

A Lightweight Approach to Datatype-Generic Rewriting

Thomas van Noort¹

joint work with

Alexey Rodriguez² Johan Jeuring^{2,3}
Stefan Holdermans² Bastiaan Heeren³

¹Inst. for Computing and Information Sciences, Radboud University Nijmegen

² Department of Information and Computing Sciences, Utrecht University

³ School of Computer Science, Open University of the Netherlands

Workshop on Generic Programming
September 20, 2008

Introduction

Interactive e-learning tools providing semantically rich feedback:

The screenshot shows a window titled "Logical Exercise Solver" with three standard window control buttons (minimize, maximize, close) in the top-left corner. The interface is divided into three main sections:

- Working area:** Contains a text input field with the logical expression $(\neg p \vee \neg q) \vee (r \leftrightarrow \neg s)$. To the right of the input field are three buttons: "Submit", "Undo", and "Hint".
- History:** A list of logical expressions showing the sequence of operations:
 - $\neg(p \vee q) \vee (r \leftrightarrow \neg s)$
 - $\neg(\neg p \vee q) \vee (r \leftrightarrow \neg s)$
 - $\neg(\neg p \rightarrow q) \vee (r \leftrightarrow \neg s)$
 - $(\neg p \rightarrow q) \rightarrow (r \leftrightarrow \neg s)$
- Feedback:** A text area containing the message: "Error: you have rewritten the expression $\neg(p \vee q)$ by $(\neg p \vee \neg q)$. Correctly distributing not over disjunction gives $(\neg p \wedge \neg q)$ ".

At the bottom of the window, a status bar reads "Welcome to the Logical Exercise Solver".

Introduction

Interactive e-learning tools providing semantically rich feedback:

The screenshot shows a window titled "Logical Exercise Solver". It has three main sections: "Working area", "History", and "Feedback".

- Working area:** Contains the expression $(\neg p \vee \neg q) \vee (r \leftrightarrow \neg s)$. To its right are buttons for "Submit", "Undo", and "Hint".
- History:** A list of expressions:
 - $\neg(p \vee q) \vee (r \leftrightarrow \neg s)$
 - $\neg(\neg p \vee q) \vee (r \leftrightarrow \neg s)$
 - $\neg(\neg p \rightarrow q) \vee (r \leftrightarrow \neg s)$
 - $(\neg p \rightarrow q) \rightarrow (r \leftrightarrow \neg s)$
- Feedback:** A text box containing the message: "Error: you have rewritten the expression $\neg(p \vee q)$ by $(\neg p \vee \neg q)$. Correctly distributing not over disjunction gives $(\neg p \wedge \neg q)$ ".

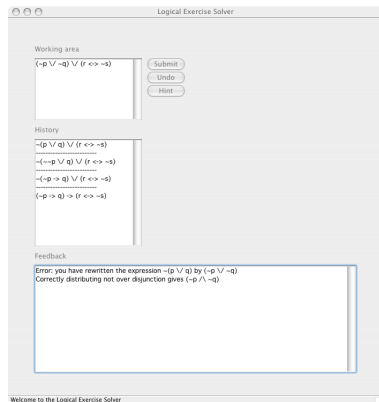
At the bottom left, there is a small text box that says "Welcome to the Logical Exercise Solver".

Term rewriting is the core of feedback generation:

- ▶ Correctly applied rule:
$$p \vee \neg p \rightarrow True$$
- ▶ Incorrectly applied rule:
$$\neg(p \wedge q) \rightarrow \neg p \wedge \neg q$$
- ▶ ...

Introduction

Interactive e-learning tools providing semantically rich feedback:



Term rewriting is the core of feedback generation:

- ▶ Correctly applied rule:
$$p \vee \neg p \rightarrow True$$
- ▶ Incorrectly applied rule:
$$\neg(p \wedge q) \rightarrow \neg p \wedge \neg q$$
- ▶ ...

Rewrite rules need to be observable and conveniently defined!

Rewriting using rules based on pattern matching

Consider a Haskell datatype for logical propositions:

```
data Prop = T | F | Not Prop | Prop :∧: Prop | Prop :∨: Prop
```

```
orTaut :: Prop → Prop
```

```
orTaut (p :∨: Not q) | p ≡ q = T
```

```
orTaut p = p
```

Rewriting using rules based on pattern matching

Consider a Haskell datatype for logical propositions:

```
data Prop = T | F | Not Prop | Prop :∧: Prop | Prop :∨: Prop
orTaut :: Prop → Prop
orTaut (p :∨: Not q) | p ≡ q = T
orTaut p                    = p
```

This approach has some drawbacks:

- ▶ No observability of rules, which is desirable for:
 - ▶ Documentation: pretty-printing rules
 - ▶ Automated testing: generating rule-specific test cases
 - ▶ Inversion: exchanging the sides of a rule
 - ▶ ...
- ▶ Lack of non-linear patterns
- ▶ Specifying a catch-all case
- ▶ No first-class pattern matching

