Technical report:

A Mechanized Theory of the Box Calculus

Joseph Fourment
joseph.fourment@epfl.ch

Yichen Xu
yichen.xu@epfl.ch

Abstract

Capture calculus is an extension of System $F_{<:}$ that tracks free variables of terms in their type, allowing one to represent capabilities while limiting their scope. While previous calculi had mechanized soundness proofs, the latest version, namely the box calculus, only had a paper proof. We present here our work on mechanizing the theory of the box calculus in Coq, and the challenges encountered along the way. Our mechanization is complete and available on GitHub.

1 Introduction

Capture checking is an experimental Scala feature that aims to provide a basis for new type system abilities, such as checked exceptions for higher order functions [Ode+21], algebraic effects [PP09], and safe region management through regions [TT97]. It does so by exposing, at the type level, information about free variables in terms.

To investigate the metatheory of capture checking, various calculi have been introduced, namely System $CF_{<:}$ [Bor+21] and later the box calculus (System $CC_{<:\Box}$) [Bra+22]. While System $CF_{<:}$ is fully mechanized in Coq, the soundness proofs of other variants of the calculus are not yet mechanized. In particular, System $CC_{<:\Box}$ differs from the initial System $CF_{<:}$ in that it uses monadic normal form (MNF) syntax. Switching to MNF has several prospects regarding the formalization of a larger fragment of Scala, notably path-dependent types [RL19].

Another difference is that System $CC_{<:\Box}$ restricts type abstractions and type applications to raw, uncaptured types. To recover the unrestricted type abstraction, one uses *boxing* which consists in hiding the capturing type behind a new modality denoted $\Box$. The seemingly simple restriction on type abstractions drastically simplifies the metatheory both in prosaic and mechanized proof as we will show.

Our work is composed of multiple steps illustrated in Figure 1.

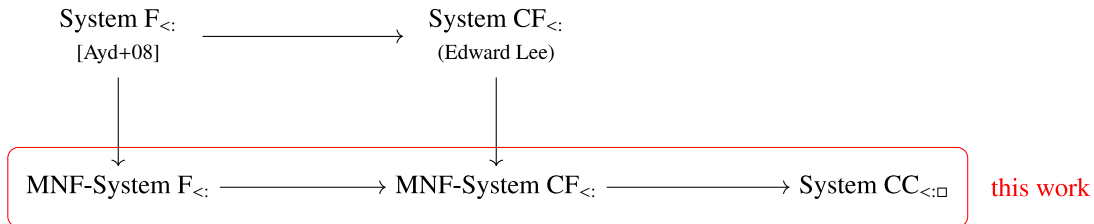

Figure 1: Roadmap for the mechanization of System $CC_{<:\Box}$

2 Pure types versus pretypes

In System $CF_{<:}$, the syntax of types is defined by two mutual syntactic categories: types and pretypes. Types correspond to pretypes prefixed by a capture set, or a type variable that will later be substituted by a type. On the other hand, pretypes can be function types, (bounded) polymorphic types or top types that can be found in System $F_{<:}$. The only difference with System $F_{<:}$ is the dependent nature of function types, since return types can refer to the name of their argument in capture sets.

However, in System $\mathrm{CC}_{<:\square}$, type variables are now considered to belong in the same syntactic categories as function types, polymorphic types and top types. This syntactic category is referred to as "pure types". The only other addition is the boxed type which masks variable dependencies represented in the capture set of a type.

$$
\begin{array}{rcl}
T, U & ::= & CR \\
 & | & X \\
Q, R & ::= & \top \\
 & | & \forall(T) \to U \\
 & | & \forall[S] \to U
\end{array}
\qquad\qquad
\begin{array}{rcl}
T, U & ::= & CR \\
 & | & R \\
Q, R & ::= & X \\
 & | & \top \\
 & | & \forall(T) \to U \\
 & | & \forall[R] \to U \\
 & | & \square T
\end{array}
$$

Figure 2: Raw syntax for type in System $\mathrm{CF}_{<:}$ (left) vs System $\mathrm{CC}_{<:\square}$ (right)

In the mechanized soundness proof of System $\mathrm{CF}_{<:}$, the raw syntax of types and pretypes is described by mutually inductive types, and we have mutually inductive judgements to ensure closedness, well-formedness, and subtyping.

