A Mechanized Theory of Quoted Code Patterns

Radosław Waśko

June 18, 2020

A Mechanized Theory of Quoted Code Patterns
Radosław Waśko
June 18, 2020

The project was supervised by Fengyun Liu and Nicolas Stucki.

$\lambda^{◑}$ 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.

$\lambda^{◑}$ 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

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

- We will first shortly describe the calculus
- then we will introduce De Bruijn indices which were one of the main challenges of the formalization and discuss an interesting example.
- then we will see an overview of the soundness proofs
- Finally we'll discuss some lessons that I learned along the way.
- 3:00

$\lambda^{◑}$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

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

$$\left(f\ \ \square(add\ 0\ 2) \right) \longrightarrow \square(2)$$

- First to understand better the calculus we will compare it with Scala
- Above we can see a simple function that takes a code value representing some computation and if that computation represents adding 0 to some other value ‘y’, we simplify it to just this second value
- we achieve that by pattern matchin on the code value and matching it against an application of the add function
- the code that is applied as the second argument is bound as a code value ‘y’ that can be than further analysed or returned
- In the example, we simplify code computing the addition $0 + 2$ into just the code value representing number two.
- below we see the analogous program in the calculus, the idea is very similar but for the syntax
- the tilde stands for match *e* against some pattern, after the question mark is the code that is evaluated on match success and can refer to new bindings introduced in the pattern
- after the double bars is the alternative that is executed if the match failed (4:30)

$\lambda^{◑}$syntax

$$
\begin{array}{lll}
t & ::= & u{:}T \\
u & ::= & n \,|\, x \,|\, \lambda x{:}T.t \,|\, t\ t \,|\, \texttt{fix}\ t \,|\, \square\ t \,|\, \$\ t \,|\, \texttt{lift}\ t \,|\, t \sim p\ ?\ t \parallel t \\
p & ::= & n \,|\, x \,|\, p\ p \,|\, \texttt{fix}\ p \,|\, \texttt{unlift}\ x \,|\, \texttt{bind}[T]\ x \,|\, \texttt{lam}[T]\ x \\
T & ::= & Nat \,|\, T{\to}T \,|\, \square T
\end{array}
$$

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

$\lambda^{◑}$syntax

$t ::= u{:}T$
$u ::= n \,|\, x \,|\, \lambda x{:}T.t \,|\, t\ t \,|\, \texttt{fix}\ t \,|\, \square\ t \,|\, \$\ t \,|\, \texttt{lift}\ t \,|\, t \sim p\ ?\ t \parallel t$
$p ::= n \,|\, x \,|\, p\ p \,|\, \texttt{fix}\ p \,|\, \texttt{unlift}\ x \,|\, \texttt{bind}[T]\ x \,|\, \texttt{lam}[T]\ x$
$T ::= Nat \,|\, T{\to}T \,|\, \square T$

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

- the calculus has the standard STLC constructs like lambda abstraction, variables and application and is extended with a fixpoint operator and numbers
- then we have the square which is a quote operator - it represents the code value of the term inside it
- dual to it is the dollar which is a splice operator - it can be put inside a quoted term and it takes some code value and inserts into the quoted code the code that this value represents
- then there's the lift operator that allows to lift an evaluated numeric value to a code value representing that number and the pattern match that was explained before
- The values are numbers, lambda abstractions, and code values containing plain terms i.e. terms that don't contain any quotes, splices or pattern matches. (6:00)

$\lambda^{\LEFTcircle}$- quotes and splices

$$(\lambda e{:}\Box Nat.\Box\ (2 + \$\ e))\ \Box\ 3 \longrightarrow \Box\ (2 + \$\ \Box\ 3) \longrightarrow \Box\ (2 + 3)$$

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

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

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

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

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

- Here we show some of the typing and evaluation rules.
- The calculus has two levels: level 0 for the top level code that is actually evaluated and level 1 for the code level. The levels can be interleaved using the quote and splice operators - at level 0 we can use quote which inside has level 1 code. In level 1 code we can use a splice to refer to some level 0 code that can be evaluated and placed inside this code value replacing the splice.
- The splice evaluation rule shows this - once some code that is spliced is evaluated to some plain value (i.e. it cannot be itself further reduced), it can take the splice's place in that code value.
- almost all evaluation happens at level 0, as level 1 represents just code values. The only evaluation rules regarding level 1 are ones that allow to propagate evaluation into splices that are in the code value and the E-SPLICE rule that reduces the splice.
- Beta reduction and evaluating pattern matching is only allowed at level 0. (8:30)

