Extending a Type Theory with Rewrite Rules

Stefan Malewski

01-12-2022

Example: Commutativity of addition

```
_+_ : ℕ → ℕ → ℕ
zero + n     = n
(suc m) + n = suc (m + n)
```

```
comm : (m n : ℕ) → m + n ≡ n + m
comm zero n    = refl                -- zero + n ≡ n + zero
comm (suc m) n = cong suc (comm m n) -- (suc m) + n ≡ n + suc m
```

Example: Commutativity of addition

```
_+_ : ℕ → ℕ → ℕ
zero + n    = n
(suc m) + n = suc (m + n)
```

```
comm : (m n : ℕ) → m + n ≡ n + m
comm zero n    = refl                 -- n ≢ n + zero
comm (suc m) n = cong suc (comm m n) -- suc (m + n) ≢ n + suc m
```

Example: Commutativity of addition

```
+comm-zero : (m : ℕ) → m + zero ≡ zero + m
+comm-suc : (m n : ℕ) → m + (suc n) ≡ (suc n) + m

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

comm : (m n : ℕ) → m + n ≡ n + m
comm zero n    = refl
comm (suc m) n = cong suc (comm m n)
```

Example: Exceptional Type Theory

```
hd : {A : Set} → List A → Maybe A
hd []        = nothing
hd (x :: xs) = just x

hd : {A : Set} (l : List A) → len l > 0 → A
hd []  ()
hd (x :: xs) _ = x
```

```
hd : {A : Set} → List A → A
hd []        = ???
hd (x :: xs) = x
```

Example: Exceptional Type Theory

```
postulate
  -- Type for exceptions
  Exc : Set

  -- Some exception
  errorEmptyList : Exc

  -- How to raise an exception
  raise : {A : Set} (exc : Exc) → A
```

Example: Exceptional Type Theory

```
postulate
  -- New induction principle for List
  catch-List : (A : Set) (P : List A → Set)
               (Pnil : P [])
               (Pcons : (a : A) (l : List A) → P l → P (a :: l))
               (Pexc : ∀ e → P (raise e))
             → (l : List a) → P l
  -- The usual cases
  catch-nil  : ∀ (A : Set) (P : List A → Set) Pnil Pcons Pexc
             → catch-List A P Pnil Pcons Pexc [] ≡ Pnil
  catch-cons : ∀ (A : Set) (P : List A → Set) Pnil Pcons Pexc a l
             → catch-List A P Pnil Pcons Pexc (cons a l)
             ≡ Pcons a l (catch-List A P Pnil Pcons Pexc l)
  -- How to handle exceptions
  catch-exc   : ∀ (A : Set) (P : List A → Set) Pnil Pcons Pexc e
              → catch-List A P Pnil Pcons Pexc (raise e) ≡ Pexc e

{-# REWRITE catch-nil catch-cons catch-exc #-}
```

Example: Exceptional Type Theory

Now, our head function looks like this[1]:

```
hd : {A : Set} → List A → A
hd []          = raise errorEmptyList
hd (x :: xs) = x
hd (raise e) = raise e
```

```
head [] ≡ raise errorEmptyList
head 1 :: raise e ≡ 1
```

[1] not really :(

Rewrite rules can be used to...

Add computation rules to existing definitions

```
m + zero ⇝ m
m + (suc n) ⇝ suc (m + n)
```

Define new primitives that compute

Exceptions, gradual types, observational equality, quotient types, etc...

The shape of rewrite rules

$$\underbrace{?x\ ?y\ ?z}_{\text{pattern variables}} \vdash \underbrace{f\ p_1\ \ldots\ p_n}_{\text{lhs}} \rightharpoonup \text{rhs}$$

$$\begin{array}{rcl} p & ::= & ?x \\ & | & C\ p_1\ \ldots\ p_n \end{array}$$

1. Pattern variables must be left-linear (i.e. appear exactly once in the lhs).
2. f must be fresh.

Rewrite rules: Do they break things? What do they brake? Let's find out!

1. Logical consistency.
2. Decidable typechecking.
3. Type safety.

Logical consistency

Adding the rewrite rule $0 \rightarrowtriangle 1$ breaks consistency, but...

Logical consistency

Adding the rewrite rule $0 \rightarrowtriangle 1$ breaks consistency, but...
... so does adding $0 = 1$ as an axiom.

Theorem

If for each rewrite rule $l \rightarrowtriangle r$ we have a proof that $\vdash e : l = r$, then the system is consistent.

Decidable typechecking

Each rewrite rule $l \rightarrowtriangle r$ is an application of the reflection rule:

$$\frac{\Gamma \vdash e : l\sigma =_A r\sigma}{\Gamma \vdash l\sigma = r\sigma}$$

This breaks decidable type checking in general, but...

Decidable typechecking

Each rewrite rule $l \rightharpoonup r$ is an application of the reflection rule:

$$\frac{\Gamma \vdash e : l\sigma =_A r\sigma}{\Gamma \vdash l\sigma = r\sigma}$$

This breaks decidable type checking in general, but...
... for confluent systems the usual algorithm is still correct if it terminates.
If the system is strongly normalizing, we have decidable typechecking.

Type safety

We cannot have any set of rewrite rules. The rule $(\mathbb{N} \to \mathbb{N}) \rightharpoonup (\mathbb{N} \to \mathbb{B})$ breaks type safety!

```
zero' : 𝔹
zero' = (λx.x : ℕ → 𝔹) zero

test = if zero' then true else false
```

Confluence to the rescue

To regain subject reduction we have two requirements:

1. Rewrite rules should only rewrite fresh symbols.
2. The combination of rewrite rules and β-reduction should be confluent

Confluence: Every way to rewrite a term yields the same result

Parallel reduction

Parallel reduction ($\Rrightarrow$) reduces all immediate redexes by one step:

`(suc a) + ((λx.x + b) 0)` $\Rrightarrow$ `suc (a + (0 + b))`

Used by Tait and Martin-Löf to prove confluence of the untyped lambda calculus.

Triangle property

Each t has a optimal reduct $\rho(t)$:

$$\begin{tikzcd}
t \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \\
u \arrow[r, Rightarrow] & \rho(t)
\end{tikzcd}$$

Triangle property

Each t has a optimal reduct $\rho(t)$:

$$\begin{tikzcd} t \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \\ u \arrow[r, Rightarrow] & \rho(t) \end{tikzcd}$$

This implies global confluence:

$$\begin{tikzcd} & t \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] \arrow[dd, Rightarrow] & \\ u \arrow[dr, Rightarrow] & & u' \arrow[dl, Rightarrow] \\ & \rho(t) & \end{tikzcd}$$

Can be checked modularly!

Revisiting the Commutativity of addition

```
+comm-zero : (m : ℕ) → m + zero ≡ zero + m
+comm-suc : (m n : ℕ) → m + (suc n) ≡ (suc n) + m

+comm-sucsuc : (m n : ℕ) → (suc m) + (suc n) ≡ suc (suc (m + n))

{-# REWRITE +comm-zero +comm-suc +comm-sucsuc #-}

comm : (m n : ℕ) → m + n ≡ n + m
comm zero n    = refl
comm (suc m) n = cong suc (comm m n)
```

Get your hands dirty

- Formalized in MetaCoq
- Supported in Agda
- Will be available in Coq soon™

References

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter, *The taming of the rew: a type theory with computational assumptions*, Proceedings of the ACM on Programming Languages **5** (2021), no. POPL, 1–29.