However, by considering type variables as pure types, we break the assumption that type substitution is stable by the syntactic category, i.e. that if we substitute in a pure type, we will obtain a pure type, and similarly if we substitute in a type we will obtain a type. Indeed, X is pure but $\{X \to CR\}X = CR$ is not. Moreover, we now have an injection from pure types to types.

Therefore, the mutual encoding of syntax that we had in the mechanization of the soundness proof of System $\mathrm{CF}_{<:}$ does not fit as well in the System $\mathrm{CC}_{<:\square}$ calculus. This duplicates a lot of the lemmas and requires mutual induction, which can be very slow for Coq to check well-foundedness. For the reasons above, we made the decision to encode the raw syntax of pure types and types in a single inductive type and just have closedness be expressed in a mutually inductive fashion. This means that we have two judgements T **type** and R **pure**, as well as an injection R **pure** $\Rightarrow$ R **type**. This change requires extra attention to not mix up captured types and pure types, but results in a drastically shorter proof, which is also faster to check. For example, pure types need to be augmented by an empty capture set to be considered as captured types.

3 Well-formedness

In System $\mathrm{CF}_{<:}$, the well formedness judgement needed to take care of variance which was somewhat complicated to deal with in the mechanized proof.

$$\frac{C \subseteq A_+ \qquad \forall x_i \in C\ x_i : S_i \in \Gamma \qquad \Gamma; A_+; A_- \vdash_T U\ \mathbf{wf}}{\Gamma; A_+; A_- \vdash_T CU\ \mathbf{wf}}\ (\text{CAPT-WF})$$

$$\frac{\Gamma; A_+; A_- \vdash_T U\ \mathbf{wf}}{\Gamma; A_+; A_- \vdash_T \{\star\}U\ \mathbf{wf}}\ (\text{UNIVERSE-WF}) \qquad \frac{X <: T \in \Gamma}{\Gamma; A_+; A_- \vdash_T X\ \mathbf{wf}}\ (\text{TVAR-WF})$$

$$\frac{\Gamma; A_-; A_+ \vdash_T S\ \mathbf{wf} \qquad (\forall x \notin L\ \ \Gamma; A_+ \cup \{x\}; A_- \vdash_T [0 \to x]T\ \mathbf{wf})}{\Gamma; A_+; A_- \vdash_R \forall(S) \to T\ \mathbf{wf}}\ (\text{FUN-WF})$$

$$\frac{\Gamma; A_-; A_+ \vdash_R S\ \mathbf{wf} \qquad (\forall X \notin L\ \ \Gamma; A_+ \cup \{X\}; A_- \vdash_T \{0 \to X\}T\ \mathbf{wf})}{\Gamma; A_+; A_- \vdash_R \forall[S] \to T\ \mathbf{wf}}\ (\text{TFUN-WF})$$

$$\Gamma; A_+; A_- \vdash_R \top\ \mathbf{wf}\ (\text{TOP-WF})$$

Figure 3: Locally-nameless presentation of the well-formedness rules for types ($\vdash_T$) and pretypes ($\vdash_R$) in System $\mathrm{CF}_{<:}$

Then, the usual well-formedness judgements for types and pretypes are defined as follows:

$$\Gamma \vdash T \textbf{ wf} := \Gamma; \textbf{dom}(\Gamma); \textbf{dom}(\Gamma) \vdash_T T \textbf{ wf}$$
$$\Gamma \vdash R \textbf{ wf} := \Gamma; \textbf{dom}(\Gamma); \textbf{dom}(\Gamma) \vdash_R R \textbf{ wf}$$

With the addition of boxes, the well-formedness judgement can be expressed in System $CC_{<:\Box}$ in a way that is closer to the well-formedness judgement of System $F_{<:}$. Indeed, in System $CF_{<:}$, type variables can occur in capture sets. This way one can instantiate a type abstraction with a captured type T, and all occurrences of the bound type X will be replaced with $\textbf{cv}(T)$. This feature is the reason why we need to parametrize the well-formedness judgement by atom sets. However, System $CC_{<:\Box}$ restricts instantiation to pure types, which means capture sets no longer need to account for type variables. Instead, one can instantiate a type abstraction with a boxed type and later use unboxing to recover the underlying captured type. Moreover, the single inductive type describing the raw syntax enables us to describe well-formedness in a single inductive type. The final well-formedness judgement is defined as follows:

$$\frac{C \subseteq \textbf{dom}(\Gamma) \cup \{\star\} \quad \Gamma \vdash R \textbf{ wf}}{\Gamma \vdash CR \textbf{ wf}} \text{(CAPT-WF)} \qquad \frac{X <: T \in \Gamma}{\Gamma \vdash X \textbf{ wf}} \text{(TVAR-WF)}$$

$$\frac{\Gamma \vdash S \textbf{ wf} \quad (\forall x \notin L \;\; \Gamma \vdash [0 \to x]T \textbf{ wf})}{\Gamma \vdash \forall(S) \to T \textbf{ wf}} \text{(FUN-WF)} \qquad \frac{\Gamma \vdash S \textbf{ wf} \quad (\forall X \notin L \;\; \Gamma \vdash \{0 \to X\}T \textbf{ wf})}{\Gamma \vdash \forall[S] \to T \textbf{ wf}} \text{(TFUN-WF)}$$

$$\Gamma \vdash \top \textbf{ wf} \text{ (TOP-WF)} \qquad \frac{\Gamma \vdash CR \textbf{ wf}}{\Gamma \vdash \Box CR \textbf{ wf}} \text{(BOX-WF)}$$

Figure 4: Locally-nameless presentation of the well-formedness rules for types in System $CC_{<:\Box}$

These design decision further cut down the length proof by a large amount, and simplified the overall mechanization process.

4 Subtyping

Another consequence of encoding types in a non-mutually inductive manner is that subtyping for types and pure types can be merged into a single non-mutual inductive judgement, just like well-formedness. The former subtyping judgements for types and pretypes in System $CF_{<:}$ was defined as in figure 5.

$$\frac{\Gamma \textbf{ wf} \quad \Gamma \vdash X \textbf{ wf}}{\Gamma \vdash_T X <: X} \text{(SUB-REFL-TVAR)} \qquad \frac{X <: S \in \Gamma \quad \Gamma \vdash_T S <: T}{\Gamma \vdash_T X <: T} \text{(SUB-TRANS-TVAR)}$$

$$\frac{\Gamma \vdash C_1 <:_C C_2 \quad \Gamma \vdash_R R_1 <: R_2}{\Gamma \vdash_T C_1 R_1 <: C_2 R_2} \text{(SUB-CAPT)} \qquad \frac{\Gamma \textbf{ wf} \quad \Gamma \vdash T \textbf{ wf}}{\Gamma \vdash_R T <: \top} \text{(SUB-TOP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_T S_2 <: S_1 \quad \Gamma \vdash S_1 \textbf{ wf} \quad \Gamma \vdash S_2 \textbf{ wf} \\ (\forall x \notin L \;\; \Gamma, x : S_1; \textbf{dom}(E) \cup \{x\}; \textbf{dom}(E) \vdash_T \{0 \to \{x\}\}T_1 \textbf{ wf}) \\ (\forall x \notin L \;\; \Gamma, x : S_2; \textbf{dom}(E) \cup \{x\}; \textbf{dom}(E) \vdash_T \{0 \to \{x\}\}T_2 \textbf{ wf}) \\ (\forall x \notin L \;\; \Gamma, x : S_2 \vdash_T \{0 \to \{x\}\}T_1 <: \{0 \to \{x\}\}T_2)\end{array}}{\Gamma \vdash_R \forall(S_1) \to T_1 <: \forall(S_2) \to T_2} \text{(SUB-FUN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_T S_2 <: S_1 \quad \Gamma \vdash S_1 \textbf{ wf} \quad \Gamma \vdash S_2 \textbf{ wf} \\ (\forall X \notin L \;\; \Gamma, X <: S_1; \textbf{dom}(E) \cup \{X\}; \textbf{dom}(E) \cup \{X\} \vdash_T \{0 \to \{X\}\}T_1 \textbf{ wf}) \\ (\forall X \notin L \;\; \Gamma, X <: S_2; \textbf{dom}(E) \cup \{X\}; \textbf{dom}(E) \cup \{X\} \vdash_T \{0 \to \{X\}\}T_2 \textbf{ wf}) \\ (\forall X \notin L \;\; \Gamma, X <: S_2 \vdash_T \{0 \to \{X\}\}T_1 <: \{0 \to \{X\}\}T_2)\end{array}}{\Gamma \vdash_R \forall[S_1] \to T_1 <: \forall[S_2] \to T_2} \text{(SUB-TFUN)}$$

