A Mechanized Theory of the Box Calculus

Joseph Fourment joseph.fourment@epfl.ch EPFL Lausanne, Switzerland

Abstract

The 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 — notably System $CF_{<:}$ — the latest version, namely the box calculus (System $CC_{<:}$), 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. While doing so, we motivate the current design of capture calculus, in particular the concept of *boxes*, from both user and metatheoretical standpoints. Our mechanization is complete and available on GitHub.

CCS Concepts: • Theory of computation \rightarrow Type structures.

Keywords: mechanized metatheory, capture checking, capture calculus, box calculus, effects, Scala

ACM Reference Format:

1 Introduction

Capture checking is an experimental Scala feature that aims to provide a basis for new type system abilities, such as checked exceptions — particularly in presence of higher-order functions — [Odersky, Boruch-Gruszecki, Brachthäuser, et al. 2021], algebraic effects [Plotkin and Pretnar 2009], and safe memory management through regions [Tofte and Talpin 1997]. It does so by exposing type level information about free variables in terms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IWACO '23, Sun 22 - Fri 27 October, 2023, Cascais, Portugal © 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00
https://doi.org/XXXXXXXXXXXXXXX

Yichen Xu yichen.xu@epfl.ch EPFL Lausanne, Switzerland

To investigate the metatheory of capture checking, two calculi have been introduced, namely System $CF_{<:}$ [Boruch-Gruszecki et al. 2021] and later the box calculus (System $CC_{<:\square}$) [Odersky, Boruch-Gruszecki, Lee, et al. 2022]. While System $CF_{<:}$ is fully mechanized in Coq, the soundness proofs of other variants of the calculus are not yet mechanized. In particular, System $CC_{<:\square}$ differs from the initial System $CF_{<:}$ in that it uses monadic normal form (MNF) syntax [Hatcliff and Danvy 1994]. Switching to MNF has several prospects regarding the formalization of a larger fragment of Scala, notably path-dependent types [Rapoport and Lhoták 2019].

Another difference is that System $CC_{<:\square}$ restricts type abstractions and type applications to *pure* (uncaptured) types. To recover the unrestricted type abstraction, one uses *boxing* which consists in hiding the capturing type behind a new modality denoted \square . 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. This paper mostly focuses on the mechanization of System $CC_{<:\square}$, the mechanizations of MNF-System $F_{<:}$ and MNF-System $CC_{<:\square}$ are also available on GitHub.

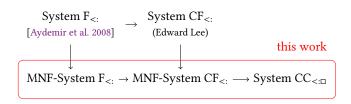


Figure 1. Roadmap for the mechanization of System CC_{<:□}

2 Motivation

Here we explain the rationale behind the capture calculus and what problem it is trying to solve, illustrating its purpose with concrete use cases expressed in Scala. We take inspiration in the examples detailed in [Odersky, Boruch-Gruszecki, Lee, et al. 2022].

2.1 Capabilities & Monadic Reflection

Monadic effects are currently the standard tool to deal with effects in functional programming. However, monads notoriously don't compose in general, we need extra machinery to compose them when it is legal to do so, such as *monad*

transformers. A more recent approach is the theory of algebraic effects with handlers [Plotkin and Pretnar 2009], which restricts the expressible effects to a subset of so-called algebraic effects, that compose out of the box. This approach has been implemented as libraries, e.g. in Haskell [Kiselyov and Ishii 2015], using an integrated ambient monad parametrized with the kind of effects that a computation is able to perform.

One drawback with any kind of monadic effect system is that effectful computations are expressed in continuation-passing style. While syntactic sugar such as Haskell's donotation or Scala's <code>for-yield</code> syntax can make such code look like direct style, arbitrary control flow structures such as <code>while</code> loops are not permitted. Moreover, monadic code often introduces some overhead due to the allocation of closures to represent the continuations, and due to the additional calls, both to the binding operation (<code>>>=</code> or flatMap) and the continuations.

Ideally, we would like to write effectful functions in directstyle. To this end, algebraic effects have also been implemented in standalone compilers, with their semantics based on delimited continuations. However, another approach is to use monadic reflection [Filinski 1999, 1994], which allows us to use monads in direct-style. Monadic reflection can be implemented on top of Scala [Brachthäuser et al. 2021] provided an implementation of delimited continuations. In particular, delimited continuations can be implemented on top of coroutines, of which Project Loom, a fork of OpenJDK that aims to implement lightweight threading, has built-in support. Using monadic reflection, effectful computations can be encoded in Scala using context functions, taking the context argument as the capability. A capability does not have any existence at runtime, it is merely a permission to perform certain functions. For example, a function f that can fail with an error of type E and can return a T will be typed f: CanThrow[E] ?=> T and can be declared as def f(using err: CanThrow[E]): T. Context functions will resolve the capability with the **given** mechanism.

