A Mechanized Theory of Quoted Patterns

Radosław Waśko

June 10, 2020

Abstract

The pattern matching on code from the new macro system of Scala 3 is modelled by a calculus called $\lambda^{\blacklozenge}$. We present a mechanized proof of soundness of the calculus in Coq and discuss encountered challenges.

1 Introduction

Scala 3 is getting a new macro system based on staged computation [1] which also supports pattern matching over code values. The theory behind the staged computation is described in the paper *A Theory of Quoted Code Patterns* [2].

For example, one can quote an expression and pattern match over it to transform it [1]

```
val code = '{ println(1+2) }
code match {
  case '{ println($x)) } =>
    '{ println("Result of " + ${Expr(x.show)} + ": " + $x) }
}
```

The code above will print: `Result of 1+2: 3`.

The language features are modelled by $\lambda^{\blacklozenge}$calculus which is an extension of *simply typed lambda calculus* with quotes, splices and pattern matching over quoted code values. The calculus is explained in more detail in section 2.

The goal of this project was to create a mechanized proof of the soundness of $\lambda^{\blacklozenge}$, based on the paper proofs in the original paper. The project consists of 1366 lines of Coq code[2], of which 585 are proofs and 455 are definitions. The definitions are explained in section 3 and the proofs in section 4.

While working on the proofs, we learned some lessons that may be useful in future work. We discuss these in section 5.

2 $\lambda^{\blacklozenge}$calculus

The system is formalized in two parts — there is an *implicitly-typed* language and an *explicitly-typed* variant in which every term is ascribed with its type. There are elaboration rules for converting *implicitly-typed* terms into *explicitly-typed*.

There are three kinds of types: a base type for natural numbers (*Nat*), a function type ($T_1 \to T_2$) and a type for code values ($\square T$).

The syntax contains the usual constructs from STLC: lambda abstractions, applications and variables. Moreover, it has natural number constants and a fixpoint operator to support recursion.

It is then extended with quote ($\square e$) and splice ($\$e$) operators typical for staged computation, a `lift` operator to lift a numeric value to a code value returning that number, and a simple pattern match construct

[1] Example from [2]

[2] The code can be found at https://github.com/radeusgd/QuotedPatternMatchingProof/

$(e \sim p \ ?\ e_{succ} \parallel e_{fail})$. The pattern match checks if e matches the pattern p: if it does, the expression reduces to e_{succ} where any bindings introduced by the pattern are available, otherwise it reduces to e_{fail}.

A non-reducible value in this calculus can be a natural number, a lambda or a quoted plain code value.

A plain term is defined as the non-staged subset of the calculus — lambdas, abstractions, constants and fixpoint expressions. Quotes, splices and pattern matching are excluded in plain terms.

Nesting of quotes and splices is regulated by *levels* - level 0 terms are the top-level terms that are executed and level 1 terms are boxed code (which can also contain level 0 terms by splicing). For simplicity, the calculus only supports two levels (so it is not possible to have code values containing code values).

Below is an example of a term in the calculus. The code implements a function that matches application of a function taking a number and replaces its argument with 42. If the match fails the original code is returned.

$$\left(\lambda e{:}\Box Nat.e \sim \left(\texttt{bind}[Nat \to Nat]\ f\ \texttt{bind}[Nat]\ x\right)\ ?\ \Box(\$f\ 42) \parallel e\right)\ \Box((\lambda x{:}Nat.x)\ 10)$$

This term will match the suspended function application and replace the argument, thus it reduces to $\Box((\lambda x{:}Nat.x)\ 42)$.

3 Definitions in Coq

We define the *explicitly-typed* language, as the safety properties are defined for it and the elaboration of the *implicitly-typed* language is quite simple.

Defining the calculus in Coq requires some simplifications. The simplifications don't limit the expressivity of the language.

3.1 Variable Binding

Variable binding is a common issue in the mechanization of calculi.

On paper we usually use textual names and define a capture-avoiding substitution operation. This is possible in Coq, but this kind of function is not so easy to define. We need to apply recursively on terms that are not structurally sub-terms of the main terms (because sometimes they may be renamed etc.). As Coq only accepts total functions, we need to show that the invocations are indeed on terms smaller in some way. Moreover, the concept of α*-equivalence* becomes much more complicated because one has to define properties in terms of equivalence classes of terms. This highly complicates the definitions and also makes it harder to reason about theorems.

3.1.1 De Bruijn indices

The usual solution to this problem is using De Bruijn indices [3]. In a term encoded using De Bruijn indices, variable names are replaced by indices — numbers that indicate *binder distance* (how many binders are in scope in the expression tree between the variable and the binder that this variable is referring to). In such notation, $\alpha-$equivalence is reduced to syntactic equality and defining capture avoiding substitution is much simpler.

For example, the identity function $\lambda x.\, x$ is encoded as $\lambda.\, \#0$ and the K combinator $\lambda x.\, \lambda y.\, x$ becomes $\lambda.\, \lambda.\, \#1$ [3].

An index greater than the number of binders that it is surrounded by refers to free variables. We can list the free variables in form of an environment next to the term, so that it is clearer what those indices refer to. In the example below, we compare a term using names and the same term with De Bruijn indices in which variables are colored accordingly to which binder (or free variable) they refer to:

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

[3] We will sometimes use colors to make it easier to see which index refers to which binder.

It is important to note, that the same index can point to different variables and the same variable can be pointed to by different indices — the index depends heavily on the context (as the *binder distance* is different in different contexts), as shown in the example below:

$$f{:}T_1 \vdash \lambda x.\ (\lambda y.\ f\ x\ y)\ x$$
$$T_1 \vdash \lambda.\ \ (\lambda.\ \#2\ \#1\ \#0)\ \#0$$

During substitution, it is important to keep track of the index. When traversing inside of a binder, free variables in the substitute term have to be updated to still refer to the same variable. For example, given the term $(\lambda.\lambda.\#1)\ (\lambda.\#2)$, where the index #2 refers to the second free variable, after β-reduction, we get $\lambda.\lambda.\#3$ — the substitute term has to be shifted to still refer to the second free variable (as it is now behind not one but two binders).