$\lambda^{\RIGHTcircle}$- un-nesting patterns

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

$\lambda^{\RIGHTcircle}$- un-nesting patterns

- One of the changes we made to the calculus to make the mechanization easier was to remove nested patterns.
- As a result of that we also turned matching each pattern into a separate syntactic form as can be seen here.
- This of course doesn't change the expressivity of the calculus as we can easily convert a nested pattern match into a series of non-nested pattern matches.
- There were actually two patterns that allowed for nesting - matching an application and the fixpoint. So instead of allowing an arbitrary nested pattern we only allow the patterns to be bind, i.e. all things can be nested are just bound to some variables. If necessary they can be than pattern matched again to emulate a nested pattern.
- 9:30

λ◑- patterns

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

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

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

Simplified — separate rule for each pattern match:

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

λ◑- patterns
Originally — 1 base rule for pattern match and rules for each pattern:
Simplified — separate rule for each pattern match:

- after the simplification, each pattern type has a separate typing rule, so instead of handling an arbitrary number of bindings like in T-PAT, we statically know what bindings are introduced. This greatly simplifies mechanization.
- similarly, every pattern type gets 3 separate evaluation rules: for success, failure and one for head reduction
- 10:30

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
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.
$\lambda x.\, x \implies \lambda.\, \#0$
$\lambda x.\, \lambda y.\, x \implies \lambda.\, \lambda.\, \#1$

- In the mechanization we need to represent variable binding. The method usually used on paper - names, is not great for mechanization because we need to be very careful about alpha-equivalence.
- To simplify the α-equivalence relation and definition of substitution, we represent variables using De Bruijn indices
- Instead of a name we have an index that specifies how many binders we have to skip (in the syntax tree) to reach the one we are bound to.
- Thus alpha equivalence becomes syntactic equivalence and it is easier to define capture avoiding substitution.
- 13:00

De Bruijn indices - free variables

Variables in the environment are differentiated by their order.

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

De Bruijn indices - free variables

Variables in the environment are differentiated by their order.

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

- We treat variables in the environment as binders and identify them by their order, since they are not named.
- The f that refers to T_1 has to skip two normal binders and one environment variable so in total it gets index 3.

Beta-reduction

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

becomes

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

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

- To illustrate, in beta reduction, we no longer have a name of the substituted variable.
- So our substitution operation just replaces the variables bound to the closest binder (that is index 0) and decreases all other indices by 1 (to indicate that with this substitution we have removed one external binder)

Beta-reduction

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

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

Beta-reduction
$(\lambda.\, t)\ v \longrightarrow t[v/]$
$T_2; T_1 \vdash (\lambda.\, \lambda.\, \#1)\ \#1$
$T_2; T_1 \vdash (\lambda.\, \#1)[\#0 \mapsto \#1/]$
$T_2; T_1 \vdash \lambda.\Big((\#1)[\#1 \mapsto \texttt{shift}\ \#1/]\Big)$
$T_2; T_1 \vdash \lambda.\Big((\#1)[\#1 \mapsto \#2/]\Big)$
$T_2; T_1 \vdash \lambda.\#2$

- When substituting into more complex terms, we need to keep the indices referring to the original binders.
- The `shift` function increases all indices in a term.
- It is used when we are substituting a term and enter a binder that we need to skip to keep the references up-to-date.
- 15:00

Multiple binders in one pattern

As an example: unpacking a tuple

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

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

Multiple binders in one pattern
As an example: unpacking a tuple
$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ t \longrightarrow t[x_1 \mapsto v_1, x_2 \mapsto v_2]$
$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ t$

- Now, I wanted to describe what I think is an interesting lesson of using De Bruijn indices in a bit more complicated settings.
- The application pattern binds two variables, so when evaluating it we need to bind two variables at one. To make the example more approachable, instead let's consider a similar issue when unpacking a tuple - we need to bind both elements at once.