Rewriting using rules as values of datatypes

Overcome the mentioned drawbacks using a datatype:

data *RuleSpec* *a* = *a* : \rightsquigarrow *a*

Rewriting using rules as values of datatypes

Overcome the mentioned drawbacks using a datatype:

```
data RuleSpec a = a :~> a
```

This approach also has some drawbacks:

- ▶ Manual extension of the domain:

```
data EProp = Metavar String | ET | EF | ENot EProp  
          | EProp :⊗: EProp | EProp :⊖: EProp
```

Rewriting using rules as values of datatypes

Overcome the mentioned drawbacks using a datatype:

```
data RuleSpec a = a :~> a
```

This approach also has some drawbacks:

- ▶ Manual extension of the domain:

```
data EProp = Metavar String | ET | EF | ENot EProp  
          | EProp :⊗: EProp | EProp :⊖: EProp
```

- ▶ Rules are defined in the extended domain using datatype-specific rewriting machinery:

```
orTaut :: Prop → Prop
```

```
orTaut = rewriteEProp orTautRule
```

```
where
```

```
orTautRule :: RuleSpec EProp
```

```
orTautRule = Metavar "p" :⊖: ENot (Metavar "p")  
           :~> ET
```

Combining the best of both worlds

Generic rewriting using rules as values of datatypes

$orTaut :: Prop \rightarrow Prop$

$orTaut = rewrite\ orTautRule$

where

$orTautRule :: Rule\ Prop$

$orTautRule = rule\ (\lambda p \rightarrow p : \forall : Not\ p : \rightsquigarrow T)$

Combining the best of both worlds

Generic rewriting using rules as values of datatypes

$orTaut :: Prop \rightarrow Prop$

$orTaut = rewrite\ orTautRule$

where

$orTautRule :: Rule\ Prop$

$orTautRule = rule\ (\lambda p \rightarrow p :V: Not\ p :~\rightsquigarrow T)$

Key features of our approach:

- ▶ Generic rewriting
- ▶ Metavariables as function arguments
- ▶ Convenient interface

Generic rewriting

A functorial representation

Recursion in a type is explicated by a functorial representation:

```
data K a r    = K a
```

```
data Id r     = Id r
```

```
data Unit r  = Unit
```

```
data (f :+ : g) r = Inl (f r) | Inr (g r)
```

```
data (f :* : g) r = f r :* : g r
```

```
infixr 6 :+ :
```

```
infixr 7 :* :
```

A functorial representation

Recursion in a type is explicated by a functorial representation:

```
data K a r    = K a
data Id r     = Id r
data Unit r   = Unit
data (f :+ : g) r = Inl (f r) | Inr (g r)
data (f :* : g) r = f r :* g r
infixr 6 :+ :
infixr 7 :* :
```

Structure of a regular datatype is captured in a type class:

```
class Functor (PF a) ⇒ Regular a where
  type PF a :: * → *
  from      :: a          → (PF a) a
  to       :: (PF a) a → a
```

What is required by the user?

User is only required to provide a single instance:

instance *Regular Prop* **where**

type *PF Prop* = *Unit :+: Unit :+: Id :+: (Id :* Id) :+: (Id :* Id)*

What is required by the user?

User is only required to provide a single instance:

instance *Regular Prop* **where**

type *PF Prop* = *Unit* *:+*: *Unit* *:+*: *Id* *:+*: (*Id* *:**: *Id*) *:+*: (*Id* *:**: *Id*)

from T = *Inl Unit*

from F = *Inr (Inl Unit)*

from (Not p) = *Inr (Inr (Inl (Id p)))*

from (p : \wedge : q) = *Inr (Inr (Inr (Inl (Id p :* Id q))))*

from (p : \vee : q) = *Inr (Inr (Inr (Inr (Id p :* Id q))))*

to (Inl Unit) = *T*

to (Inr (Inl Unit)) = *F*

to (Inr (Inr (Inl (Id p)))) = *Not p*

to (Inr (Inr (Inr (Inl (Id p : Id q))))* = *p : \wedge : q*

to (Inr (Inr (Inr (Inr (Id p : Id q))))* = *p : \vee : q*

Generic functions

By providing such an instance, we get functionality for free:

- ▶ Mapping a function to the immediate children of a functor:

```
class Functor f where  
  fmap :: (a → b) → f a → f b
```

Generic functions

By providing such an instance, we get functionality for free:

- ▶ Mapping a function to the immediate children of a functor:

```
class Functor f where  
  fmap :: (a → b) → f a → f b
```

- ▶ Folding the immediate children of a functor:

```
class Crush f where  
  crush :: (a → b → b) → b → f a → b
```

Generic functions

By providing such an instance, we get functionality for free:

- ▶ Mapping a function to the immediate children of a functor:

```
class Functor f where  
  fmap :: (a → b) → f a → f b
```

- ▶ Folding the immediate children of a functor:

```
class Crush f where  
  crush :: (a → b → b) → b → f a → b
```