3.1.2 *DbLib*

As handling variable binding is common to nearly all calculi formalizations, there are some Coq libraries that help automate parts of reasoning about them. In this project we are using the *DbLib* library [4], some other choices are discussed in section 5.3. When using *DbLib* we freely define our syntactic forms as some inductive type. We have two mutually inductive types - typed terms (`typedterm`) and untyped terms (`term`), and we can substitute an untyped term into both of these types. To use the automation facilities provided by the library, we need to define:

- an instance of the typeclass `Var` which is used for constructing variables - it requires a function `var` that takes a De Bruijn index (which is simply a natural number) and returns an instance of the type that we will be substituting (in our case `term`); for us the instance is just calling the constructor `var x = VAR x`[4]
- instances of the typeclass `Traverse` which requires a function `traverse` taking 3 parameters: a function `f` representing the operation that is applied to the term-tree, a natural number `l` representing the current level and a term `t` that is traversed. It has to recursively traverse all subterms in `t`, increasing `l` when entering a binder, and each encountered instance of a variable `VAR x` has to be replaced with `f l x`.

Afterwards some lemmas have to be proven to show that the `traverse` function satisfies the required constraints. For example it has to be injective. If the function is not overly complex, these properties are derived automatically by tactics provided by the library.

Once this is defined, substitution, shift etc. are automatically derived from the `traverse` function.

3.2 Patterns

Defining patterns is particularly complicated as the patterns can be arbitrarily nested, so a pattern match could introduce an arbitrary amount of bindings. This is possible to handle, but is much harder to mechanize. To simplify the mechanization, we forbid nested patterns. This does not limit the expressivity of the calculus as any nested pattern match can be converted to a series of non-nested patterns.

For example, we can convert

$$e \sim ((\texttt{lam}[Nat \to Nat]\ f)\ 42)\ ?\ f\ \square(10)\ \|\ e_{fail}$$

into

$$e \sim (\texttt{MatchApp}\ [Nat \to Nat]\ a_1\ [Nat]\ a_2)\ ?$$
$$\left(a_1 \sim (\texttt{MatchLam}[Nat \to Nat]\ f)\ ?\ \left(a_2 \sim \texttt{MatchNat}\ 42\ ?\ (f\ \square(10))\ \|\ e_{fail}\right)\ \|\ e_{fail}\right)\ \|\ e_{fail}$$

(where `MatchApp` pattern is a special pattern which matches an application and binds the first and second term to a_1 and a_2 respectively).

[4] The constructor is capitalized (`VAR`) instead of just `Var` to avoid name clash with the mentioned typeclass.

Even after the simplification, the App pattern still introduces two variable bindings, so we need to be able to support it. This is still much simpler than the arbitrary amount of bindings that we started with.

We first tried to define the patterns as a separate inductive definition and have a single syntactic form for the pattern matching:

```
Inductive pattrn :=
| PNat (n : nat)
| PVar (x : DeBruijnIndex)
| PBindUnlift
| PBindApp (T1 T2 : type)
| PBindLam (T : type)
| PBindFix.

Inductive typedterm :=
| TypedTerm (u: term) (t: type)
with term :=
| Nat (n : nat)
| Var (x : DeBruijnIndex)
| Lam (argT: type) (ebody : typedterm)
| App (e1 e2 : typedterm)
| Fix (e : typedterm)
| Lift (e : typedterm)
| PatternMatch (e : typedterm) (pat : pattrn) (success failure : typedterm).
```

Unfortunately this gives rise to 3 syntactic forms (untyped terms, typed terms and patterns) and it seemed to be too much for *DbLib* — the automatic facilities that help us prove the basic lemmas to make the library usable failed and I was also unable to fix the proofs manually.

So in the end we created a separate syntactic form for each pattern. This may look strange, but it is in fact equivalent to the previous formulation, the difference is only technical / notational.

```
Inductive typedterm :=
| TypedTerm (u: term) (t: type)
with term :=
| Nat (n : nat)
| VAR (x : DeBruijnIndex)
| Lam (argT: type) (ebody : typedterm)
| App (e1 e2 : typedterm)
| Fix (e : typedterm)
| Lift (e: typedterm)
| Quote (e : typedterm)
| Splice (e : typedterm)
| MatchNat (e : typedterm) (n : nat) (es ef : typedterm)
| MatchVar (e : typedterm) (x : term) (es ef : typedterm)
| MatchApp (e : typedterm) (T1 T2 : type) (es ef : typedterm)
| MatchUnlift (e : typedterm) (es ef : typedterm)
| MatchLam (e : typedterm) (T : type) (es ef : typedterm)
| MatchFix (e : typedterm) (T : type) (es ef : typedterm).
```

One other modification that we have to apply here is to the variable matching pattern. This pattern checks if the boxed code is a reference to some variable, so after the De Bruijn translation, this pattern also contains a De Bruijn index for the variable that is matched (see `PVar` in the first listing). This occurrence is unusual, because we need to update it when shifting all indices in a term, so that it always refers to the original variable (why we have to shift is explained in the previous section). It has to be treated differently, because it should never be substituted with some concrete value.

However in the library that we use, the `traverse` function has type `(nat → nat → term) → (nat → T → T)` (where `T` is either a `term` or `typedterm`). `traverse f l t` traverses the term `t`, incrementing `l`

when entering binders and replacing a variable with index `x` with the result of `f l x`. As seen in the type signature, result of `f l x` can be any `term`. In theory the variable in the pattern could be substituted with something else. We later prove that this is not possible, by proving *Preservation*, but we cannot prove this directly within the definition.

Instead, we replace the raw De Bruijn index in `MatchVar` with an arbitrary term. This satisfies all the constraints required by the library and ensures that this index is updated when necessary. Unfortunately this also allows for terms that are ill-formed syntactically, (variable match over something other than a variable). This issue is however addressed in the typing rules — only instances of `MatchVar` in which `x` has the form `VAR y` are accepted. Thus, when we prove Preservation, we also prove that this variable is never substituted for (as the term resulting from such substitution would be ill-typed).

