which are nothing more than modules parametrized over the pattern variables of the clause (you can even give a name to the where module using the module M where syntax[14]). Here a 'local' rewrite rule in a where block should only apply for the specific arguments to the function that are used in the clause, not those of a recursive call[15].

4.4 Irrelevance and Prop

Another feature of Agda is *definitional irrelevance*, which comes in the two flavours of irrelevant function types $.A \to B$[16] and the universe Prop of definitionally proof-irrelevant propositions[17]. For rewrite rules with irrelevant parts in their patterns matching should never fail because this would mean a supposedly irrelevant term is not actually irrelevant. However, it should still be allowed to bind a variable in an irrelevant position, since we might want to use that variable in (irrelevant positions of) the right-hand side.[18] This means in irrelevant positions we allow:

1. pattern variables $x\ \bar{y}$ where $\bar{y}$ are all the bound variables in scope, and
2. non-patterns u that do not bind any variables.

Both of these will always match any given term, but only the former binds a variable.

Together with the ability to have non-linear patterns, this allows us to have rewrite rules such as $\mathsf{transportR-refl} : \mathsf{transportR}\ P\ \mathsf{refl}\ x \equiv x$ where $\mathsf{transportR} : (P : A \to \mathsf{Set}_\ell) \to x \doteq y \to P\ x \to P\ y$ and $x \doteq y$ is the equality type in Prop. The constructor refl here is irrelevant, so this rule does not actually match against the constructor refl. Instead, Agda checks that the two arguments x and y are definitionally equal, and applies the rewrite rule if this is the case.

[14] https://agda.readthedocs.io/en/v2.6.0.1/language/let-and-where.html#where-blocks
[15] https://github.com/agda/agda/issues/1652
[16] https://agda.readthedocs.io/en/v2.6.0.1/language/irrelevance.html
[17] https://agda.readthedocs.io/en/v2.6.0.1/language/prop.html
[18] See https://github.com/agda/agda/issues/2300.

4.5 Universe level polymorphism

Universe level polymorphism allows Agda programmers to write definitions that are polymorphic in the universe level of a type parameter. Since the type Level of universe levels is a first-class type in Agda, it interacts natively with rewrite rules: patterns can bind variables of type Level just as any other type. This allows us for example to define rewrite rules such as map − id that work on level-polymorphic lists.

The type Level supports two operations lsuc : Level → Level and _⊔_ : Level → Level → Level. These operations have a complex equational structure: _⊔_ is associative, commutative, and idempotent, and lsuc distributes over _⊔_, just to name a few of the laws. This causes trouble when a rewrite rule matches against one of these symbols: how should it determine whether a given level matches $a \sqcup b$ when _⊔_ is commutative?[19] For this reason it is not allowed to have rewrite rules that match against lsuc or _⊔_ (i.e. expressions containing these symbols are treated as non-patterns).

This restriction on patterns of type Level is seems reasonable enough, but it is often not satisfied by rewrite rules that match on function types — like the cong−Π rule we used in the encoding of observational type theory (Sect. 2.6). The problem is that if $A : \mathsf{Set}_{\ell_1}$ and $B : \mathsf{Set}_{\ell_2}$, then the function type $(x : A) \to B$ has type $\mathsf{Set}_{\ell_1 \sqcup \ell_2}$, so there is no sensible position to bind the variables ℓ_1 and ℓ_2.

To allow rewrite rules such as cong−Π, we need to find a different position where these variables of type Level can be bound. In the internal syntax of Agda, function types $(x : A) \to B$ are annotated with the sorts of A and B. So the 'real' function type of Agda is of the form $(x : A : \mathsf{Set}_{\ell_1}) \to (B : \mathsf{Set}_{\ell_2})$. This means that if we allow rewrite rules to bind pattern variables in these hidden annotations, we are saved.[20] The matching rule for function types now becomes:

$$\frac{\begin{array}{c}\Gamma;\Phi \vdash [A : \mathsf{Set}\ \ell_1 \mathbin{/\!/} p] \Rightarrow \sigma_1;\Psi_1 \qquad \Gamma;\Phi \vdash [\ell_1 : \mathsf{Level} \mathbin{/\!/} q] \Rightarrow \sigma_2;\Psi_2 \\ \Gamma;\Phi(x : A) \vdash [B : \mathsf{Set}\ \ell_2 \mathbin{/\!/} r] \Rightarrow \sigma_3;\Psi_3 \qquad \Gamma;\Phi \vdash [\ell_2 : \mathsf{Level} \mathbin{/\!/} s] \Rightarrow \sigma_4;\Psi_4\end{array}}{\begin{array}{c}\Gamma;\Phi \vdash [(x : A : \mathsf{Set}\ \ell_1) \to (B : \mathsf{Set}\ \ell_2) \mathbin{/\!/} (x : p : \mathsf{Set}_q) \to (r : \mathsf{Set}_s)] \\ \Rightarrow (\sigma_1 \uplus \sigma_2 \uplus \sigma_3 \uplus \sigma_4);(\Psi_1 \cup \Psi_2 \cup \Psi_3 \cup \Psi_4)\end{array}} \tag{10}$$

Thanks to this rule, also the universe-polymorphic version of the rewrite rules in Sect. 2.6 are accepted by Agda.

4.6 Metavariables and constraint solving

To automatically fill in the values of implicit arguments, Agda inserts *metavariables* as their placeholders. These metavariables are then solved during typechecking by the constraint solver. A full description of Agda's constraint solver is out of the scope of this paper, but let me discuss the most important ways it is impacted by rewrite rules.

4.6.1 Blocking tags

The constraint solver needs to know when a reduction is blocked on a particular metavariable. Usually it is possible to point out a single metavariable, but this is no longer the case when rewrite rules are involved:

[19] Issue #2090 (`https://github.com/agda/agda/issues/2090`) and issue #2299 (`https://github.com/agda/agda/issues/2299`) show some of the things that would go wrong.
[20] See also `https://github.com/agda/agda/issues/3971`.

- With overlapping rewrite rules, reduction can be blocked on a set of metavariables. For example, if we try to reduce the expression $X + Y$ where X and Y are metavariables of type Nat and _+_ is defined with the rewrite rules from Sect. 2.1, then this expression might reduce further when either X or Y is instantiated to a constructor. So a postponed constraint involving this expression has to be woken up when either metavariable is instantiated.
- For higher-order matching, matching checks whether a particular variable occurs freely in the body of a lambda or pi. When metavariables are involved, a variable occurrence may be *flexible*: whether or not the variable occurs depends on the instantiation of a particular metavariable[21]. In this case reduction is blocked on the set of all metavariables with potentially unbound variables in their arguments.
- When a rewrite rule with non-linear patterns or non-patterns is blocked on the conversion check because of an unsolved metavariable, reduction can be blocked on the metavariable that is preventing the conversion check from succeeding.[22][23]

Currently the Agda implementation uses only an approximation of the set of metavariables it encounters, i.e. only the first metavariable encountered. This is harmless because the current implementation of Agda will eventually try again to solve all postponed constraints, even ianyway quite generous in how often it retries to solve sleeping constraints. If in the future Agda would be changed to be more careful in letting sleeping constraints lie, a more precise tracking of blocking metavariables would also be desirable.

4.6.2 Pruning and constructor-like symbols

When adding new rewrite rules, we also keep track of what symbols are *constructor-like*. This is important for the pruning phase of the constraint solver. For example, let us consider a constraint $X \stackrel{?}{=} Y\ (\mathsf{f}\ x)$. Since the metavariable X does not depend on the variable x, the constraint solver attempts to *prune* the dependency of Y on x. If f is a regular postulate without any rewrite rules, there is no way that Y could depend on f x without also depending on x, so the dependency of Y on its first argument is pruned away. However, if there is a rewrite rule where f plays the role of a constructor — say a rule g (f y) ⟶ true — then the assignment $X := \mathsf{true}$ and $Y := \lambda y.\, \mathsf{g}\ y$ is a valid solution to the constraint where Y *does* depend on its argument, so it should **not** be pruned away. In general, an argument should not be pruned if the head symbol is constructor-like, i.e. if there is at least one rewrite rule that matches against the symbol.

