A library for lightweight higher-order rewriting in Haskell

Emil Axelsson   Andrea Vezzosi
Chalmers University of Technology

TFP 2015, Sophia Antipolis

Introduction: Embedded DSLs

Functional for loop from the Feldspar EDSL:

```
forLoop
    :: Data Int                         -- Number of iterations
    → Data s                            -- Initial state
    → (Data Int → Data s → Data s)      -- Loop body
    → Data s                            -- Final state
```

Example, sum-of-squares:

```
sumOfSquares n = forLoop n 0 $ λi s → i*i + s
```

Embedded DSLs, simplification

Simplification rules:

```
forLoop 0 INIT (λi s → _)       ⟶  INIT

forLoop _ INIT (λi s → s)       ⟶  INIT

forLoop L INIT (λi s → BODY)    ⟶  cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

- Informal notation
- Meta-variables in SMALLCAPS

Embedded DSLs, simplification

$$\texttt{forLoop 0}\ \ \text{INIT}\ \ (\lambda \texttt{i s} \to \texttt{\_})\ \ \longrightarrow\ \ \text{INIT}$$
$$\texttt{forLoop \_}\ \ \text{INIT}\ \ (\lambda \texttt{i s} \to \texttt{s})\ \ \longrightarrow\ \ \text{INIT}$$

Embedded DSLs, simplification

```
forLoop 0 INIT (λi s → _) ⟶ INIT
forLoop _ INIT (λi s → s) ⟶ INIT
```

Assume:

```
type Data a = Data'
data Data'  = Int Int | ForLoop Data' Data' Data'
            | Cond Data' Data' Data' | Lam Name Data' | Var Name | ...
```

Embedded DSLs, simplification

```
forLoop 0 INIT (λi s → _) ⟶ INIT
forLoop _ INIT (λi s → s) ⟶ INIT
```

Assume:

```
type Data a = Data'
data Data'  = Int Int | ForLoop Data' Data' Data'
            | Cond Data' Data' Data' | Lam Name Data' | Var Name | ...
```

Possible implementation:

```
simplify (ForLoop (Int 0) init _)                          = init
simplify (ForLoop _ init (Lam i (Lam s (Var s')))) | s == s' = init
```

- No guarantee of well-typedness
- Comparison of variable names
- Leaks the representation of `Data`
  - Especially problematic for non-standard encodings, such as Data Types à la Carte

Embedded DSLs, simplification

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

Possible implementation:

```
-- Yuck! Ugly code ...
```

Easy to forget:

- Check free variables
- Substitute on the RHS

Goal

EDSL techniques gives us nice and safe syntax for constructing expressions:

```
sumOfSquares n = forLoop n 0 $ λi s → i*i + s
```

- Type-safe
- Scope-safe
- Abstract

Remember:

```
forLoop :: Data Int
        → Data s
        → (Data Int → Data s → Data s)
        → Data s
```

Goal

EDSL techniques gives us nice and safe syntax for constructing expressions:

```
sumOfSquares n = forLoop n 0 $ λi s → i*i + s
```

- Type-safe
- Scope-safe
- Abstract

Remember:

```
forLoop :: Data Int
        → Data s
        → (Data Int → Data s → Data s)
        → Data s
```

Can we use the same techniques for pattern matching and rewriting?

Solution

A higher-order rewriting library:
https://github.com/emilaxelsson/ho-rewriting

Solution

A higher-order rewriting library:
https://github.com/emilaxelsson/ho-rewriting

Instead of:

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

Solution

A higher-order rewriting library:
https://github.com/emilaxelsson/ho-rewriting

Instead of:

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

write:

```
rule_for3 len init body =
    forLoop (mvar len) (mvar init) (λi s → body -$- i)
      ⟹
    cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))
```

Solution

```
rule_for3 len init body =
    forLoop (mvar len) (mvar init) (λi s → body -$- i)
      ⟹
    cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))
```

Solution

```
rule_for3 len init body =
    forLoop (mvar len) (mvar init) (λi s → body -$- i)
      ⟹
    cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))
```

- Type-safe
  - `forLoop`, `cond`, etc. are the same as for construction

Solution

```
rule_for3 len init body =
    forLoop (mvar len) (mvar init) (λi s → body -$- i)
      ⟹
    cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))
```

- Type-safe
  - `forLoop`, `cond`, etc. are the same as for construction
- Scope-safe
  - Uses Haskell's λ notation
  - `body` is not allowed to have `i` or `s` as free variables

Solution