- ▶ Zipping the immediate children of two functors:

```
class Zip f where  
  fzipM :: Monad m ⇒ (a → b → m c) → f a → f b → m (f c)
```

Additionally, two non-monadic versions are defined:

```
fzip :: (Zip f, Monad m) ⇒ (a → b → c) → f a → f b → m (f c)  
fzip' :: Zip f ⇒ (a → b → c) → f a → f b → f c
```

Defining rewrite rules

Rules consist of a left-hand side and a right-hand side:

data *RuleSpec* *a* = *a* : \rightsquigarrow *a*

lhs, *rhs* :: *RuleSpec* *a* \rightarrow *a*

lhs (*x* : \rightsquigarrow $_$) = *x*

rhs ($_$: \rightsquigarrow *y*) = *y*

What should be the type of rules that rewrite *Prop* values?

Defining rewrite rules

Rules consist of a left-hand side and a right-hand side:

data *RuleSpec* *a* = *a* : \rightsquigarrow *a*

lhs, *rhs* :: *RuleSpec* *a* \rightarrow *a*

lhs (*x* : \rightsquigarrow $_$) = *x*

rhs ($_$: \rightsquigarrow *y*) = *y*

What should be the type of rules that rewrite *Prop* values?

RuleSpec Prop values cannot contain metavariables!

Extending the domain

Add a case for a metavariable using a type-indexed type:

```
type Ext f = K Metavar :+ : f
```

```
type Metavar = Int
```

Extending the domain

Add a case for a metavariable using a type-indexed type:

```
type Ext f = K Metavar :+ : f
```

```
type Metavar = Int
```

Extend the pattern functor recursively:

```
type Scheme f = Fix (Ext f)
```

```
newtype Fix f = In (f (Fix f))
```

Extending the domain

Add a case for a metavariable using a type-indexed type:

```
type Ext f = K Metavar :+ : f  
type Metavar = Int
```

Extend the pattern functor recursively:

```
type Scheme f = Fix (Ext f)  
newtype Fix f = In (f (Fix f))
```

Define a scheme for a regular type:

```
type SchemeOf a = Scheme (PF a)
```

Extending the domain

Add a case for a metavariable using a type-indexed type:

```
type Ext f = K Metavar :+ : f  
type Metavar = Int
```

Extend the pattern functor recursively:

```
type Scheme f = Fix (Ext f)  
newtype Fix f = In (f (Fix f))
```

Define a scheme for a regular type:

```
type SchemeOf a = Scheme (PF a)
```

Then, rules are defined on the extended domain:

```
type Rule a = RuleSpec (SchemeOf a)
```

Constructor functions for schemes

Constructing *Scheme* values:

$$\text{metavar} :: \text{Metavar} \rightarrow \text{Scheme } f$$
$$\text{metavar} = \text{In} \circ \text{Inl} \circ K$$
$$\text{pf} :: f(\text{Scheme } f) \rightarrow \text{Scheme } f$$
$$\text{pf} = \text{In} \circ \text{Inr}$$

Constructor functions for schemes

Constructing *Scheme* values:

$$\text{metavar} :: \text{Metavar} \rightarrow \text{Scheme } f$$
$$\text{metavar} = \text{In} \circ \text{Inl} \circ K$$
$$\text{pf} :: f(\text{Scheme } f) \rightarrow \text{Scheme } f$$
$$\text{pf} = \text{In} \circ \text{Inr}$$

Embedding *a* values into their extended counterparts:

$$\text{toScheme} :: \text{Regular } a \Rightarrow a \rightarrow \text{SchemeOf } a$$
$$\text{toScheme} = \text{pf} \circ \text{fmap } \text{toScheme} \circ \text{from}$$

A convenient view on schemes

Distinguishing metavariables from pattern functor values:

```
data SchemeView f = Metavar Metavar | PF (f (Scheme f))  
schemeView:: Scheme f → SchemeView f  
schemeView (In (Inl (K x))) = Metavar x  
schemeView (In (Inr r))      = PF r
```

A convenient view on schemes

Distinguishing metavariables from pattern functor values:

```
data SchemeView f = Metavar Metavar | PF (f (Scheme f))  
schemeView :: Scheme f → SchemeView f  
schemeView (In (Inl (K x))) = Metavar x  
schemeView (In (Inr r))      = PF r
```

Folding *Scheme* values:

```
foldScheme :: Functor f  
            ⇒ (Metavar → a) → (f a → a) → Scheme f → a  
foldScheme f g scheme =  
  case schemeView scheme of  
    Metavar x → f x  
    PF r      → g (fmap (foldScheme f g) r)
```

Matching a term

Matching the left-hand side of a rule returns a substitution:

```
type Subst a = Map Metavar (a, SchemeOf a)
```

Matching a term

Matching the left-hand side of a rule returns a substitution:

```
type Subst a = Map Metavar (a, SchemeOf a)  
match :: (Regular a, Crush (PF a), Zip (PF a), Monad m)  
        ⇒ SchemeOf a → a → m (Subst a)  
match scheme term =
```

