A Mechanized Theory of Quoted Code Patterns

Radosław Waśko

June 18, 2020

$\lambda^{◑}$ is an extension of *simply typed lambda calculus* that adds splices, quotes and quoted pattern matching. It formalizes quoted pattern matching which is being added in Scala 3.

The goal of this semester project was to create mechanized proofs of soundness of that calculus, based on the paper proofs in the original paper. The project consists of 1366 lines of Coq code, of which 585 are the proofs and 455 are definitions.

Overview

1. $\lambda^{◑}$calculus
2. De Bruijn indices
   - Multiple binders in one pattern
3. Proving soundness
4. Lessons learned

λ°example - compared with Scala

```
def f(e: Expr[Int]): Expr[Int] =
  e match {
    case '{ add(0, $y) } => y
    case _ => e
  }

f(${add(0, 2)})
```

which evaluates to ${2} corresponds to

$$f = \lambda e{:}\square Nat.e \sim \Big(add\ 0\ (\mathtt{bind}[Nat]\ y)\Big)\ ?\ y \parallel e$$

$$\Big(f\ \ \square(add\ 0\ 2)\Big) \longrightarrow \square(2)$$

$\lambda^{◑}$syntax

$$
\begin{array}{lcl}
t & ::= & u{:}T \\
u & ::= & n \mid x \mid \lambda x{:}T.t \mid t\ t \mid \mathtt{fix}\ t \mid \Box\ t \mid \$\ t \mid \mathtt{lift}\ t \mid t \sim p\ ?\ t \parallel t \\
p & ::= & n \mid x \mid p\ p \mid \mathtt{fix}\ p \mid \mathtt{unlift}\ x \mid \mathtt{bind}[T]\ x \mid \mathtt{lam}[T]\ x \\
T & ::= & Nat \mid T \rightarrow T \mid \Box T
\end{array}
$$

Definitions from the paper *A Theory of Quoted Code Patterns*

$\lambda^{◑}$- quotes and splices

$$\frac{\Gamma \vdash^{0} t \in Nat}{\Gamma \vdash^{0} (\texttt{lift}\ t){:}\Box Nat \ \in\ \Box Nat}\ (\textsc{T-Lift})$$

$$\frac{\Gamma \vdash^{1} t \in T}{\Gamma \vdash^{0} (\Box\ t){:}\Box T \ \in\ \Box T}\quad (\textsc{T-Box})$$

$$\frac{\Gamma \vdash^{0} t \in \Box T}{\Gamma \vdash^{1} (\$\ t){:}T \ \in\ T}\quad (\textsc{T-Unbox})$$

$$(\$\ \Box\ \hat{t}\ ){:}T \longrightarrow^{1} \hat{t}{:}T \qquad (\textsc{E-Splice})$$

$$(\texttt{lift}\ n){:}T \longrightarrow^{0} (\Box\ n){:}T\ (\textsc{E-Lift-Red})$$

$\lambda^{◑}$- un-nesting patterns

$$
\begin{array}{rcl}
\texttt{MatchNat}\ t_1\ n\ t_2\ t_3 & \equiv & t_1 \sim n\ ?\ t_2 \parallel t_3 \\
\texttt{MatchVar}\ t_1\ x\ t_2\ t_3 & \equiv & t_1 \sim x\ ?\ t_2 \parallel t_3 \\
\texttt{MatchApp}\ t_1\ T_1\ T_2\ b_0\ b_1\ t_2\ t_3 & \equiv & t_1 \sim (\texttt{bind}[T_1]\ b_0)\ (\texttt{bind}[T_1 \to T_2]\ b_1)\ ?\ t_2 \parallel t_3 \\
\texttt{MatchUnlift}\ t_1\ b_0\ t_2\ t_3 & \equiv & t_1 \sim \texttt{unlift}\ b_0\ ?\ t_2 \parallel t_3 \\
\texttt{MatchLam}\ t_1\ (T_1 \to T_2)\ b_0\ t_2\ t_3 & \equiv & t_1 \sim \texttt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3 \\
\texttt{MatchFix}\ (t_1{:}T_1)\ b_0\ t_2\ t_3 & \equiv & (t_1{:}T_1) \sim \texttt{fix}\ (\texttt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3
\end{array}
$$