5 Related work

Our extension of dependent type theory with rewrite rules resembles in many ways the Dedukti system [11, 9, 5]. Both systems support dependent types and higher-order non-linear rewrite rules. There are however some important differences:

- Dedukti was built up from the ground based on rewrite rules. In contrast, we start from a general dependently typed language (Agda) and extend it with rewrite rules.
- Dedukti is based on the Logical Framework (LF), while our language is build from Martin-Löf's intuitionistic type theory.

[21] https://github.com/agda/agda/issues/1663
[22] https://github.com/agda/agda/issues/1987
[23] https://github.com/agda/agda/issues/2302

- Dedukti has universes à la Tarski: a universe is a set of codes that can be interpreted as types by an interpretation function. In contrast, Agda uses universes à la Russell: elements of a universe *are* types without need of an interpretation function.
- Dedukti uses an untyped conversion algorithm, while Agda uses a typed one. Hence we can support η-equality for functions and record types, which is not possible (directly) in Dedukti.
- Dedukti provides external tools for checking confluence and termination of the rewrite system given by the user. We do not yet provide such checks, although we plan to add them in the future.[24]

The Calculus of Algebraic Constructions (CAC) [7] extends the Calculus of Constructions with functions and predicates defined by higher-order rewrite rules. Compared to our implementation of rewrite rules, CAC is more limited in that it only allows for decidable theories, but it provides criteria for checking subject reduction and strong normalization of the rewrite rules.

Coq modulo theory (CoqMT) [18, 6, 13] also extends the Coq proof assistant with a decidable theory. The equational theory in CoqMTU must be first-order, but can include equational rules such as commutativity, which cannot be expressed as rewrite rules. CoqMTU also provides strong guarantees for confluence, subject reduction, strong normalization, and consistency of the theory. Unfortunately, the implementation of CoqMT[25] has not been updated to work with the current version of Coq.

The Zombie language [17] is another dependently typed language where definitional equality can be extended with user-provided equations that are applied automatically by the typechecker. Instead of rewrite rules, Zombie computes the congruence closure of the given equations and uses this during conversion checking. An important difference with our approach is that the definitional equality in Zombie does not include β-equality, which makes it easier to extend it in other directions. The congruence closure algorithm used by Zombie is untyped, which means it cannot handle η-equality of functions or records. It also does not include true higher-order rules.

6 Future work

Safe(r) rewrite rules

This paper is about how to add rewrite rules to Agda or similar languages. By their design rewrite rules are a very unsafe feature of Agda. Compared to using postulate, rewrite rules do not break logical soundness of the theory, but they can break core assumptions of Agda such as confluence of reduction and even type preservation. So using rewrite rules is like building your own type theory, which means you have to do your own meta-theory to make sure everything is safe.

Ideally, Agda would be able to detect if a given set of rewrite rules is 'safe', in the sense that they do not break the usual properties of Agda programs such as subject reduction and decidable typechecking. The development version of Agda 2.6.1 includes an experimental flag `--confluence-check`, which checks the *local* confluence of the declared rewrite rules. For checking termination, we could make use of the dependency pairs criterion as done by

[24] The development version of Agda already includes an experimental flag `--confluence-check`, checking *local* confluence of rewrite rules.
[25] `https://github.com/strub/coqmt`

SizeChangeTool for Dedukti [8]. Combining the results of these checks would allow us to prove injectivity of Π types, and hence subject reduction of our type theory.

Local rewrite rules

When programming in a dependently typed language, we rely on terms computing to their values. However, this fails when we work with abstract values (e.g. module parameters): until they are instantiated, they are opaque symbols without any computational behaviour. This actively encourages users to work with concrete values and discourages abstraction.