Matching a term

Matching the left-hand side of a rule returns a substitution:

```
type Subst a = Map Metavar (a, SchemeOf a)  
match :: (Regular a, Crush (PF a), Zip (PF a), Monad m)  
        ⇒ SchemeOf a → a → m (Subst a)  
match scheme term =  
  case schemeView scheme of  
    Metavar x →  
      return (singleton x (term, toScheme term))
```

Matching a term

Matching the left-hand side of a rule returns a substitution:

```
type Subst a = Map Metavar (a, SchemeOf a)
match :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
        => SchemeOf a -> a -> m (Subst a)
match scheme term =
  case schemeView scheme of
    Metavar x ->
      return (singleton x (term, toScheme term))
    PF r ->
      fzip (,) r (from term) >>=
      crush matchOne (return empty)
```

Matching a term

Matching the left-hand side of a rule returns a substitution:

```
type Subst a = Map Metavar (a, SchemeOf a)  
match :: (Regular a, Crush (PF a), Zip (PF a), Monad m)  
       ⇒ SchemeOf a → a → m (Subst a)
```

```
match scheme term =
```

```
  case schemeView scheme of
```

```
    Metavar x →
```

```
      return (singleton x (term, toScheme term))
```

```
    PF r      →
```

```
      fzip (,) r (from term) >>=
```

```
      crush matchOne (return empty)
```

```
where
```

```
  matchOne (term1, term2) msubst =
```

```
    do subst1 ← msubst
```

```
      subst2 ← match (apply subst1 term1) term2
```

```
      return (union subst1 subst2)
```

Building a term

Using the obtained substitution comes in two flavours:

- ▶ Applying a substitution to a term:

$apply :: Regular\ a \Rightarrow Subst\ a \rightarrow SchemeOf\ a \rightarrow SchemeOf\ a$
 $apply\ subst = foldScheme\ findMetavar\ pf$

where

$findMetavar\ x = maybe\ (metavar\ x)\ snd\ (lookup\ x\ subst)$

Building a term

Using the obtained substitution comes in two flavours:

- ▶ Applying a substitution to a term:

$apply :: Regular\ a \Rightarrow Subst\ a \rightarrow SchemeOf\ a \rightarrow SchemeOf\ a$
 $apply\ subst = foldScheme\ findMetavar\ pf$

where

$findMetavar\ x = maybe\ (metavar\ x)\ snd\ (lookup\ x\ subst)$

- ▶ Instantiating each metavariable in a term:

$inst :: Regular\ a \Rightarrow Subst\ a \rightarrow SchemeOf\ a \rightarrow a$
 $inst\ subst = foldScheme\ findMetavar\ to$

where

$findMetavar\ x = maybe\ \perp\ fst\ (lookup\ x\ subst)$

Rewriting a term

Everything comes together in a single function:

$$\text{rewriteM} :: (\text{Regular } a, \text{Crush } (PF\ a), \text{Zip } (PF\ a), \text{Monad } m) \\ \Rightarrow \text{Rule } a \rightarrow a \rightarrow m\ a$$

$\text{rewriteM } r\ term =$

$$\mathbf{do}\ subst \leftarrow \text{match } (lhs\ r)\ term \\ \text{return } (inst\ subst\ (rhs\ r))$$

Rewriting a term

Everything comes together in a single function:

```
rewriteM :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
          => Rule a -> a -> m a
rewriteM r term =
  do subst <- match (lhs r) term
      return (inst subst (rhs r))
```

The non-monadic version always succeeds:

```
rewrite :: (Regular a, Crush (PF a), Zip (PF a))
         => Rule a -> a -> a
rewrite r term = maybe term id (rewriteM r term)
```

Example

Now, we are able to use the generic rewriting machinery:

orTaut :: *Prop* → *Prop*

orTaut = *rewrite orTautRule*

Example

Now, we are able to use the generic rewriting machinery:

$orTaut :: Prop \rightarrow Prop$

$orTaut = rewrite orTautRule$

where

$orTautRule :: Rule Prop$

$orTautRule =$

$In (Inr (Inr (Inr (Inr (Inr ($
 $Id (In (Inl (K 1)))) :*$
 $Id (In (Inr (Inr (Inl ($
 $Id (In (Inl (K 1))))))))))))))$

$:\rightsquigarrow$

$In (Inr (Inl Unit))$

Example

Now, we are able to use the generic rewriting machinery:

```
orTaut :: Prop → Prop
orTaut = rewrite orTautRule
where
  orTautRule :: Rule Prop
  orTautRule =
    In (Inr (Inr (Inr (Inr (Inr (
      Id (In (Inl (K 1))) :*
      Id (In (Inr (Inr (Inl (
        Id (In (Inl (K 1))))))))))))))
    :~>
    In (Inr (Inl Unit))
```

Although helper functions could help, these rule definitions:

- ▶ Remain absolutely unreadable
- ▶ Do not allow the use of the original datatype constructors
- ▶ Contain structure information that is already provided

Metavariables as function arguments

Rules without metavariables

Consider a rule without any metavariables:

notTrue :: *RuleSpec Prop*

notTrue = *Not T:↔ F*

Rules without metavariables

Consider a rule without any metavariables:

$$\text{notTrue} :: \text{RuleSpec Prop}$$
$$\text{notTrue} = \text{Not } T : \rightsquigarrow F$$

Transform the rule specification to a rule:

$$\text{notTrueRule} :: \text{Rule Prop}$$
$$\text{notTrueRule} = \text{rule}_0 \text{ notTrue}$$

Rules without metavariables

Consider a rule without any metavariables:

$$\begin{aligned} \text{notTrue} &:: \text{RuleSpec Prop} \\ \text{notTrue} &= \text{Not } T : \rightsquigarrow F \end{aligned}$$

Transform the rule specification to a rule:

$$\begin{aligned} \text{notTrueRule} &:: \text{Rule Prop} \\ \text{notTrueRule} &= \text{rule}_0 \text{ notTrue} \end{aligned}$$

Applying *toScheme* does the trick:

$$\begin{aligned} \text{rule}_0 &:: \text{Regular } a \Rightarrow \text{RuleSpec } a \rightarrow \text{Rule } a \\ \text{rule}_0 r &= \text{toScheme } (\text{lhs } r) : \rightsquigarrow \text{toScheme } (\text{rhs } r) \end{aligned}$$

Rules with one metavariable

Single metavariable as argument of a rule:

orTaut :: *Prop* → *RuleSpec Prop*

orTaut *p* = *p* : \forall : *Not* *p* : \rightsquigarrow *T*

Rules with one metavariable

Single metavariable as argument of a rule:

$orTaut :: Prop \rightarrow RuleSpec Prop$

$orTaut p = p : \forall : Not p : \rightsquigarrow T$

Transform the rule specification to a rule:

$orTautRule :: Rule Prop$

$orTautRule = rule_1 orTaut$

Rules with one metavariable

Single metavariable as argument of a rule:

$$\begin{aligned} \text{orTaut} &:: \text{Prop} \rightarrow \text{RuleSpec Prop} \\ \text{orTaut } p &= p : \forall : \text{Not } p : \rightsquigarrow T \end{aligned}$$

Transform the rule specification to a rule:

$$\begin{aligned} \text{orTautRule} &:: \text{Rule Prop} \\ \text{orTautRule} &= \text{rule}_1 \text{ orTaut} \end{aligned}$$

Requires introduction of a single metavariable:

$$\begin{aligned} \text{rule}_1 &:: (\text{Regular } a, \text{Zip } (PF \ a), \text{LR } (PF \ a)) \\ &\Rightarrow (a \rightarrow \text{RuleSpec } a) \rightarrow \text{Rule } a \\ \text{rule}_1 \ r &= \text{introMetavar } (\text{lhs} \circ r) : \rightsquigarrow \text{introMetavar } (\text{rhs} \circ r) \end{aligned}$$

Diff algorithm for one metavariable

How are metavariables inserted in the right position?

Diff algorithm for one metavariable

How are metavariables inserted in the right position?

Apply the rule to two different values:

$introMetavar :: (Regular\ a, Zip\ (PF\ a), LR\ (PF\ a))$

$\Rightarrow (a \rightarrow a) \rightarrow SchemeOf\ a$

$introMetavar\ f = insertMetavar\ 1\ (f\ left)\ (f\ right)$

Diff algorithm for one metavariable

How are metavariables inserted in the right position?

Apply the rule to two different values:

$$\begin{aligned} \text{introMetavar} &:: (\text{Regular } a, \text{Zip } (PF\ a), LR\ (PF\ a)) \\ &\Rightarrow (a \rightarrow a) \rightarrow \text{SchemeOf } a \\ \text{introMetavar } f &= \text{insertMetavar } 1\ (f\ \text{left})\ (f\ \text{right}) \end{aligned}$$

Position of a metavariable is indicated by an *fzip* failure:

$$\begin{aligned} \text{insertMetavar} &:: (\text{Regular } a, \text{Zip } (PF\ a)) \\ &\Rightarrow \text{Metavar} \rightarrow a \rightarrow a \rightarrow \text{SchemeOf } a \\ \text{insertMetavar } v\ x\ y &= \\ &\quad \text{case } fzip\ (\text{insertMetavar } v)\ (\text{from } x)\ (\text{from } y)\ \text{of} \\ &\quad \quad \text{Just } str \rightarrow pf\ str \\ &\quad \quad \text{Nothing} \rightarrow \text{metavar } v \end{aligned}$$

Rules with two metavariables

Two metavariables as arguments of a rule:

$deMorgan :: Prop \rightarrow Prop \rightarrow RuleSpec Prop$

$deMorgan\ p\ q = Not\ (p\ :\wedge\ q) :\rightsquigarrow Not\ p\ :\vee\ Not\ q$

Rules with two metavariables

Two metavariables as arguments of a rule:

$deMorgan :: Prop \rightarrow Prop \rightarrow RuleSpec Prop$

$deMorgan\ p\ q = Not\ (p \wedge q) \rightsquigarrow Not\ p \vee Not\ q$

Transform the rule specification to a rule:

$deMorganRule :: Rule Prop$

$deMorganRule = rule_2\ deMorgan$

Rules with two metavariables

Two metavariables as arguments of a rule:

$$\begin{aligned} deMorgan &:: Prop \rightarrow Prop \rightarrow RuleSpec Prop \\ deMorgan\ p\ q &= Not\ (p\ :\wedge\ q)\ :\rightsquigarrow\ Not\ p\ :\vee\ Not\ q \end{aligned}$$

Transform the rule specification to a rule:

$$\begin{aligned} deMorganRule &:: Rule Prop \\ deMorganRule &= rule_2\ deMorgan \end{aligned}$$

Requires introduction of two metavariables:

$$\begin{aligned} rule_2 &:: (Regular\ a,\ Zip\ (PF\ a),\ LR\ (PF\ a)) \\ &\Rightarrow (a \rightarrow a \rightarrow RuleSpec\ a) \rightarrow Rule\ a \\ rule_2\ r &= \\ &\quad introMetavar_2\ (\lambda x\ y \rightarrow lhs\ (r\ x\ y))\ :\rightsquigarrow \\ &\quad introMetavar_2\ (\lambda x\ y \rightarrow rhs\ (r\ x\ y)) \end{aligned}$$

Diff algorithm for two metavariables

Requires two steps, each step inserting a single metavariable:

$$\begin{aligned} \text{introMetavar}_2 &:: (\text{Regular } a, \text{Zip } (PF\ a), \text{LR } (PF\ a)) \\ &\Rightarrow (a \rightarrow a \rightarrow a) \rightarrow \text{SchemeOf } a \end{aligned}$$
$$\text{introMetavar}_2\ f = \text{term}_1\ \text{'mergeSchemes'}\ \text{term}_2$$

where

$$\text{term}_1 = \text{insertMetavar } 1\ (f\ \text{left}\ \text{left})\ (f\ \text{right}\ \text{left})$$
$$\text{term}_2 = \text{insertMetavar } 2\ (f\ \text{left}\ \text{left})\ (f\ \text{left}\ \text{right})$$

Diff algorithm for two metavariables

Requires two steps, each step inserting a single metavariable:

$$\begin{aligned} \text{introMetavar}_2 &:: (\text{Regular } a, \text{Zip } (PF\ a), \text{LR } (PF\ a)) \\ &\Rightarrow (a \rightarrow a \rightarrow a) \rightarrow \text{SchemeOf } a \end{aligned}$$
$$\text{introMetavar}_2\ f = \text{term}_1\ \text{'mergeSchemes'}\ \text{term}_2$$

where

$$\text{term}_1 = \text{insertMetavar } 1\ (f\ \text{left}\ \text{left})\ (f\ \text{right}\ \text{left})$$
$$\text{term}_2 = \text{insertMetavar } 2\ (f\ \text{left}\ \text{left})\ (f\ \text{left}\ \text{right})$$

Schemes are merged by keeping the inserted metavariables:

$$\text{mergeSchemes} :: \text{Zip } f \Rightarrow \text{Scheme } f \rightarrow \text{Scheme } f \rightarrow \text{Scheme } f$$
$$\text{mergeSchemes } p@(In\ x)\ q@(In\ y) =$$

case (schemeView p, schemeView q) **of**

$$(\text{Metavar } _ , _) \rightarrow p$$
$$(_, \text{Metavar } _) \rightarrow q$$
$$_ \rightarrow In\ (fzip'\ \text{mergeSchemes } x\ y)$$

Convenient interface

A general pattern for building rules

A clear pattern is visible in the $rule_n$ functions:

$$\underbrace{(f \text{ left left} \cdots \text{ left})}_{\text{base}} \bowtie \underbrace{\left\{ \begin{array}{l} (f \text{ right left} \cdots \text{ left}) \\ (f \text{ left right} \cdots \text{ left}) \\ \vdots \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\ (f \text{ left left} \cdots \text{ right}) \end{array} \right\}}_{\text{diag}}$$

A general pattern for building rules

A clear pattern is visible in the $rule_n$ functions:

$$\underbrace{(f \text{ left left} \cdots \text{ left})}_{\text{base}} \bowtie \underbrace{\left\{ \begin{array}{l} (f \text{ right } \text{ left } \cdots \text{ left }) \\ (f \text{ left } \text{ right } \cdots \text{ left }) \\ \vdots \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\ (f \text{ left } \text{ left } \cdots \text{ right }) \end{array} \right.}_{\text{diag}}$$