3.3 Semantics and Typing

The typing judgement is defined as an inductive type `TypEnv → Level → typedterm → type → Prop` and the small-step semantics relation is defined as `Level → typedterm → typedterm → Prop`.

We use Coq's Notations to make our judgements more readable. The typing judgement is expressed as `G ⊢(L) e ∈ T` and the semantics relation as `e1 -->(L) e2` which quite closely resembles the notation used in the paper.

We also define a relation `evaluates` which is the reflexive transitive closure of small-step semantics at level 0 that is used in tests. For example we can define a term representing applying the identity function to a number and prove that it typechecks and evaluates correctly:

```
Definition id_nat := (Lam TNat (VAR 0 : TNat) : TNat ==> TNat).
Definition id_applied := (App id_nat (Nat 42 : TNat) : TNat).
Lemma id_app_types : ∅ ⊢(L0) id_applied ∈ TNat.
  repeat econstructor.
Qed.
Lemma id_app_evals : evaluates id_applied (Nat 42 : TNat).
  eapply star_step. repeat econstructor.
  unfold substitute. simpl_subst_all. auto.
Qed.
```

Another important change from the paper version is the typing environment. On paper it is defined as a (partial) mapping from variable names to types and levels. *DbLib* is also providing support for using environments with De Bruijn indices. We used the approach suggested by the library authors — our environment is represented by a partial mapping from natural numbers to types. It is defined as a list of options. This integrates very well with operations on De Bruijn indices. For example, when the environment is extended while going inside a binder, the new type is just prepended to the list. This both handles the new type and the fact that other indices are shifted.

Our definitions always extend the environment without leaving holes, so we could have used lists instead lists of options. But we choose the latter, because the *DbLib* library provides some helpful tactics for handling environments defined that way. This also simplifies the definition of the *Weakening* and *Substitution* lemmas, as we can use an operation that inserts a variable to an environment with an arbitrary index, possibly leaving holes.

As our typing environment has to keep track of level of each variable, the final `TypEnv` type is `list (option (level * type))`. The judgement from the original paper $\Gamma(x^i) = T$ is expressed as `lookup` x Γ `= Some (L`i`,` T`)`.

Extending the environment, i.e. $\Gamma; x^i : T$ is expressed as `insert x (Li, T)` Γ, where the new `x` is the De Bruijn index corresponding to x. In the typing rules we only extend the environment by introducing a new binder which is the closest binder now (i.e. its index will be 0) and all the other binder's indices are shifted by 1. This operation is expressed by `insert 0 (Li, T)` Γ.

So for example, the abstraction rule

$$\frac{\Gamma, x^i{:}T_1 \vdash^i t_2 \in T_2}{\Gamma \vdash^i (\lambda x{:}T_1.t_2){:}T_1 \to T_2 \ \in \ T_1 \to T_2}$$

is expressed as

```
| T_Abs : forall Li G T1 T2 t2,
    insert 0 (Li, T1) G ⊢(Li) t2 ∈ T2 ->
    G ⊢(Li) (Lam T1 t2 : T1 ==> T2) ∈ (T1 ==> T2)
```

4 Proving Soundness

We want to prove

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

and

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

First we had to reformulate the theorems into our language. Thanks to Coq's notations they don't differ much:

```
Theorem Progress : forall t T,
  ∅ ⊢(L0) t ∈ T ->
  isvalue t \/ exists t', t -->(L0) t'.

Theorem Preservation : forall t1 T G L,
  G ⊢(L) t1 ∈ T ->
  forall t2,
  t1 -->(L) t2 ->
  G ⊢(L) t2 ∈ T.
```

4.1 Induction

As we have two mutually inductive syntactic categories (the untyped terms and typed terms that wrap the untyped ones), we need an induction principle that handles them well. The default principle generated by Coq is not convenient to use as it requires separate properties for typed terms and untyped terms. However in proofs, we usually want to only prove the property on typed terms. We created a custom induction principle that only requires a property on typed terms, because the analogous property on untyped terms is inferred. The idea is further discussed in section **??**.

4.2 Progress

The progress theorem in the original proof relies mostly on an auxiliary lemma - *Level Progress*. It is defined in two parts, namely:

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.

Unsurprisingly, when proving by induction, each case needs the other. One approach to formulating this in Coq would be to define mutually recursive lemmas using the `Lemma ... with` syntax. This is however quite prone to mistakes, because the fixpoint check is only done at the end, so it is easy to create a self-referencing proof that will be rejected only after typing `Qed` at its end.

Instead we formulate this as a single lemma with a disjunction:

```
Lemma LevelProgress : forall t G T L,
  RestrictedLevel G L1 ->
  G ⊢(L) t ∈ T ->
     (L = L0 /\ (isvalue t \/ exists t', t -->(L0) t'))
  \/ (L = L1 /\ (not (isvalue (Quote t : □T)) -> exists t', t -->(L1) t')).
```

This allows us to prove both cases based on syntactic induction on t. Thanks to the induction principle defined earlier, we have all the necessary assumptions.

In the proof, in each case we `destruct L` getting two cases where `L` has a concrete value. Therefore, in the inductive hypothesis conclusion only one of the alternatives is true. We proved two helper lemmas that can be used simplify the inductive hypothesis, detecting the possible alternative and eliminating the impossible one.

The *Level Progress* proof is quite long as, beside some trivial cases, each syntactic form required a slightly different treatment. However after proving that lemma, the *Progress* theorem is a very simple corollary.

4.3 Preservation

To prove Preservation we need two auxiliary lemmas - *Weakening* and *Substitution*. These are closely related to handling variables in the environment, so they have to be quite deeply modified when translating to De Bruijn indices.