To improve this situation, we could allow *local* rewrite rules on module parameters to be added to the context. For example, we could parametrize a module over a value $\emptyset$ and a binary operation $\_ \cdot \_$ together with rewrite rules $\emptyset \cdot y \longrightarrow y$ and $x \cdot \emptyset \longrightarrow x$. When instantiating the module parameters, we have to check that the rewrite rules are indeed satisfied by the given values.

Having local rewrite rules greatly complicates checking of confluence and termination. So the future will have to point out if there is a reasonable way to allow local rewrite rules while maintaining subject reduction of the language.

Custom η rules

Rewrite rules allow us to add custom β rules to our type theory, but it would be useful to also allow custom η rules. This would for example allow us to add η-rules for datatypes such as Vec, making any vector of length zero definitionally equal to [].

Where rewrite rules allow extending the reduction relation of the theory, custom η rules would allow extending the conversion checker directly. Since conversion in Agda is type-directed, it would make sense to allow custom η rules that match against the type of a constraint. Thus much of the matching algorithm in this paper could be reused for η rules.

7 Conclusion

This paper documents the process of integrating user-defined rewrite rules into a general-purpose dependently typed language, and all the weird interactions that I encountered along the way. Rewrite rules allow you to extend the power of a dependently typed language on a much deeper level than normally allowed. They can be used as a convenient feature to make more terms typecheck without using explicit rewrite statements, and they allow advanced users to experiment with new evaluation rules, without actually modifying the typechecker. If you are an Agda user, I hope reading this paper has given you a deeper understanding of rewrite rules and allows you to harness their power responsibly. And if you are implementing your own dependently typed language, I hope you consider adding rewrite rules as way to make it both easier to use and more extensible.

References

1 *Homotopy Type Theory - Univalent Foundations of Mathematics: The Univalent Foundations Program*. Institute for Advanced Study, 2013. URL: `https://homotopytypetheory.org/book/`.

2 Agda development team. *Agda 2.6.0.1 documentation*, 2019. URL: `http://agda.readthedocs.io/en/v2.6.0.1/`.

3 Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In Stephanie Weirich, editor, *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP*

2013, Boston, Massachusetts, USA, September 24, 2013, pages 13–24. ACM, 2013. doi:10.1145/2502409.2502411.

4 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.

5 Ali Assaf and Guillaume Burel. Translating HOL to dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015.*, volume 186 of *EPTCS*, pages 74–88, 2015. doi:10.4204/EPTCS.186.8.

6 Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub, and Qian Wang. CoQMTU: A higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 143–151. IEEE Computer Society, 2011. doi:10.1109/LICS.2011.37.

7 Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005. doi:10.1017/S0960129504004426.

8 Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany.*, volume 131 of *LIPIcs*, pages 9:1–9:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.9.

9 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The $\lambda\pi$-calculus modulo as a universal proof language. In *the Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, 2012.

10 Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns - definitional equality for all. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2014. doi:10.1007/978-3-642-54833-8_6.

11 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007. doi:10.1007/978-3-540-73228-0_9.

12 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *PACMPL*, 3(POPL):3:1–3:28, 2019. doi:10.1145/3290316.

13 Jean-Pierre Jouannaud and Pierre-Yves Strub. Coq without type casts: A complete proof of coq modulo theory. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 474–489. EasyChair, 2017. URL: http://www.easychair.org/publications/paper/340342.

14 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

15 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991. doi:10.1093/logcom/1.4.497.

16 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option - an exceptional type theory. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, Lecture Notes in Computer Science, pages 245–271. Springer, 2018. doi:10.1007/978-3-319-89884-1_9.

17 Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 369–382. ACM, 2015. doi:10.1145/2676726.2676974.

18 Pierre-Yves Strub. Coq Modulo Theory. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2010. doi:10.1007/978-3-642-15205-4_40.