2.2 Example

Error handling. The **try** block can be used to generate a **CanThrow** capability.

```
def sqrt(x: Double)
          (using err: CanThrow[NegSqrt]): Double =
    if x < 0 then err.throw(NegSqrt(x)) else x ^ 0.5

def sqrtThunk(x: Double): () => Option[Double] =
    try () => Some(sqrt(x))
    catch case exc: NegSqrt => (() => None)
```

At the beginning of the **try** block, the compiler will insert the value **val** err: **CanThrow**[NegSqrt] = ???. Note that its definition is unimportant, as capabilities are erased at runtime

However, there is a pitfall in this approach. Consider now the value val f = sqrtThunk(-1). The function f has type

() => Option[Double] but f() will throw an uncaught error, despite the surrounding try block, because the handler is no longer in scope when the thunk is called.

Resource management. Consider a function with File defined below:

```
def withFile[T](p: String, f: OutputStream => T): T =
  val file = new FileOutputStream(p)
  val result = f(file)
  file.close()
  result
```

The function withFile allows one to temporarily open a file, write in it, and then finally close the file. This is analogous with the **try**-with-resources idiom. However, consider a function that creates a logger, returning a closure that contains the temporary stream:

```
def makeLogger(p: String): String => () =
   withFile(p, stream => str => stream.write(str))
```

Calling the resulting closure of makeLogger is ill-behaved, because it is executed after the stream is closed.

2.3 The Capture Calculus

The purpose of the capture calculus is to prevent cases such as makeLogger by statically preventing scoped capabilities to leak in closures. It does so by enabling the types to represent which free variables are captured in terms. As the capture calculus is meant to be integrated in Scala, a functional language with subtyping, it is based on System $F_{<:}$. The main addition of the capture calculus compared to System $F_{<:}$ is the notion of a *capture set*. A capture set is a finite set of variables attached to a type, such a type is called a *capturing type*, and is denoted CR where C is the capture set and R is the underlying type which must be uncaptured (we say that R is a *pure* type). For example, the type $\{a, b, c\}$ **Int** is the type of terms of pure type **Int** that can mention a, b and c as free variables and no more — but potentially less due to subtyping, or more precisely *subcapturing*.

The subcapturing relation defines a partial order on capture set. The relation is analogous to the powerset poset ordered by inclusion, with the important difference that variables present in a capture set can be expanded into their own capture set with a combination of the (sc-set) and (sc-var) rules written below:

$$\frac{\Gamma \vdash \{x_1\} <:_C C \qquad \dots \qquad \Gamma \vdash \{x_n\} <:_C C}{\Gamma \vdash \{x_1, \dots, x_n\} <:_C C} \text{ (sc-set)}$$

$$\frac{x : C_1 R \in \Gamma \qquad \Gamma \vdash C_1 <:_C C_2}{\Gamma \vdash \{x\} <:_C C_2} \text{ (sc-var)}$$

Note that, as variables can occur in types, the capture calculus is dependent, in a weak sense. More precisely, we want to allow the return type of a function to reference the name of its parameter, so instead of having the usual function

type as $T \to U$, function types are denoted $\forall (x : T) \to U$, where x can be referenced in a capture set of U.

Going back to our sqrt example, we can no longer type sqrtThunk. Informally, this is because the resulting closure now has type

{err} (() => Option[Double]) which cannot be returned
as it captures a local capability. Note that the closure does
not need to capture x because of the (sc-var) rule of the subcapturing judgement, as the capture set of x itself is empty.

```
def sqrtThunk(x: Double) =
   try
   val err: CanThrow[NegSqrt] = ???
   () => Some(sqrt(x)(using err))
   // has type {err} (() => Option[Double])
   catch case exc: NegSqrt => () => None
```

2.4 Effect Polymorphism

A typical challenge of effect systems is the notion of *effect polymorphism*. In presence of higher-order functions, we often want to allow the parameter function to be effectful. Most effect systems have explicit effect polymorphism, consider for example this signature for map in Koka [Leijen 2014].

```
fun map(xs: list<a>, f: a -> e b): e list<b>
```