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
unpack (v_1, v_2) as (x_1, x_2) in $t \longrightarrow t[x_1 \mapsto v_1, x_2 \mapsto v_2]$
unpack (v_1, v_2) as $(\bullet, \bullet)$ in $t \rightarrow ?$

- First, when using De Bruijn indices, we don't have the names - so we have to remove x_1 and x_2, instead we can just say that the closest binder binds the second element (as it is the closest one), and the second index binds the first tuple element.

Multiple binders in one pattern

As an example: unpacking a tuple

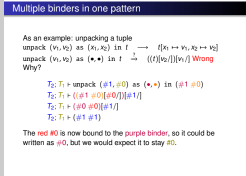
$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (x_1, x_2)\ \texttt{in}\ t \quad \longrightarrow \quad t[x_1 \mapsto v_1, x_2 \mapsto v_2]$

$\texttt{unpack}\ (v_1, v_2)\ \texttt{as}\ (\bullet, \bullet)\ \texttt{in}\ t \quad \overset{?}{\to} \quad ((t)[v_2/])[v_1/]$ 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 $\#0$.



- What will be the evaluation rule? The first guess is to just replace the elements one by one. But this leads to an error - the green/olive #0 is substituted for the orange one, but later the result is again substituted. So that #0 is actually inside the scope of the second binder, but the substituted value was not updated. This breaks the references.
- Instead, to make up for v_2 being inside scope of the second substitution, we need to `shift` it to keep indices up-to-date. Now we can see that when we shift the term it evaluates correctly - the #0 becomes a #1 inside of the binder and is later decremented back to #0 upon the second substitution. All the time it refers to the correct variable from the environment.
- 19:00

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:

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

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

- What will be the evaluation rule? The first guess is to just replace the elements one by one. But this leads to an error - the green/olive #0 is substituted for the orange one, but later the result is again substituted. So that #0 is actually inside the scope of the second binder, but the substituted value was not updated. This breaks the references.
- Instead, to make up for v_2 being inside scope of the second substitution, we need to `shift` it to keep indices up-to-date. Now we can see that when we shift the term it evaluates correctly - the #0 becomes a #1 inside of the binder and is later decremented back to #0 upon the second substitution. All the time it refers to the correct variable from the environment.
- 19:00

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

- The progress theorem is standard, but getting there is less so.
- We use a lemma that characterizes progress on both levels. The progress theorem is a directo corollary of this lemma.

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 - 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.

- The progress theorem is standard, but getting there is less so.
- We use a lemma that characterizes progress on both levels. The progress theorem is a directo corollary of this lemma.
- At level 0 we mimic the progress theorem, but instead of an empty environment we allow the environemnt to contain level 1 variables. That is because when evaluating level 0 code nested inside of splices, as level 1 abstractions are not reduced, the variables introduced by them may be in scope.
- At level 1 we say that if the term inside the code value is not plain, i.e. it contains some splices, it can be further reduced.
- The two parts of the lemma depend on each other as to define reduction of splices inside level 1 code we need reduction at level 0 and vice-versa.

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
Theorem (Preservation)
If $\Gamma \vdash^i t \in T$ and $t \longrightarrow^i t'$, then $\Gamma \vdash^i t' \in T$.

- The preservation theorem is also rather standard.
- The substitution lemma that is used in it is however a bit more interesting.

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

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

- In the substitution lemma, the third assumption is particularly interesting. It states that we can substitute a level 1 variable only if the term that we subsitute into doesn't contain any pattern matches.
- This is necessary to ensure that we don't replace a variable inside the pattern matching against a variable with something else.
- This assumption is ok, because most of the substitutions are at level 0 (beta reduction or pattern matching both introduce variables at level 0).
- There is only one case where we are substituting a level 1 variable that is *matching a lambda* and in there we know that the term that we substitute into is plain, so it can't contain a pattern match.
- 23:00