Weakening is originally defined as

Lemma (Weakening)**.** *If* $\Gamma^{[1]} \vdash^i t \in T, x \notin FV(\Gamma)$*, then* $\Gamma, x^j : T \vdash^i t \in T$*.*

The modification is rather intuitive, as explained in section 3.3, extending the environment is defined using `insert` and instead of a variable name x we have an index `x`. The fact that $x \notin FV(\Gamma)$ is expressed by showing the property for `shift x t` which shifts the indices inside of `t` so that the `x`th variable is not bound in that term, and other indices are modified in the same way as they would have been shifted by `insert`, so that they still correspond to the original binders. It is defined as:

```
Lemma Weakening: forall G L t T,
  G ⊢(L) t ∈ T ->
  forall x L' T' G',
  insert x (L', T') G = G' ->
  G' ⊢(L) (shift x t) ∈ T.
```

The *Substitution* lemma is a bit more complicated than usual:

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* $\Gamma \vdash^i t_2[x \mapsto t_1] \in T_2$*.*

The first two conditions are usual, the third one is exceptional. The third condition says that we can substitute inside arbitrary terms for variables at level 0, but when substituting a variable at level 1, the term that we are substituting into cannot contain any pattern matches. Without this assumption, we may arrive at a situation when we are substituting a variable inside a pattern matching a particular variable (ie. $(t \sim x \ ?\ t_a \parallel t_b)[x \mapsto t_c]$), we would get a result that is not even syntactically correct nor typable. So we just disallow this kinds of situations (note that the variable pattern only applies to level 1 variables).

At first it may seem like a too big restriction, making the lemma too weak. But in fact this is enough, because we are mostly doing substitutions at level 0. The only situation when we are substituting a level 1 variable is when lifting a lambda in the `lam`$[T]\ e$ pattern[5]. But in this particular case, we know that the

[5]See figure 5 in the Appendix for details.

whole lambda term is plain, so the e inside it does not contain the pattern match, as required.

Once we understand the nuance about the third condition, translating the lemma into Coq is quite simple, so we skip it.

The Preservation proof also uses a few more auxiliary lemmas to simplify some properties. For example we prove that a plain term doesn't contain pattern matches (which is almost by definition) or that renaming (ie. shifting indices in the De Bruijn formulation) doesn't introduce pattern matches if they weren't present originally.

We extensively use the tactics provided by *DbLib* for dealing with substitutions and define some more tactics specific to our formulation that help simplifying substitutions in the proof context.

5 Discussion

5.1 Multiple binders at once

Patterns that represent multiple binders and are substituted at once have to be considered very carefully as it is non-trivial to judge which indices to shift and how.

First, let's see more closely how beta-reduction works with De Bruijn indices.

To show how the indices behave, we assume there are some free variables that are not reduced (these are put on the left side of $\vdash$ indicating the environment). In our language this happens because we have variables at two levels, and while level 0 variables are reduced, level 1 are not.

For clarity, to show the intermediate step, we define the substitution operation $(e_1)[e_2/]$ that substitutes all De Bruijn indices equal to 0 in e_1 with e_2 and decreases other indices (to indicate that after substitution we have got rid of one binder). As the substitution operation is an intermediate step, the enclosing lambda is removed already, but the indices are not yet shifted, so to keep the De Bruijn indices consistent, we treat the parentheses around (e_1) as a binder. In $(e_1)[e_2]$, e_1 is still inside a binder that will replace its first free variable with e_2. With these definitions, beta-reduction will proceed as follows:

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

Now, let's analyse how we can define matching a 2-element tuple (in our case we were matching an application, but we have chosen this example as it is more familiar). Let's define `unpack` e `as` (x_1, x_2) `in` t as the operation that deconstructs a tuple $e = (e_1, e_2)$ and binds e_1 to x_1 and e_2 to x_2 in t.

As we are in the realm of De Bruijn indices, we modify that syntax to `unpack` e `as` $(\bullet, \bullet)$ `in` t where the two bullets count as two binders. Inside of t, the closest De Bruijn index (0) is bound to e_2 and the next one (1) to e_1 (we use the order that may seem reversed but that is to express that e_2 corresponds to the second bullet which is closer to t than the one that e_1 corresponds to which is consistent with the indices representing the *binder distance*). To illustrate, `unpack` (e_1, e_2) `as` $(\bullet, \bullet)$ `in` $(\#1\ \#0) \longrightarrow (e_1\ e_2)$.

It is tempting to define the reduction rule as `unpack` (e_1, e_2) `as` $(\bullet, \bullet)$ `in` $t \longrightarrow ((t)[e_2/])[e_1/]$. But let's analyse how that would work on an example with free variables:

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

We would expect to get $(\#1\ \#0)$ as the result, but instead we get $(\#1\ \#1)$. We can see that something is wrong in the second line already. The $\#0$ that will be substituted, is kept as-is, as index 0. But that variable went inside the scope of the $\bullet$ binder (represented in the substitution operation as the parentheses $(\ )$). So in that context, the index #0 actually points to the $(\ )$ binder and not T_1 as it should have (that's why we marked it red, because it actually refers to something else than what we wanted). The issue becomes completely

clear after performing the first substitution - both variables inside are now #0 so they necessarily refer to the same thing, even though initially they were not meant to.

The issue is that, when performing substitutions one-by-one, the substituted expression e_2 goes inside the scope of the other binder corresponding to e_1, so references in e_2 can be mistakenly captured to refer to e_1.

To deal with this, we need to make sure that the indices inside of e_2 are updated correctly, so that the binder corresponding to e_1 is out of its scope. That is exactly what the `shift` function is for. `shift` simply increases all indices in a term by one, allowing us to go inside a scope of a binder, but without binding to it (so that all variables still reference the same things from the outside).

The updated reduction rule will be `unpack` (e_1, e_2) `as` $(\bullet, \bullet)$ `in` $t \longrightarrow ((t)[\texttt{shift}\ e_2/])[e_1/]$ where `shift` e_2 ensures that all variables in e_2 still point to the same things, even though we moved e_2 inside a temporary.