Figure 5: Locally-nameless presentation of the subtyping rules for types ($\vdash_T$) and pretypes ($\vdash_R$) in System $CF_{<:}$

In the mechanized proof for System $CC_{<:\square}$, the new single subtyping judgement is:

$$\frac{\Gamma \textbf{ wf} \qquad \Gamma \vdash X \textbf{ wf}}{\Gamma \vdash X <: X} \text{ (SUB-REFL-TVAR)} \qquad \frac{X <: S \in \Gamma \qquad \Gamma \vdash S <: T}{\Gamma \vdash X <: T} \text{ (SUB-TRANS-TVAR)}$$

$$\frac{\Gamma \vdash C_1 <:_C C_2 \qquad \Gamma \vdash R_1 <: R_2 \qquad R_1 \textbf{ pure} \qquad R_2 \textbf{ pure}}{\Gamma \vdash C_1 R_1 <: C_2 R_2} \text{ (SUB-CAPT)}$$

$$\frac{\Gamma \textbf{ wf} \qquad \Gamma \vdash T \textbf{ wf} \qquad T \textbf{ pure}}{\Gamma \vdash T <: \top} \text{ (SUB-TOP)} \qquad \frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \square T_1 <: \square T_2} \text{ (SUB-BOX)}$$

$$\frac{\Gamma \vdash C_2 R_2 <: C_1 S_1 \qquad (\forall x \notin L \;\; \Gamma, x : C_2 S_2 \vdash \{0 \to \{x\}\} T_1 <: \{0 \to \{x\}\} T_2)}{\Gamma \vdash \forall (S_1) \to T_1 <: \forall (S_2) \to T_2} \text{ (SUB-FUN)}$$

$$\frac{\Gamma \vdash R_2 <: R_1 \qquad (\forall X \notin L \;\; \Gamma, X <: R_2 \vdash \{0 \to \{x\}\} T_1 <: \{0 \to \{x\}\} T_2)}{\Gamma \vdash \forall [R_1] \to T_1 <: \forall [R_2] \to T_2} \text{ (SUB-TFUN)}$$

Figure 6: Locally-nameless presentation of the subtyping rules in System $CC_{<:\square}$

However, we encountered an issue when trying to prove the transitivity of subtyping. Transitivity of subtyping is one of the only "high-level" lemmas that we prove using mutual induction, since it depends on whether the middle type is pure or not. Given the structure of types and pretypes, Coq cannot check that a naive mutually inductive proof terminates, because of the injection from pure types to types. Using a custom combined induction principle lets us prove the transitivity of subtyping in a way that Coq can check for its termination.

5 Reduction rules

The small-step semantics of System $F_{<:}$ in Aydemir's proof [Ayd+08] are specified using a standard binary relation on terms. Using monadic normal form, we can specify our semantics using an abstract machine. During our adaptation of System $F_{<:}$ to MNF-System $F_{<:}$, we took care of defining the small-step semantics in a similar way to that of the System $CC_{<:\square}$ semantics illustrated in figure 7.

$$\begin{array}{rrcl} \text{Store context: } S & ::= & [\,] & | \;\; \textbf{let } x = v \textbf{ in } S \\ \text{Evaluation context: } E & ::= & [\,] & | \;\; \textbf{let } x = E \textbf{ in } e \end{array}$$

$$\begin{array}{llll} S[E[xy]] & \longrightarrow S[E[[z \to y]e]] & \text{if } S(x) = \lambda(z : T).\, e & \text{(APP)} \\ S[E[x[T]]] & \longrightarrow S[E[\{X \to T\}e]] & \text{if } S(x) = \Lambda[X <: T].\, e & \text{(TAPP)} \\ S[E[C \multimap x]] & \longrightarrow S[E[y]] & \text{if } S(x) = \square y & \text{(OPEN)} \\ S[E[\textbf{let } x = y \textbf{ in } e]] & \longrightarrow S[E[[x \to y]e]] & & \text{(RENAME)} \\ S[E[\textbf{let } x = y \textbf{ in } v]] & \longrightarrow S[\textbf{let } x = v \textbf{ in } E[e]] & \text{if } E \neq [\,] & \text{(LIFT)} \end{array}$$