Lessons learned

- ‘unit’ testing before starting proofs

Lessons learned
- ‘unit’ testing before starting proofs

- now I wanted to discuss some lessons that I learned during the project
- First one is that it is a good idea to write unit tests - for example use the definition of the calculus in Coq to write some simple terms in that calculus and proving that they typecheck and evaluate as expected. This allows to catch errors in the definitions that may be harder to spot deep inside a proof. Moreover, it serves as nice examples of the calculus.
- We choose to develop the project iteratively. That is, I started with simply typed lambda calculus, then extended it with splices and quotes and only then I added the pattern matching (which was the hardest part) and at last the fixpoint operator.
- This had a downside that I missed some requirements that the pattern matching imposed on the definitions (for example multiple binders) and had to change the library after encoding the simple types. On the other hand this approach offered more confidence that even if I wasn’t able to finish the final proof, I was able to deliver something that would still be a reasonable outcome.
- Using notations and an editor that supports ligatures was a rather cosmetic choice but it allowed to make the theorems look almost like the ones on paper. Making assumptions readable was very helpful when I had to look at lots of them.

Lessons learned

- 'unit' testing before starting proofs
- iterative development

└Lessons learned

Lessons learned

- 'unit' testing before starting proofs
- iterative development

- now I wanted to discuss some lessons that I learned during the project
- First one is that it is a good idea to write unit tests - for example use the definition of the calculus in Coq to write some simple terms in that calculus and proving that they typecheck and evaluate as expected. This allows to catch errors in the definitions that may be harder to spot deep inside a proof. Moreover, it serves as nice examples of the calculus.
- We choose to develop the project iteratively. That is, I started with simply typed lambda calculus, then extended it with splices and quotes and only then I added the pattern matching (which was the hardest part) and at last the fixpoint operator.
- This had a downside that I missed some requirements that the pattern matching imposed on the definitions (for example multiple binders) and had to change the library after encoding the simple types. On the other hand this approach offered more confidence that even if I wasn't able to finish the final proof, I was able to deliver something that would still be a reasonable outcome.
- Using notations and an editor that supports ligatures was a rather cosmetic choice but it allowed to make the theorems look almost like the ones on paper. Making assumptions readable was very helpful when I had to look at lots of them.

Lessons learned

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

$$\begin{array}{l}\textbf{Theorem Preservation} : \forall\, t_1\ T\ G\ L,\\ \quad G \vdash(L)\ t_1 \in T \to \\ \quad \forall\, t_2,\\ \quad t_1 \longrightarrow(L)\ t_2 \to \\ \quad G \vdash(L)\ t_2 \in T.\end{array}$$

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.

- now I wanted to discuss some lessons that I learned during the project
- First one is that it is a good idea to write unit tests - for example use the definition of the calculus in Coq to write some simple terms in that calculus and proving that they typecheck and evaluate as expected. This allows to catch errors in the definitions that may be harder to spot deep inside a proof. Moreover, it serves as nice examples of the calculus.
- We choose to develop the project iteratively. That is, I started with simply typed lambda calculus, then extended it with splices and quotes and only then I added the pattern matching (which was the hardest part) and at last the fixpoint operator.
- This had a downside that I missed some requirements that the pattern matching imposed on the definitions (for example multiple binders) and had to change the library after encoding the simple types. On the other hand this approach offered more confidence that even if I wasn’t able to finish the final proof, I was able to deliver something that would still be a reasonable outcome.
- Using notations and an editor that supports ligatures was a rather cosmetic choice but it allowed to make the theorems look almost like the ones on paper. Making assumptions readable was very helpful when I had to look at lots of them.

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
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.

- By default Coq names the hypotheses with increasing numbers. If definitions are changed, proofs using hypotheses by name tend to break, because the numbers shift. So it turned out it is good to make the names explicit wherever possible, to make the proofs more robust to irrelevant updates.

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.`

- In practice, we couldn't always make the names stable. For example when calling inversion on the typing judgement which had more than 10 possible cases it is unreasonable to name them all by hand.
- But we often needed to find some particular hypothesis, like the one shown here stating that some term types to some code value type.
- In this scenario writing tactics that find the right hypothesis by pattern matching was extremely useful. It often allowed to shrink a case-by-case proof that differed only by the name of the hypothesis to a proof that just called this tactic on each branch.

Thank you :)

Questions?

Thank you :)
Questions?