We can revisit our example:

$T_2; T_1 \vdash$ `unpack (#1,#0) as (•,•) in (#1 #0)`
$T_2; T_1 \vdash$ `((#1 #0)[(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)`

To sum up, when doing multiple substitutions one-by-one, we need to keep in mind that terms that have been substituted first are inside the scope of all further substitutions and can capture variables from them (and get their indices mixed-up completely!). To avoid this problem we need to `shift` the terms to preserve consistency of the references. When substituting more than 2 binders at once, we may even need to `shift` some terms multiple times.

5.2 Induction principle for related types

In our language we have typed and untyped terms that are mutually inductive. We often want to prove some properties about them using syntactic induction. However the induction principles generated by default may not always be sufficient.

To illustrate the concept we will use a simplified definition:

```
Inductive Typed :=
| typed (u: Untyped) (T: Type)
with
Untyped :=
| leaf
| branch (t1 t2 : Typed).
```

The default induction principle generated by Coq for `Typed` is:

```
forall P : Typed -> Prop,
  (forall (u : Untyped) (T : Type), P (typed u T)) ->
  forall t : Typed, P t
```

When proving a theorem about typed terms, we need to show that some property holds for all untyped terms, but we don't get any inductive hypotheses about them in this induction principle. For most theorems such principle is too weak.

To improve on that, we can use Coq's `Scheme` to generate mutually inductive principles.

```
Scheme Typed_mutualind := Induction for Typed Sort Prop
  with Untyped_mutualind := Induction for Untyped Sort Prop.
```

This gives us the following principle for typed terms:

```
forall (P : Typed -> Prop) (P0 : Untyped -> Prop),
  (forall u : Untyped, P0 u -> forall T : Type, P (typed u T)) ->
```

```
  P0 leaf ->
  (forall t1 : Typed, P t1 -> forall t2 : Typed, P t2 -> P0 (branch t1 t2)) ->
  forall t : Typed, P t
```

This is much better. The typed terms are now destructed and we have to prove cases for each of the two syntactic forms. Moreover, for *branch* which contains typed terms as elements, we have the inductive hypothesis that the property holds for both of its elements.

This induction principle is already good enough, because it gives us all the necessary assumptions. In this principle, we have two properties that we prove and an assumption that relates them. The downside is that when using this induction principle, we usually have a goal with one property and the other one has to be defined every time. In the setting of typed and untyped terms, we can do better.

We can choose `P0` such that `P0 u -> forall T : Type, P (typed u T)`. We then get:

```
Lemma Typed_syntactic :
  forall (P : Typed -> Prop),
    (forall T, P (typed leaf T)) ->
    (forall (t1 t2 : Typed) T, P t1 ->  P t2 -> P (typed (branch t1 t2) T)) ->
    forall t : Typed, P t.
```

We wrap all of the untyped variants into typed variants with an arbitrary type. This allows us to reuse the property defined for typed terms. This is very convenient, as now we can just do `induction term using Typed_syntactic` and we will get goals for all the untyped cases.

5.3 Variable binding libraries

As handling variable binding is a common problem for all programming language formalizations, there are some libraries that help dealing with it. Below we describe three libraries that we approached and our experiences using them.

autosubst [5] is quite simple to use. We use special types when defining the syntactic forms: a separate type `var` for the De Bruijn index which is an alias for `nat` but it is tagged so that the library knows this is an index and handles it properly; and a type constructor `bind` that wraps any type, indicating that anything inside it consumes one binder. All properties are derived automatically. A big advantage of this library is that its simplification tactics are quite powerful, at least in our experience they worked out of the box most of the time. It is a very user-friendly and robust solution for simple calculi.

Unfortunately, *autosubst* was too simple for our use-case. In theory it supports having two types of syntax forms, but we had issues with using it. It also only allows binding one variable per syntactic form whereas we need to bind at least two (for `MatchApp`). It may be possible to work around this issue by introducing some dummy syntactic forms (but before we thought about this possible trick we already switched to the more flexible *DbLib*).

We chose to use *DbLib* [4]. It is a bit harder to start with but is much more flexible. Instead of auto-generating how the binders behave (like *autosubst* did), the user defines the syntactic forms and a `traverse` function which describes how the binders behave (the `traverse` function is described in section 3.1.2). This allows for having multiple bindings. The library also supports multiple syntactic categories, although when using that feature it may be necessary to manually call the tactics that automate derivation of basic properties. For more complex syntactic structures, these tactics failed, so we had to keep the structure quite simple. The tactics for simplifying substitution were a bit less robust than in *autosubst* and sometimes needed some help from the user, but it seems reasonable given that this library allows for more complicated syntax definitions.

Another approach worth mentioning is *Locally Nameless* [6]. It is more of an approach than a library (although the author shares a library with useful tactics). It is a hybrid approach that uses De Bruijn indices for bound variables but uses some arbitrary naming (as one would in a normal programming language) for the free variables (so the syntax needs two forms for variables - one for bound variables and another

one for free variables). Using De Bruijn indices for bound variables makes it easier to manually define the substitution operation (whereas in the other libraries it was derived automatically) which is split into two operations - there is one function for substituting free variables and separate one for substituting the first DeBruijn index (which is called *opening* the term).

It seems to be the most flexible approach but also the most complex one, as the user has to do most of the work and the amount of automation provided by the library is rather limited. It is also based on quite complicated principles (De Bruijn indices are hard to understand on their own and mixing them with normal names makes it even more difficult). It is a good choice for complex calculi for which the other libraries would be too limited.

In this project, the choice of the library was quite highly influenced by my level of understanding. I started with *autosubst* which was the simplest one and quite easy to learn, but then it seemed that it imposed some constraints that didn't allow to express important parts of the calculus (although it may have been possible to work around them with some clever but ugly tricks). Then I tried both *Locally Nameless* and *DbLib*, but at that point *Locally Nameless* was still too hard to understand it to be able to use it. *DbLib* was a nice middle-ground between flexibility and complexity and it still offered decent automation. After using it and understanding better more nuances of using De Bruijn indices, I can appreciate the beauty of *Locally Nameless* approach, but given its limited amount of automation, it is a library best used for more complex calculi.