The corresponding functions are captured in a type class:

```
class Regular (Target a)  $\Rightarrow$  Builder a where  
  type Target a :: *  
  base           :: a  $\rightarrow$  RuleSpec (Target a)  
  diag          :: a  $\rightarrow$  [RuleSpec (Target a)]
```

A general pattern for building rules

A clear pattern is visible in the $rule_n$ functions:

$$\underbrace{(f \text{ left left} \cdots \text{ left})}_{\text{base}} \bowtie \underbrace{\left\{ \begin{array}{l} (f \text{ right left} \cdots \text{ left}) \\ (f \text{ left right} \cdots \text{ left}) \\ \vdots \quad \vdots \quad \vdots \quad \ddots \quad \vdots \\ (f \text{ left left} \cdots \text{ right}) \end{array} \right\}}_{\text{diag}}$$

The corresponding functions are captured in a type class:

```
class Regular (Target a)  $\Rightarrow$  Builder a where  
  type Target a :: *  
  base           :: a  $\rightarrow$  RuleSpec (Target a)  
  diag          :: a  $\rightarrow$  [RuleSpec (Target a)]
```

The associated type synonym describes the domain:

```
Target (           RuleSpec Prop)  $\equiv$  Prop  
Target (   Prop  $\rightarrow$  RuleSpec Prop)  $\equiv$  Prop  
Target (Prop  $\rightarrow$  Prop  $\rightarrow$  RuleSpec Prop)  $\equiv$  Prop
```

Inductive type class instances

Instance for a rule without metavariables:

```
instance Regular a  $\Rightarrow$  Builder (RuleSpec a) where  
  type Target (RuleSpec a) = a  
  base x                       = x  
  diag x                        = [x]
```

Inductive type class instances

Instance for a rule without metavariables:

```
instance Regular a  $\Rightarrow$  Builder (RuleSpec a) where  
  type Target (RuleSpec a) = a  
  base x                       = x  
  diag x                       = [x]
```

Instance for a rule with $n + 1$ metavariables:

```
instance (Builder a, Regular b, LR (PF b))  
   $\Rightarrow$  Builder (b  $\rightarrow$  a) where  
  type Target (b  $\rightarrow$  a) = Target a  
  base f                       = base (f left)  
  diag f                       = base (f right) : diag (f left)
```

Rule with n metavariables

We capture the pattern using the overloaded functions:

$$\begin{aligned} \text{rule} &:: (\text{Builder } r, \text{Zip } (PF \text{ (Target } r))) \Rightarrow r \rightarrow \text{Rule } (\text{Target } r) \\ \text{rule } f &= \text{foldr1 mergeRules rules} \end{aligned}$$

Rule with n metavariables

We capture the pattern using the overloaded functions:

$rule :: (Builder\ r, Zip\ (PF\ (Target\ r))) \Rightarrow r \rightarrow Rule\ (Target\ r)$

$rule\ f = foldr1\ mergeRules\ rules$

where

$mergeRules\ x\ y =$

$mergeSchemes\ (lhs\ x)\ (lhs\ y) \rightsquigarrow$

$mergeSchemes\ (rhs\ x)\ (rhs\ y)$

Rule with n metavariables

We capture the pattern using the overloaded functions:

$rule :: (Builder\ r, Zip\ (PF\ (Target\ r))) \Rightarrow r \rightarrow Rule\ (Target\ r)$

$rule\ f = foldr1\ mergeRules\ rules$

where

$mergeRules\ x\ y =$

$mergeSchemes\ (lhs\ x)\ (lhs\ y) \rightsquigarrow$

$mergeSchemes\ (rhs\ x)\ (rhs\ y)$

$rules = zipWith\ (ins\ (base\ f))\ (diag\ f)\ [1..]$

Rule with n metavariables

We capture the pattern using the overloaded functions:

$rule :: (Builder\ r, Zip\ (PF\ (Target\ r))) \Rightarrow r \rightarrow Rule\ (Target\ r)$

$rule\ f = foldr1\ mergeRules\ rules$

where

$mergeRules\ x\ y =$

$mergeSchemes\ (lhs\ x)\ (lhs\ y) : \rightsquigarrow$

$mergeSchemes\ (rhs\ x)\ (rhs\ y)$

$rules = zipWith\ (ins\ (base\ f))\ (diag\ f)\ [1..]$

$ins\ x\ y\ v =$

$insertMetavar\ v\ (lhs\ x)\ (lhs\ y) : \rightsquigarrow$

$insertMetavar\ v\ (rhs\ x)\ (rhs\ y)$

Rule with n metavariables

We capture the pattern using the overloaded functions:

$$\text{rule} :: (\text{Builder } r, \text{Zip } (PF (\text{Target } r))) \Rightarrow r \rightarrow \text{Rule } (\text{Target } r)$$
$$\text{rule } f = \text{foldr1 } \text{mergeRules } \text{rules}$$

where

$$\text{mergeRules } x \ y =$$
$$\text{mergeSchemes } (\text{lhs } x) \ (\text{lhs } y) \ : \rightsquigarrow$$
$$\text{mergeSchemes } (\text{rhs } x) \ (\text{rhs } y)$$
$$\text{rules} = \text{zipWith } (\text{ins } (\text{base } f)) \ (\text{diag } f) \ [1 \dots]$$
$$\text{ins } x \ y \ v =$$
$$\text{insertMetavar } v \ (\text{lhs } x) \ (\text{lhs } y) \ : \rightsquigarrow$$
$$\text{insertMetavar } v \ (\text{rhs } x) \ (\text{rhs } y)$$