Figure 7: Reduction rules for System $CC_{<:\square}$

Working with holed contexts as in figure 7 is however cumbersome in mechanized proofs. We therefore represent our expressions by an abstract machine state triplet $\langle S \mid E \mid e \rangle$ where S is a list of value bindings representing the store context, and E is a list of continuations representing the evaluation context. Our small-step semantics for System $F_{<:}$ is given in figure 8. Note that the only additional rule is the (LET) rule which simply builds up the evaluation context if the current focused expression is a let-binding.

$$\frac{x = \lambda(T)e \in S \qquad y = v \in S}{\langle S \mid E \mid x\,y\rangle \longrightarrow \langle S \mid E \mid [0 \to y]e\rangle}\ (\textsc{app}) \qquad \frac{x = \Lambda[R]e \in S \qquad R\ \mathbf{pure}}{\langle S \mid E \mid x[R]\rangle \longrightarrow \langle S \mid E \mid \{0 \to R\}e\rangle}\ (\textsc{tapp})$$

$$\frac{x = \Box y \in S}{\langle S \mid E \mid C \multimapinv x\rangle \longrightarrow \langle S \mid E \mid y\rangle}\ (\textsc{open}) \qquad \frac{x = v \in S}{\langle S \mid e :: E \mid x\rangle \longrightarrow \langle S \mid E \mid [0 \to x]e\rangle}\ (\textsc{rename})$$

$$\langle S \mid e :: E \mid v\rangle \longrightarrow \langle S, x = v \mid E \mid [0 \to x]e\rangle\ (\textsc{lift})$$

$$\langle S \mid E \mid \mathbf{let}\ e_1\ \mathbf{in}\ e_2\rangle \longrightarrow \langle S \mid e_2 :: E \mid e_1\rangle\ (\textsc{let})$$

Figure 8: Reduction rules for the System $CC_{<:\Box}$ abstract machine

6 Discussion

6.1 Speeding up the proof checking

Judging the quality of a mechanized proof does not reduce to whether the proof assistant accepts it. Specifically, we want our proofs to be fast to check and robust to changes in definitions. However, these aspects are somewhat in tension: robust proofs tend to use more automation which leads them to be shorter but also longer to verify. Instead of writing all proof steps, we rely on tactics to do the tedious, overly formal work for us. One such tactic is the `pick fresh` tactic [Ayd+08] which gathers all variables in context and generates a fresh variable together with a proof that it is actually fresh (i.e. that it is not in the gathered set of atoms). Then, we prove freshness goals of the form $x \notin \ldots$ using the `fsetdec` tactic from the Coq standard library. The `fsetdec` tactic can sometimes take a few seconds to complete when there are lots of atoms in scope. A possible improvement over the current status is to differentiate atoms by what they stand for, i.e. atoms representing term variables should not be confused with atoms standing for type variables. One could parametrize the `atom` type by a syntactic category (such as `exp` or `typ`) and specifying in the `pick fresh` tactic when we want a fresh term variable or a fresh type variable. By doing so, the gathered sets of atoms should be smaller and we can hope that the calls to `fsetdec` will be faster.

6.2 Using automation libraries

The current winner of the POPLMark challenge, in which programming language researchers compete to mechanize the soundness of System $F_{<:}$ in the least amount of code, is AutoSubst [STS15]. AutoSubst is a Coq library tailored for automating the proof of substitution lemmas, which account for a large portion of the overall proof.

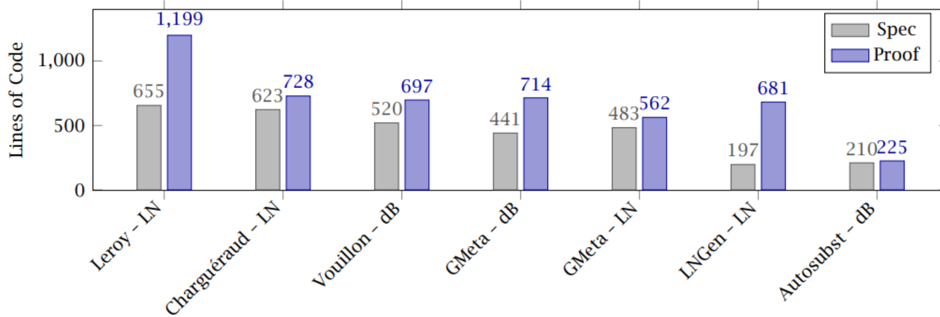

Figure 9: Comparison of various submissions to the POPLMark challenge (source: [STS15]

Our current mechanization is based on Arthur Charguéraud's locally nameless proof [Cha12], and only uses small tactic libraries, namely LibTactics and TaktikZ. One could wonder if AutoSubst could be applied to System $CC_{<:\Box}$ and how small would the soundness proof be if we were to use it. One major difference is that AutoSubst uses De Bruijn indices instead of locally nameless.