A Complete rules of type theory with user-defined rewrite rules

A.1 Syntax

Terms $\boxed{u, v, w, A, B, C, p, q}$

$$
\begin{array}{rcll}
u, v, w, A, B, C, p, q & ::= & x\ \bar{u} & \text{(variable applied to arguments)} \\
& | & \mathsf{f}\ \bar{u} & \text{(function symbol applied to arguments)} \\
& | & \lambda x.\, u & \text{(lambda abstraction)} \\
& | & (x : A) \to B & \text{(dependent function type)} \\
& | & \mathsf{Set}_i & (i\text{th universe})
\end{array}
$$

Substitutions Substitutions $\boxed{\sigma}$ are lists of variable-term pairs $[u_1 \,/\, x_1, \ldots, u_n \,/\, x_n]$. Application of a substitution to a term $\boxed{u\sigma}$ is defined as usual, avoiding variable capture by α-renaming where necessary.

Application Application $\boxed{u\ v}$ is a partial operation on terms and is defined as follows:

$$
\begin{aligned}
(x\ \bar{u})\ v &= x\ (\bar{u}; v) \\
(\mathsf{f}\ \bar{u})\ v &= \mathsf{f}\ (\bar{u}; v) \\
(\lambda x.\, u)\ v &= u[v \,/\, x]
\end{aligned}
$$

Contexts $\boxed{\Gamma, \Delta, \Phi, \Xi}$

$$
\begin{array}{rcll}
\Gamma, \Delta, \Phi, \Xi & ::= & \cdot & \text{(empty context)} \\
& | & \Gamma(x : A) & \text{(context extension)}
\end{array}
$$

Declarations $\boxed{d}$

$$
\begin{array}{rcll}
d & ::= & \mathsf{f} : A & \text{(function symbol)} \\
& | & \forall \Delta.\, \mathsf{f}\ \bar{p} : A \longrightarrow v & \text{(rewrite rule)}
\end{array}
$$

A.2 Typing rules

We assume a global signature Σ containing declarations and rewrite rules, which is implicit in all the judgements.

Typing $\boxed{\Gamma \vdash u : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\mathsf{f} : A \in \Sigma}{\Gamma \vdash \mathsf{f} : A} \qquad \frac{\Gamma (x : A) \vdash u : B}{\Gamma \vdash \lambda x.\, u : (x : A) \to B} \qquad \frac{\Gamma \vdash u : (x : A) \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B[v \,/\, x]}$$

$$\frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma (x : A) \vdash B : \mathsf{Set}_j}{\Gamma \vdash (x : A) \to B : \mathsf{Set}_{i \sqcup j}} \qquad \frac{}{\mathsf{Set}_i : \mathsf{Set}_{1+i}} \qquad \frac{\Gamma \vdash A = B : \mathsf{Set}_i \qquad \Gamma \vdash u : A}{\Gamma \vdash u : B}$$

Conversion $\boxed{\Gamma \vdash u = v : A}$

$$\frac{\Gamma \vdash u \longrightarrow u' \qquad \Gamma \vdash u' = v : A}{\Gamma \vdash u = v : A} \qquad \frac{\Gamma \vdash v \longrightarrow v' \qquad \Gamma \vdash u = v' : A}{\Gamma \vdash u = v : A} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x = x : A}$$

$$\frac{\mathsf{f} : A \in \Sigma}{\Gamma \vdash \mathsf{f} = \mathsf{f} : A} \qquad \frac{\Gamma (x : A) \vdash u\ x = v\ x : B}{\Gamma \vdash u = v : (x : A) \to B} \qquad \frac{\Gamma \vdash u_1 = u_2 : (x : A) \to B \qquad \Gamma \vdash v_1 = v_2 : A}{\Gamma \vdash u_1\ v_1 = u_2\ v_2 : B[v_1 \,/\, x]}$$

$$\frac{\Gamma \vdash A_1 = A_2 : \mathsf{Set}_i \qquad \Gamma (x : A_1) \vdash B_1 = B_2 : \mathsf{Set}_j}{\Gamma \vdash (x : A_1) \to B_1 = (x : A_2) \to B_2 : \mathsf{Set}_{i \sqcup j}}$$