5.4 Testing before proving

Adding the pattern matching was a very delicate matter, especially as we had to diverge slightly from the original calculus (by defining the separate syntactic categories for all pattern types). It was easy to make mistakes in the definitions of types and syntax (for example one of the bugs was having the two variables bound by app in different order in the typing judgement and in the semantics). Such mistakes lead to unprovable theorems, but they may not be easy to notice early during proving. To avoid such issues, I decided to write some *unit tests* - I defined simple programs in the calculus that I know should work and tried proving that they do type-check and evaluate correctly. These proofs were usually rather easy but they helped find lots of simple mistakes. Moreover, they can serve as examples of how to use the calculus in the formalized version which helps understanding.

It seems like a very good approach to write some simple tests (for example lemmas about the desired properties of some specific simple terms, like the ones at the end of section 3.3) before moving on to the general soundness proofs.

5.5 Iterative development

We have approached the proofs by first trying to prove a subset of the calculus corresponding to the *simply typed lambda calculus*, as a warm-up. Then we have been adding more features one by one and updating the proofs. While such approach sometimes requires rewriting substantial parts of the proofs, it allows to keep the scope more manageable. Moreover, it allows for partial results along the way, so it is a kind of a safety net. Even if some later stage were to turn out too problematic to do, the iterative approach ensures that there are partial outcomes that may stand on their own.

5.6 Proof stability, naming

Another good practice that I learnt along the way is trying to make the proofs resistant to changes. It is quite common to have to add some additional assumption in one of the definitions or simply extend the calculus with new definitions. If we don't write our proofs carefully, a small modification of the definitions may require us to rewrite most of them, while writing them well may result in not having to change anything.

Firstly, it's good to use as much automation as possible, as it is usually robust to small changes. When introducing hypotheses (with `intro` or `assert`) it is good to give them names, because the automatic naming may very easily change if an additional assumption is added in the future.

In big case-by-case proofs (for example syntactic induction), I usually try to come up with some basic strategy that solves at least the simple goals (using `induction X; try solve [general strategy].`), but in more complex proofs we may still have to manually approach many goals. It is then a good idea when focusing a new goal to write a comment shortly describing the current goal. This helps when by adding some additional rules, the order of the goal changes (result in applying tactics to wrong goals and very confusing errors). These comments help to quickly catch situations when the current goal is different than would be expected.

I didn't find a way to name hypotheses introduced by calling the `inversion` tactics (which I use extensively), so I still had to deal with unpredictable hypothesis names. It is also problematic when combining tactics. For example in the chain `induction t; inversion H1.`, `H1` may introduce different sets of hypotheses for each case. Sometimes I also want to call `inversion` but the particular hypothesis that I want to examine will be called differently in different branches of a tactic chain similar to the one just mentioned. The solution to these issues is matching the goal - the ability to write tactics that match the current goal or available hypotheses against some patterns. For example we can write:

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

which defines a tactic that finds a hypothesis containing a typing judgement for some quoted type (a $\square T$ for some T) and runs `inversion` on this hypothesis. Now we can write `induction t; invV.` and regardless of what that hypothesis would be called, we can find it and inside of `invV` bind its name to `H`. This is a very powerful technique that allows to write proofs robust to changes.

6 Summary

We have described the $\lambda^{\textcircled{\scriptsize\textonehalf}}$calculus and various challenges in formalizing it in Coq and proving its soundness. We discuss some interesting problems that had to be solved along the way. Our main contribution is the mechanized formalization of the calculus along with its soundness proofs that can be found at https://github.com/radeusgd/QuotedPatternMatchingProof/.

References

[1] Dotty documentation - macros. `https://dotty.epfl.ch/docs/reference/metaprogramming/macros.html`.

[2] Fengyun Liu and Nicolas Stucki. A theory of quoted code patterns. 2020.

[3] de Ng Dick Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. 1972.

[4] François Pottier. Coq library for working with de bruijn indices. `https://github.com/coq-community/dblib`.

[5] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015.

[6] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning - JAR*, 49:1–46, 10 2012.

Appendix

To simplify the proofs, we remove nested patterns leaving only simpler non-nested pattern matching. This requires changing some typing and semantic rules.

For clarity, we show how the syntax of our patterns corresponds to non-nested patterns in the original syntax in figure 1. As we use De Bruijn indices, the definitions in the mechanization look slightly different. So we introduce fake names for variables bound in the pattern matches that correspond to the De Bruijn indices. b_0 stands for the introduced variable name that corresponds to the most closely bound De Bruijn index and b_1 stands for the next index.

$$
\begin{aligned}
\texttt{MatchNat}\ t_1\ n\ t_2\ t_3 &\equiv t_1 \sim n\ ?\ t_2 \parallel t_3 && (1)\\
\texttt{MatchVar}\ t_1\ x\ t_2\ t_3 &\equiv t_1 \sim x\ ?\ t_2 \parallel t_3 && (2)\\
\texttt{MatchApp}\ t_1\ T_1\ T_2\ b_0\ b_1\ t_2\ t_3 &\equiv t_1 \sim (\texttt{bind}[T_1]\ b_0)\ (\texttt{bind}[T_1 \to T_2]\ b_1)\ ?\ t_2 \parallel t_3 && (3)\\
\texttt{MatchUnlift}\ t_1\ b_0\ t_2\ t_3 &\equiv t_1 \sim \texttt{unlift}\ b_0\ ?\ t_2 \parallel t_3 && (4)\\
\texttt{MatchLam}\ t_1\ (T_1 \to T_2)\ b_0\ t_2\ t_3 &\equiv t_1 \sim \texttt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3 && (5)\\
\texttt{MatchFix}\ (t_1{:}T_1)\ b_0\ t_2\ t_3 &\equiv (t_1{:}T_1) \sim \texttt{fix}\ (\texttt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3 && (6)\\
& && (7)
\end{aligned}
$$

Figure 1: Pattern syntax clarification

As we have separate syntactic forms for each pattern, we remove the general T-Pat rule and its helper pattern typing rules and we replace them with rules outlined in figure 3. The rules that were kept are listed in 2.