6.3 Finding unused definitions

The GitHub repository containing the soundness proof is an accumulation of various branches on top of each other, each proving soundness for a different variant of capture calculus. Dealing with such a project means that unused lemmas and definitions are accumulated over the years. Indeed, the usual workflow is to change the definitions, adapt every proof in order. However, we cannot know in advance which lemmas are going to be useful in the soundness proof. Hence, we decided to write a small script to detect unused lemmas to further cut down the length of the proof.

The script works by calling `coq-dpdgraph`, which generated a text dump of the dependency graph. Then, after specifying the roots (the main results of the proof, i.e. progress and preservation), the script parses the textual representation of the graph into an in-memory graph and computes all reachable reverse dependencies. The unreached nodes are then sorted in topological order and by order of the modules to allow removing unused lemmas progressively.

One caveat is that `coq-dpdgraph` does not treat mutually inductive lemmas as interdependent. Therefore we needed to eliminate these false positives by adding unused lemmas within a "with" block as roots.

The script detected ~100 unused lemmas (on a total of ~600), which in total resulted in a decrement of ~1.5k LOC (on a total of ~9k LOC).

7 Conclusion

The mechanization of the soundness proof of System $CC_{<:\square}$ has shown that the box calculus is a cleaner and simpler formulation of capture checking. However, the restrictions needed to simplify the metatheory, namely forcing the user to box types before instantiating a polymorphic type, mean that we need a mechanism to infer those boxes to make the calculus accessible and seamlessly integrate with existing code. Nevertheless, the mechanization of System $CC_{<:\square}$ is almost half as long as the one of System $CF_{<:}$, going from ~12k LOC to ~7k LOC. Our proof is available on GitHub at https://github.com/felko/ccsubbox.

8 Future work

8.1 Extending the box calculus to account for a larger fragment of Scala

In the POPLMark tutorial, we proved soundness for System $F_{<:}$ extended with sum types, and in the MNF proof, we instead extended System $F_{<:}$ with product types. However, neither sum nor product types are featured in System $CF_{<:}$ or System $CC_{<:\square}$. An interesting extension to the project would be to bring those constructs to System $CC_{<:\square}$. Another extension could be path-dependent types and DOT in general.

8.2 Box inference

To integrate seamlessly with existing Scala code, we need a mechanism to infer boxes. This means we have to elaborate a more permissive calculus that allows type applications for capturing types, without requiring boxing, call it System $CC_{<:}$, into System $CC_{<:\square}$. A possible extension of this project would be to write a formally proven elaborator, showing a bisimulation between the reduction relation in System $CC_{<:}$ and the reduction relation between box-inferred terms in System $CC_{<:\square}$.

References

[TT97] Mads Tofte and Jean-Pierre Talpin. "Region-based memory management". In: *Information and computation* 132.2 (1997), pp. 109–176.

[Ayd+08] Brian Aydemir et al. "Engineering formal metatheory". In: *Acm sigplan notices* 43.1 (2008), pp. 3–15.

[PP09] Gordon Plotkin and Matija Pretnar. "Handlers of algebraic effects". In: *European Symposium on Programming*. Springer. 2009, pp. 80–94.

[Cha12] Arthur Charguéraud. "The locally nameless representation". In: *Journal of automated reasoning* 49.3 (2012), pp. 363–408.

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. “Autosubst: Reasoning with de Bruijn terms and parallel substitutions”. In: *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6*. Springer. 2015, pp. 359–374.

[RL19] Marianna Rapoport and Ondřej Lhoták. “A path to DOT: formalizing fully path-dependent types”. In: *arXiv preprint arXiv:1904.07298* (2019).

[Bor+21] Aleksander Boruch-Gruszecki et al. “Tracking Captured Variables in Types”. In: *arXiv preprint arXiv:2105.11896* (2021).

[Ode+21] Martin Odersky et al. “Safer exceptions for Scala”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. 2021, pp. 1–11.

[Bra+22] Jonathan Immanuel Brachthäuser et al. “Scoped Capabilities for Polymorphic Effects”. In: (2022).