Reduction $\boxed{\Gamma \vdash u \longrightarrow v}$

$$\frac{\begin{array}{c}\mathsf{f} : B \in \Sigma \qquad (\forall \Xi.\mathsf{f}\ \bar{p} : C \longrightarrow v) \in \Sigma \\ \Gamma \Xi; \cdot \vdash [(\bullet : B)\ \bar{u} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma; \Psi \qquad \forall (\Phi \vdash v \stackrel{?}{=} p : A) \in \Psi.\ \Gamma \Phi \vdash v = p\sigma : A\end{array}}{\Gamma \vdash \mathsf{f}\ \bar{u} \longrightarrow v\sigma}$$

Matching $\boxed{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} p] \Rightarrow \sigma; \Psi}$

$$\frac{x : B \in \Gamma}{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} x] \Rightarrow [u \,/\, x]; \emptyset} \qquad \frac{}{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} v] \Rightarrow []; \{\Phi \vdash u \stackrel{?}{=} v : A\}}$$

$$\frac{\Gamma \Phi \vdash u \longrightarrow^* \mathsf{f}\ \bar{v} \qquad \mathsf{f} : B \in \Sigma \qquad \Gamma; \Phi \vdash [(\bullet : B)\ \bar{v} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma; \Psi}{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} \mathsf{f}\ \bar{p}] \Rightarrow \sigma; \Psi}$$

$$\frac{\Gamma \Phi \vdash u \longrightarrow^* x\ \bar{v} \qquad x : B \in \Phi \qquad \Gamma; \Phi \vdash [(\bullet : B)\ \bar{v} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma; \Psi}{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} x\ \bar{p}] \Rightarrow \sigma; \Psi}$$

$$\frac{\Gamma \Phi \vdash A \longrightarrow^* (x : B) \to C \qquad \Gamma; \Phi (x : B) \vdash [u\ x : C \mathbin{/\!\!/} p\ x] \Rightarrow \sigma; \Psi}{\Gamma; \Phi \vdash [u : A \mathbin{/\!\!/} p] \Rightarrow \sigma; \Psi}$$

$$\frac{\Gamma \Phi \vdash A \longrightarrow^* (x : B) \to C \qquad \Gamma; \Phi \vdash [B \mathbin{/\!\!/} p] \Rightarrow \sigma_1; \Psi_1 \qquad \Gamma; \Phi (x : B) \vdash [C \mathbin{/\!\!/} q] \Rightarrow \sigma_2; \Psi_2}{\Gamma; \Phi \vdash [A : D \mathbin{/\!\!/} (x : p) \to q] \Rightarrow \sigma_1 \uplus \sigma_2; \Psi_1 \cup \Psi_2}$$

Spine matching $\boxed{\Gamma; \Phi \vdash [(\bullet : A)\ \bar{u} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma; \Psi}$

$$\frac{}{\Gamma; \Phi \vdash [(\bullet : A)\ \cdot \mathbin{/\!\!/} \cdot] \Rightarrow []; \emptyset} \qquad \frac{\begin{array}{c}\Gamma \Phi \vdash A \longrightarrow^* (x : B) \to C \qquad \Gamma; \Phi \vdash [u : B \mathbin{/\!\!/} p] \Rightarrow \sigma_1; \Psi_1 \\ \Gamma; \Phi \vdash [(\bullet : C[u \,/\, x])\ \bar{u} \mathbin{/\!\!/} \bar{p}] \Rightarrow \sigma_2; \Psi_2\end{array}}{\Gamma; \Phi \vdash [(\bullet : A)\ u; \bar{u} \mathbin{/\!\!/} p; \bar{p}] \Rightarrow \sigma_1 \uplus \sigma_2; \Psi_1 \cup \Psi_2}$$

A.3 Checking declarations

A declaration of a function symbol $\mathsf{f} : A$ is valid if $\Gamma \vdash A : \mathsf{Set}_i$. A declaration of a rewrite rule $\forall \Delta.\, \mathsf{f}\ \bar{p} : A \longrightarrow v$ is valid if:

- Each variable in Δ occurs at least once in a pattern position in $\bar{p}$.
- $\Delta \vdash \mathsf{f}\ p : A$ and $\Delta \vdash v : A$
- There is no term w such that $\Delta \vdash \mathsf{f}\ \bar{p} \longrightarrow w$.