$\lambda^{◑}$- patterns

Originally — 1 base rule for pattern match and rules for each pattern:

$$\frac{\Gamma \vdash^0 t_1 \in \Box T_1 \qquad \Gamma \vdash_p p \in T_1 \rightsquigarrow \Gamma_p \qquad \Gamma;\Gamma_p \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim p \;?\; t_2 \;\|\; t_3){:}T \;\in\; T} \quad \text{(T-Pat)}$$

$$\Gamma \vdash_p \texttt{unlift}\ x \in Nat \rightsquigarrow \{x^0{:}Nat\} \qquad \text{(T-Pat-Unlift)}$$

Simplified — separate rule for each pattern match:

$$\frac{\Gamma \vdash^0 t_1 \in \Box Nat \qquad \Gamma;\, b_0^0 : Nat \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim \texttt{unlift}\ b_0 \;?\; t_2 \;\|\; t_3){:}T \;\in\; T} \quad \text{(T-Pat-Unlift)}$$

De Bruijn indices

To simplify the α-equivalence relation and definition of substitution, instead of using normal names, we represent variables using De Bruijn indices
The index specifies how many binders we have to skip (in the syntax tree) to reach the one we are bound to.

$$\lambda x. x \implies \lambda. \#0$$

$$\lambda x. \lambda y. x \implies \lambda. \lambda. \#1$$

De Bruijn indices - free variables

Indices greater than the number of binders surrounding it represent the free variables.

$$f{:}T_1;\ g{:}T_2 \vdash \lambda x.\ \lambda y.\ f\ x\ y$$
$$T_1;\ T_2 \vdash \lambda.\ \lambda.\ \#3\ \#1\ \#0$$

Beta-reduction

$(\lambda x.\, t)\ v \quad \longrightarrow \quad t[x \mapsto v]$

becomes

$(\lambda.\, t)\ v \quad \longrightarrow \quad t[v/]$

$$T_2; T_1 \vdash (\lambda.\ \#0\ \#1)\ \#1$$
$$T_2; T_1 \vdash (\#0\ \#1)[\#1/]$$
$$T_2; T_1 \vdash \#1\ \#0$$

Beta-reduction

$$(\lambda.\, t)\ v \quad \longrightarrow \quad t[v/]$$

$$
\begin{aligned}
&T_2; T_1 \vdash (\lambda.\, \lambda.\, \#1)\ \#1 \\
&T_2; T_1 \vdash (\lambda.\, \#1)[\#0 \mapsto \#1/] \\
&T_2; T_1 \vdash \lambda.\, (\#1)[\#1 \mapsto \texttt{shift}\ \ \#1/] \\
&T_2; T_1 \vdash \lambda.\, (\#1)[\#1 \mapsto \#2/] \\
&T_2; T_1 \vdash \lambda. \#2
\end{aligned}
$$

Multiple binders in one pattern

As an example: unpacking a tuple

$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ t \quad \longrightarrow \quad t[x_1 \mapsto v_1, x_2 \mapsto v_2]$

Multiple binders in one pattern

As an example: unpacking a tuple

$$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ t \quad \longrightarrow \quad t[x_1 \mapsto v_1, x_2 \mapsto v_2]$$

$$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ t \quad \rightarrow \quad ?$$

Multiple binders in one pattern

As an example: unpacking a tuple

$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ t \quad \longrightarrow \quad t[x_1 \mapsto v_1, x_2 \mapsto v_2]$

$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ t \quad \xrightarrow{?} \quad ((t)[v_2/])[v_1/]$ Wrong

Why?

$$T_2; T_1 \vdash \texttt{unpack}\ (\#1, \#0)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ (\#1\ \#0)$$
$$T_2; T_1 \vdash ((\#1\ \#0)[\#0/])[\#1/]$$
$$T_2; T_1 \vdash (\#0\ \#0)[\#1/]$$
$$T_2; T_1 \vdash (\#1\ \#1)$$

The red #0 is now bound to the purple binder, so it could be written as $\#0$, but we would expect it to stay $\#0$.