Similarly, we need separate E-Pat-Succ, E-Pat-Fail and E-Pat for each pattern as they are separate syntactic forms. This however makes the *match* function obsolete, as all matching rules are directly expressed by these separate rules, as listed in figure 5. The other semantic rules that were kept are listed in figure 4. The notation $\hat{t}$ is a shorthand for saying t is plain.

Explicitly-typed Terms Typing $\boxed{\Gamma \vdash^i t \in T}$

$$\Gamma \vdash^i n{:}Nat \in Nat \quad \text{(T-Nat)} \qquad \frac{\Gamma \vdash^i t \in T{\to}T}{\Gamma \vdash^i (\texttt{fix}\ t){:}T \ \in \ T} \quad \text{(T-Fix)}$$

$$\frac{\Gamma(x^i) = T}{\Gamma \vdash^i x{:}T \in T} \quad \text{(T-Var)} \qquad \frac{\Gamma \vdash^0 t \in Nat}{\Gamma \vdash^0 (\texttt{lift}\ t){:}\Box Nat \ \in \ \Box Nat} \quad \text{(T-Lift)}$$

$$\frac{\Gamma, x^i{:}T_1 \vdash^i t_2 \in T_2}{\Gamma \vdash^i (\lambda x{:}T_1.t_2){:}T_1{\to}T_2 \ \in \ T_1{\to}T_2} \quad \text{(T-Abs)} \qquad \frac{\Gamma \vdash^1 t \in T}{\Gamma \vdash^0 (\Box\ t){:}\Box T \ \in \ \Box T} \quad \text{(T-Box)}$$

$$\frac{\Gamma \vdash^i t_1 \in T_1{\to}T_2 \qquad \Gamma \vdash^i t_2 \in T_1}{\Gamma \vdash^i (t_1\ t_2){:}T_2 \ \in \ T_2} \quad \text{(T-App)} \qquad \frac{\Gamma \vdash^0 t \in \Box T}{\Gamma \vdash^1 (\$\ t){:}T \ \in \ T} \quad \text{(T-Unbox)}$$

Figure 2: Explicitly-typed Terms Typing (retained rules)

Explicitly-typed Terms Typing (added rules) $\boxed{\Gamma \vdash^i t \in T}$

$$\frac{\Gamma \vdash^0 t_1 \in \Box Nat \qquad \Gamma \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim n \ ?\ t_2 \parallel t_3) : T \ \in \ T} \quad \text{(T-Pat-Nat)}$$

$$\frac{\Gamma \vdash^0 t_1 \in \Box T_1 \qquad \Gamma(x^1) = T_1 \qquad \Gamma \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim x \ ?\ t_2 \parallel t_3) : T \ \in \ T} \quad \text{(T-Pat-Var)}$$

$$\frac{\Gamma \vdash^0 t_1 \in \Box T_2 \qquad \Gamma;\, b_1^0 : \Box(T_1 \to T_2);\, b_0^0 : \Box T_1 \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim (\mathtt{bind}[T_1]\ b_0)\ (\mathtt{bind}[T_1 \to T_2]\ b_1)\ ?\ t_2 \parallel t_3) : T \ \in \ T} \quad \text{(T-Pat-App)}$$

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

$$\frac{\Gamma \vdash^0 t_1 \in \Box T_1 \to T_2 \qquad \Gamma;\, b_0^0 : \Box T_1 \to \Box T_2 \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim \mathtt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3) : T \ \in \ T} \quad \text{(T-Pat-Lam)}$$

$$\frac{\Gamma \vdash^0 t_1 \in \Box T_1 \qquad \Gamma;\, b_0^0 : \Box(T_1 \to T_1) \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 ((t_1 : T_1) \sim \mathtt{fix}\ (\mathtt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3) : T \ \in \ T} \quad \text{(T-Pat-Fix)}$$

Figure 3: Explicitly-typed Terms Typing (added rules)