```
rule_for3 len init body =
    forLoop (mvar len) (mvar init) (λi s → body -$- i)
      ⟹
    cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))
```

- Type-safe
  - `forLoop`, `cond`, etc. are the same as for construction
- Scope-safe
  - Uses Haskell's λ notation
  - `body` is not allowed to have `i` or `s` as free variables
- Abstract
  - Uses only "smart constructors"

The rest

- Higher-order rewriting
- Some words about the implementation

Higher-order matching

Previous rule:

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

Higher-order matching

Previous rule:

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

Slight change in the rule:

```
forLoop L INIT (λi s → BODY i) ⟶ cond (L === 0) INIT (BODY (L-1))
```

Higher-order matching

Previous rule:

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

Slight change in the rule:

```
forLoop L INIT (λi s → BODY i) ⟶ cond (L === 0) INIT (BODY (L-1))
```

Semantics: BODY i matches any expression that

- can be β-expanded to the form `expr i`
- such that `i` and `s` do not occur freely in `expr`

Higher-order matching

Previous rule:

```
forLoop L INIT (λi s → BODY) ⟶ cond (L === 0) INIT (BODY [i ↦ L-1])
  where s ∉ freeVars BODY
```

Slight change in the rule:

```
forLoop L INIT (λi s → BODY i) ⟶ cond (L === 0) INIT (BODY (L-1))
```

Semantics: `BODY i` matches any expression that

- can be β-expanded to the form `expr i`
- such that `i` and `s` do not occur freely in `expr`

- No risk for variables escaping
- Forgetting to apply BODY on the LHS is a type error

Higher-order matching, example

Pattern: `forLoop` L INIT (λ`i s` → BODY `i`)

Term: `forLoop 10 0 (`λ`x y` → `x-2)`

Higher-order matching, example

Pattern: `forLoop` L INIT (λ`i s` → BODY `i`)

Term: `forLoop 10 0` (λ`x y` → `x-2`)

β-expand `x-2` to the form `expr x`

Higher-order matching, example

Pattern: `forLoop` $\textsc{l}$ $\textsc{init}$ `(`λ`i s` $\rightarrow$ $\textsc{body}$ `i)`

Term: `forLoop 10 0 (`λ`x y` $\rightarrow$ `x-2)`

β-expand `x-2` to the form `expr x`: `(`λ`z` $\rightarrow$ `z-2) x`

Higher-order matching, example

Pattern: `forLoop` $\textsc{l}$ $\textsc{init}$ `(`λ`i s` $\to$ $\textsc{body}$ `i)`

Term: `forLoop 10 0 (`λ`x y` $\to$ `x-2)`

β-expand `x-2` to the form `expr x`: `(`λ`z` $\to$ `z-2) x`

Resulting substitution:

```
[ L    ↦ 10
, INIT ↦ 0
, BODY ↦ λz → z-2
]
```

Higher-order matching, example

Pattern: `forLoop` L INIT `(`λ`i s` $\to$ BODY `i)`

Term: `forLoop 10 0 (`λ`x y` $\to$ `x-2)`

β-expand `x-2` to the form `expr x`: `(`λ`z` $\to$ `z-2) x`

Resulting substitution:

$$
\begin{array}{lll}
[\ \mathrm{L} & \mapsto & \mathtt{10} \\
,\ \mathrm{INIT} & \mapsto & \mathtt{0} \\
,\ \mathrm{BODY} & \mapsto & \lambda\mathtt{z} \to \mathtt{z\text{-}2} \\
]
\end{array}
$$

Or:

$$
\begin{array}{lll}
[\ \mathrm{L} & \mapsto & \mathtt{10} \\
,\ \mathrm{INIT} & \mapsto & \mathtt{0} \\
,\ \mathrm{BODY} & \mapsto & \lambda\mathtt{x} \to \mathtt{x\text{-}2} \\
]
\end{array}
$$

Higher-order matching, example

Pattern: `forLoop` $\textsc{l}$ $\textsc{init}$ `(`λ`i s` $\to$ $\textsc{body}$ `i)`

Term: `forLoop 10 0 (`λ`x y` $\to$ `x-2)`

β-expand `x-2` to the form `expr x`: `(`λ`z` $\to$ `z-2) x`

Resulting substitution:

$$
\begin{array}{lll}
[\ \textsc{l} & \mapsto & \texttt{10} \\
,\ \textsc{init} & \mapsto & \texttt{0} \\
,\ \textsc{body} & \mapsto & \lambda\texttt{z} \to \texttt{z-2} \\
]
\end{array}
$$

Or:

$$
\begin{array}{lll}
[\ \textsc{l} & \mapsto & \texttt{10} \\
,\ \textsc{init} & \mapsto & \texttt{0} \\
,\ \textsc{body} & \mapsto & \lambda\texttt{x} \to \texttt{x-2} \\
]
\end{array}
$$

no renaming needed!

Higher-order matching, simplified semantics

$l \stackrel{?}{=} t \leadsto \sigma$

$$\frac{}{\_ \stackrel{?}{=} t \leadsto []}\ \textsc{Wild} \qquad \frac{l \stackrel{?}{=} t \leadsto \sigma}{\lambda v.l \stackrel{?}{=} \lambda v.t \leadsto \sigma}\ \textsc{Lam}$$

$$\frac{a = b \quad l_1 \stackrel{?}{=} t_1 \leadsto \sigma_1 \quad \ldots \quad l_n \stackrel{?}{=} t_n \leadsto \sigma_n}{a\ l_1 \ldots l_n \stackrel{?}{=} b\ t_1 \ldots t_n \leadsto concat(\sigma_1 \ldots \sigma_n)}\ \textsc{Atom}$$

$$\frac{freeVars(\lambda v_1 \ldots \lambda v_n.t) = \emptyset \quad distinct(v_1 \ldots v_n)}{M\ v_1 \ldots v_n \stackrel{?}{=} t \leadsto [M \mapsto \lambda v_1 \ldots \lambda v_n.t]}\ \textsc{Meta}$$

- l and t in η-long, β-short normal form
- α-renaming ignored here

Miller's pattern restriction

$$\frac{\mathit{freeVars}(\lambda v_1 \dots \lambda v_n.t) = \emptyset \quad \mathit{distinct}(v_1 \dots v_n)}{M\ v_1 \dots v_n \stackrel{?}{=} t \rightsquigarrow [M \mapsto \lambda v_1 \dots \lambda v_n.t]}\ \textsc{meta}$$

META only allows meta-variables applied to *distinct object variables*

- "Pattern restriction"
- Ensures efficient implementation and a most general solution
- No renaming needed

Our library caches free variables when rewriting bottom-up

- Same complexity as first-order rewriting!

Implementation

```
rule_for3 len init body =
    forLoop (mvar len) (mvar init) (λi s → body -$- i)
      ⟹
    cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))

term = forLoop 10 0 (λx y → x-2)
```

Three different data types:

- `LHS`: Left-hand sides of rules
- `RHS`: Right-hand sides of rules
- `Data`: EDSL expressions

Slightly different syntactic categories:

- `mvar` only allowed in `LHS` and `RHS`
- Wildcards only allowed in `LHS`
- Pattern restriction only applies to `LHS`

Implementation

Tagless embedding of language constructs (simplified):

```
forLoop :: (ForLoop r, Bind r)
        ⇒ r Int → r s → (Var r Int → Var r s → r s) → r s

instance ForLoop Data; instance Bind Data
instance ForLoop LHS;  instance Bind LHS
instance ForLoop RHS;  instance Bind RHS

mvar :: MetaVar r ⇒ Name → r a

instance MetaVar LHS
instance MetaVar RHS
```

- Pattern restriction almost captured in types
- More details, see paper

Summary

Smart constructors make creation of expressions

- Type-safe
- Scope-safe
- Abstract

... even if the representation does not have these properties.

Summary

Smart constructors make creation of expressions

- Type-safe
- Scope-safe
- Abstract

. . . even if the representation does not have these properties.

Now we can get the same benefits for pattern matching and rewriting.

`https://github.com/emilaxelsson/ho-rewriting`

Higher-order matching

First-order matching:

$$l \stackrel{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!] l = t$$

- Matching followed by substitution gives back the original term
- Syntactic matching

Higher-order matching

First-order matching:

$$l \stackrel{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!] l = t$$

- Matching followed by substitution gives back the original term
- Syntactic matching

Higher-order matching:

$$l \stackrel{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!] l \equiv_{\alpha,\beta,\eta} t$$

- Matching followed by substitution gives back a term that is α, β, η-equivalent to the original term
- Semantic matching

Future work

- More traversal strategies
  - May be more costly
- Conditional rules
- Use in Feldspar
  - Combine with “low-level” rules where needed