Multiple binders in one pattern

As an example: unpacking a tuple

$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ t \quad \longrightarrow \quad t[x_1 \mapsto v_1, x_2 \mapsto v_2]$

We need to use the `shift` operation:

$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ t \quad \longrightarrow \quad ((t)[\texttt{shift}\ v_2/])[v_1/]$

$$T_2; T_1 \vdash \texttt{unpack}\ (\#1, \#0)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ (\#1\ \#0)$$
$$T_2; T_1 \vdash ((\#1\ \#0)[(\texttt{shift}\ \#0)/])[\#1/]$$
$$T_2; T_1 \vdash ((\#1\ \#0)[\#1/])[\#1/]$$
$$T_2; T_1 \vdash (\#0\ \#1)[\#1/]$$
$$T_2; T_1 \vdash (\#1\ \#0)$$

Proving soundness - progress

Theorem (Progress)

If $\emptyset \vdash^0 t \in T$, then t is a value or there exists t' such $t \longrightarrow^0 t'$

Proving soundness - progress

Lemma (Level Progress)

For any given term t, we have:

(1) *If $\Gamma^{[1]} \vdash^{0} t \in T$, then t is a value or there exists t' such that $t \longrightarrow^{0} t'$.*

(2) *If $\Gamma^{[1]} \vdash^{1} t \in T$ and $(\Box t) : \Box T$ is not a value, then there exists t' such that $t \longrightarrow^{1} t'$.*

where $\Gamma^{[1]}$ means that the environment only contains level 1 variables.

Proving soundness - preservation

Theorem (Preservation)

If $\Gamma \vdash^i t \in T$ *and* $t \longrightarrow^i t'$, *then* $\Gamma \vdash^i t' \in T$.

Proving soundness - preservation

Lemma (Substitution)

If

(1) $\Gamma \vdash^{j} t_1 \in T_1$,

(2) $\Gamma, x^{j} : T_1 \vdash^{i} t_2 \in T_2$ *and*

(3) $j = 0$ *or* t_2 *does not contain pattern matches,*

then $\Gamma \vdash^{i} t_2[x \mapsto t_1] \in T_2$.

Why the third assumption? $x^1 : T \vdash (\Box\ x) \sim x\ ?\ e_1 \parallel e_2$

Lessons learned

- ‘unit’ testing before starting proofs

Lessons learned

- ‘unit’ testing before starting proofs
- iterative development

Lessons learned

- ‘unit’ testing before starting proofs
- iterative development
- notations

```
Theorem Preservation : ∀ t₁ T G L,
    G ⊢(L) t₁ ∈ T →
    ∀ t₂,
    t₁ →(L) t₂ →
    G ⊢(L) t₂ ∈ T.
```

Lessons learned

- proof stability
    - predictable names

Prefer `assert (`*Hypothesis*`) as HypX.`
instead of just `assert (`*Hypothesis*`).`
`intro Ht1typ Hreduct.` instead of `intros.` if the hypothesis names are then used somewhere explicitly.
etc.

Lessons learned

- proof stability
  - predictable names
  - tactics using pattern matching to find right hypothesis regardless of name

```
Ltac invV :=
  match goal with
  | H: ?G ⊢(L0) ?v ∈ □(?T) |- _ => inversion H; subst
  end.
```

which matches for example: `H3: G ⊢(L0) (Quote t : T1) ∈ □`*Nat*.
We can then replace

```
destruct typing_judgement.
- inversion H2.
- inversion H4.
... (* many more branches *)
```

by just `destruct typing_judgement; invV.`

Thank you :)

Questions?