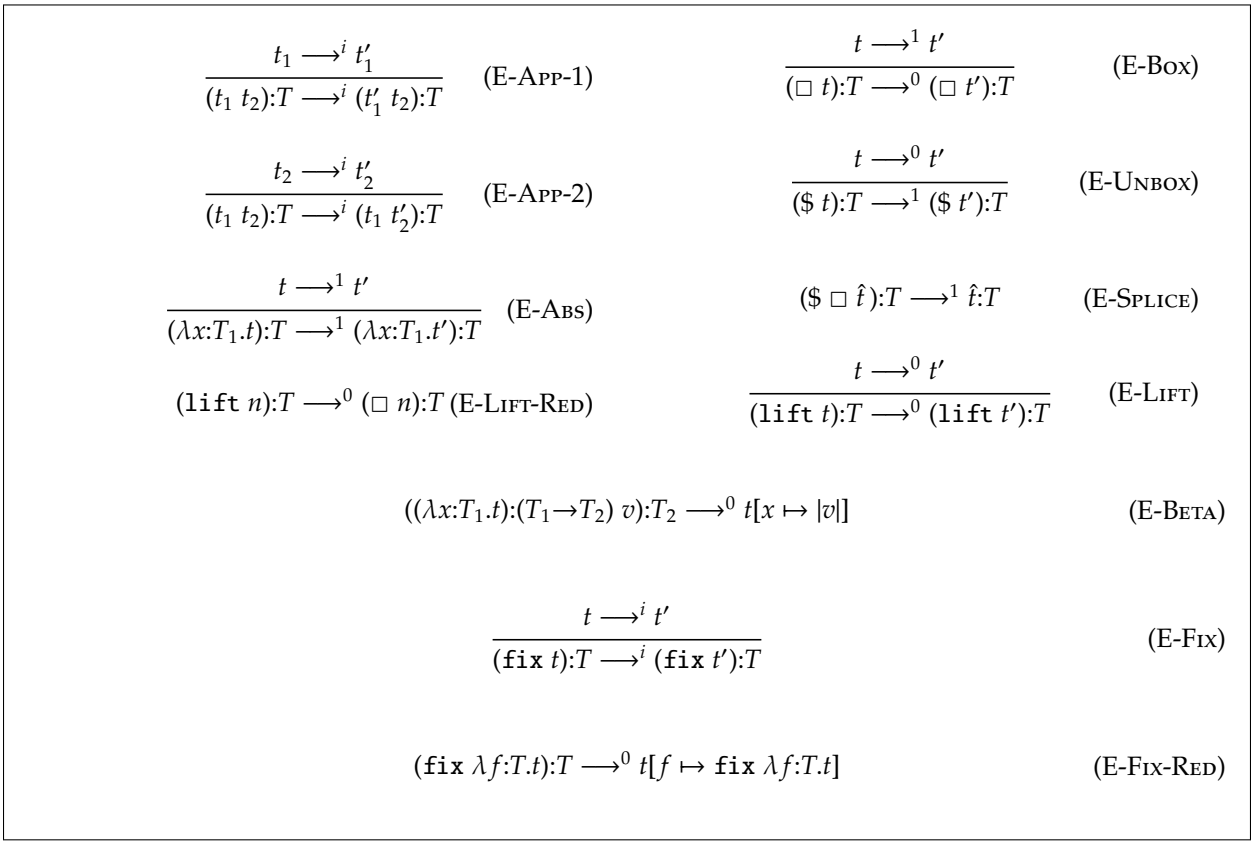

Figure 4: Small-step operational semantics (retained rules)

$$\frac{t \longrightarrow^0 t'}{(t \sim n\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t' \sim n\ ?\ t_2 \parallel t_3){:}T} \text{(E-PatNat-Red)}$$

$$((\Box\ n){:}\Box Nat \sim n\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_2 \text{(E-PatNat-Succ)}$$

$$\frac{\hat{t} \neq n}{((\Box\ \hat{t}){:}\Box Nat \sim n\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_3} \text{(E-PatNat-Fail)}$$

$$\frac{t \longrightarrow^0 t'}{(t \sim x\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t' \sim x\ ?\ t_2 \parallel t_3){:}T} \text{(E-PatVar-Red)}$$

$$((\Box\ x){:}\Box T_1 \sim x\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_2 \text{(E-PatVar-Succ)}$$

$$\frac{\hat{t} \neq x}{((\Box\ \hat{t}){:}\Box T_1 \sim x\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_3} \text{(E-PatVar-Fail)}$$

$$\frac{t \longrightarrow^0 t'}{(t \sim (\texttt{bind}[T_1 \to T_2]\ b_1)\ (\texttt{bind}[T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t' \sim (\texttt{bind}[T_1 \to T_2]\ b_1)\ (\texttt{bind}[T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T} \text{(E-PatApp-Red)}$$

$$\frac{\hat{t} = (e_1\ e_2)}{((\Box\ \hat{t}){:}\Box T_3 \sim (\texttt{bind}[T_1 \to T_2]\ b_1)\ (\texttt{bind}[T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_2[b_0 \mapsto \Box\ e_2][b_1 \mapsto \Box\ e_1]} \text{(E-PatApp-Succ)}$$

$$\frac{\hat{t} \neq (e_1\ e_2)}{((\Box\ \hat{t}){:}\Box T_3 \sim (\texttt{bind}[T_1 \to T_2]\ b_1)\ (\texttt{bind}[T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_3} \text{(E-PatApp-Fail)}$$

$$\frac{t \longrightarrow^0 t'}{(t \sim \texttt{unlift}\ b_0\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t' \sim \texttt{unlift}\ b_0\ ?\ t_2 \parallel t_3){:}T} \text{(E-PatUnlift-Red)}$$

$$((\Box\ n){:}\Box Nat \sim \texttt{unlift}\ b_0\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_2[b_0 \mapsto n] \text{(E-PatUnlift-Succ)}$$

$$\frac{\hat{t} \neq n}{((\Box\ \hat{t}){:}\Box T_1 \sim \texttt{unlift}\ b_0\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_3} \text{(E-PatUnlift-Fail)}$$

$$\frac{t \longrightarrow^0 t'}{(t \sim \texttt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t' \sim \texttt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3){:}T} \text{(E-PatLam-Red)}$$

$$\frac{\hat{t} = \lambda x{:}T_1.e \qquad x' \text{ is fresh}}{((\Box\ \hat{t}){:}\Box(T_1 \to T_2) \sim \texttt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_2[b_0 \mapsto \lambda x'{:}\Box T_1.\Box\ e[x \mapsto \$\ x']]} \text{(E-PatLam-Succ)}$$

$$\frac{\hat{t} \neq \lambda x{:}T_1.e}{((\Box\ \hat{t}){:}\Box T_1 \to T_2 \sim \texttt{lam}[T_1 \to T_2]\ b_0\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_3} \text{(E-PatLam-Fail)}$$

$$\frac{t \longrightarrow^0 t'}{(t \sim \texttt{fix}\ (\texttt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t' \sim \texttt{fix}\ (\texttt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T} \text{(E-PatFix-Red)}$$

$$\frac{\hat{t} = \texttt{fix}\ e}{((\Box\ \hat{t}{:}\Box T_1) \sim \texttt{fix}\ (\texttt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_2[b_0 \mapsto \Box\ e]} \text{(E-PatFix-Succ)}$$

$$\frac{\hat{t} \neq \texttt{fix}\ t'}{((\Box\ \hat{t}{:}\Box T_1) \sim \texttt{fix}\ (\texttt{bind}[T_1 \to T_1]\ b_0)\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 t_3} \text{(E-PatFix-Fail)}$$

Figure 5: Small-step operational semantics (added rules)