Rules are now uniformly typed:

$$\text{notTrueRule}, \text{orTautRule}, \text{deMorganRule} :: \text{Rule Prop}$$
$$\text{notTrueRule} = \text{rule } (\text{Not } T \quad \quad \quad : \rightsquigarrow F)$$
$$\text{orTautRule} = \text{rule } (\lambda p \rightarrow p : \forall : \text{Not } p \quad \quad \quad : \rightsquigarrow T)$$
$$\text{deMorganRule} = \text{rule } (\lambda p \ q \rightarrow \text{Not } (p \wedge : q) \quad \quad \quad : \rightsquigarrow \text{Not } p : \forall : \text{Not } q)$$

The finishing touch

The interface functions expose the generic functions used:

$$\begin{aligned} \text{rewrite} &:: (\textit{Regular } a, \textit{Crush } (PF\ a), \textit{Zip } (PF\ a)) \\ &\Rightarrow \textit{Rule } a \rightarrow a \rightarrow a \end{aligned}$$

The finishing touch

The interface functions expose the generic functions used:

$$\begin{aligned} \text{rewrite} &:: (\textit{Regular } a, \textit{Crush } (PF\ a), \textit{Zip } (PF\ a)) \\ &\Rightarrow \textit{Rule } a \rightarrow a \rightarrow a \end{aligned}$$

A type class synonym summarizes all the constraints:

$$\begin{aligned} \mathbf{class} &(\textit{Regular } a, \textit{Crush } (PF\ a), \textit{Zip } (PF\ a), \textit{LR } (PF\ a)) \\ &\Rightarrow \textit{Rewrite } a \end{aligned}$$

The finishing touch

The interface functions expose the generic functions used:

$$\begin{aligned} \text{rewrite} &:: (\textit{Regular } a, \textit{Crush } (PF\ a), \textit{Zip } (PF\ a)) \\ &\Rightarrow \textit{Rule } a \rightarrow a \rightarrow a \end{aligned}$$

A type class synonym summarizes all the constraints:

$$\begin{aligned} \mathbf{class} &(\textit{Regular } a, \textit{Crush } (PF\ a), \textit{Zip } (PF\ a), \textit{LR } (PF\ a)) \\ &\Rightarrow \textit{Rewrite } a \end{aligned}$$

The interface functions now hide the implementation details:

$$\text{rewrite} :: \textit{Rewrite } a \Rightarrow \textit{Rule } a \rightarrow a \rightarrow a$$

The finishing touch

The interface functions expose the generic functions used:

$$\begin{aligned} \text{rewrite} &:: (\text{Regular } a, \text{Crush } (PF\ a), \text{Zip } (PF\ a)) \\ &\Rightarrow \text{Rule } a \rightarrow a \rightarrow a \end{aligned}$$

A type class synonym summarizes all the constraints:

$$\begin{aligned} \mathbf{class} &(\text{Regular } a, \text{Crush } (PF\ a), \text{Zip } (PF\ a), \text{LR } (PF\ a)) \\ &\Rightarrow \text{Rewrite } a \end{aligned}$$

The interface functions now hide the implementation details:

$$\text{rewrite} :: \text{Rewrite } a \Rightarrow \text{Rule } a \rightarrow a \rightarrow a$$

An additional instance for *Prop* is a small price to pay:

$$\mathbf{instance} \text{Rewrite } \text{Prop}$$

Concluding remarks

Generic rewriting using rules as values of datatypes

Concluding remarks

Generic rewriting using rules as values of datatypes

A lightweight generic rewriting library with great features:

- ▶ Full observability of rules
- ▶ Rules are defined in the original domain
- ▶ Convenient interface

Concluding remarks

Generic rewriting using rules as values of datatypes

A lightweight generic rewriting library with great features:

- ▶ Full observability of rules
- ▶ Rules are defined in the original domain
- ▶ Convenient interface

However, there are some limitations:

- ▶ Initial experiments showed a decrease (7 ~ 8 times) in performance compared to using rules as functions
- ▶ Only applicable to regular datatypes
- ▶ Metavariables are not allowed to be inspected

Generic programming libraries

The library will be released on Hackage soon:

- ▶ `rewriting`

The regular datatype restriction will be relieved by:

- ▶ `multirec`, fixed-point representation for systems of datatypes (amongst others, mutually recursive datatypes)

Ad: the following library has just been released:

- ▶ `emgm`, a lightweight generic programming library