In the capture calculus, we instead use subcapturing to allow higher-order functions to perform some effects. In particular, we have a universal capture set, denoted {*}, to which all capture sets are subcapturing. As a syntactic sugar, we denote A => B impure functions from A to B, i.e. {*} (A -> B), and A -> B for pure functions — those that are not capturing any capability. The signature of the map method can be expressed with this syntax as def map[B](f: A => B): List[B], which is exactly the standard signature already present in the Scala standard library. Hence, the capture calculus can represent effect polymorphism with very low or even non-existent syntactic overhead, and can therefore be retrofitted into existing code without requiring an entire rewrite.

The universal capability {*} can also be used to prevent variables from being leaked from continuations. Indeed, a particularity of {*} is that generic types cannot be instantiated with a type whose capture set is universal. Conceptually, all variable of a capture set of a type argument must be present in the current environment at the instantiation site, which is never the case for the universal capture set, otherwise we could mint capabilities out of thin air, defeating the point.

To illustrate how the universal capture set can be used to prevent capabilities from leaking, recall our withFile example. Let us write its signature in the following way:

Now, our faulty makeLogger function is prevented at compiletime. It instantiates T := {stream} (String => Unit), but the capability stream is not in scope at the call site of withFile, so we need to widen it to the next bigger capture set. We have no other choice than to use {*} which is forbidden by the restriction discussed above.

2.5 Capture Tunneling

```
Consider a simple Pair type.
class Pair[+T, +U](x: T, y: U) {
    def fst: T = x
    def snd: U = y
}
```

Imagine that we want to bundle two impure values, one that can throw errors, sqrt: {err} (Double -> Double) and one that has access to the filesystem to print strings, log: {fs} (String -> Unit). What should be the capture set of val p = Pair(sqrt, log)? One option is to float the capture sets outwards, and take their union, i.e. {err,fs}. However, such a type wouldn't be precise enough, p.fst would be capturing the fs capability despite the fact that it does not access the filesystem. Consider a method mapFirst that maps the first element of the pair.

If capture sets were propagated, one would have to annotate the p parameter with the universal capture set, and consequently the return type would also have to capture {*}, making the result type intolerably imprecise.

For this reason, the capture set of generic type arguments is not propagated beyond the instantiation site, and in our example, the capture set of $val\ p = Pair(sqrt, log)$ would thus be empty. Then, at the use site of the field, we reveal its sealed capture sets. For instance, we would like to type the closure () => p.fst(0.0) as {err} () -> Double. This behavior is dubbed *capture tunneling*, it allows us to describe capture sets tightly, and avoid operations on generic types to gradually lose in precision.

2.6 Boxes

Note that, if type parameters were constrained to be pure (uncaptured), then the problem described in the previous section would not arise, since there would be no capture sets. Obviously, requiring all type arguments to be pure would be overly restrictive. For this purpose, System $CC_{<:\square}$ introduces the concept of *boxes*. Box (\square) is a modality that hides the captured variables of its argument, allowing us to treat a captured type as pure. The purpose of boxing is to enforce type arguments to be pure without losing in expressivity compared to System $F_{<:.}$ More precisely, one can instantiate a generic type with an impure type CR, by putting it into a box, denoted $\square CR$. At any point, we can box a term, as long as its captured variables are currently in scope. Then, we have to explicitly tell when we want to reveal (or *unbox*) the captured variables of a boxed term, when we want to

use it, again provided that the context in which we unbox contains underlying capture set. It is the mechanism that enables capture tunneling, and thus more precise capturing, particularly in the presence of generics.

Going back to our **Pair** example, p would have to be declared as following:

```
val p: Pair[□ {err} (Double -> Double),
      □ {fs} (String -> Unit)] =
Pair(□ sqrt, □ log)
```

Then, one can recover the underling type behind the box using the unboxing syntax $C \hookrightarrow x$.

```
val sqrt: {err} (Double → Double) =
  {err} ← p.fst
```

This is only legal if the err capability is in scope at the unboxing site, e.g. if the unbox happens inside a **try**-block.

While the addition of boxes is a significant departure from System $F_{<:}$, it turns out that boxing and unboxing can be inferred in many cases in practice, as shown in [Xu and Odersky 2023], keeping the syntactic overhead low.

3 Metatheory

We now focus on the metatheory of the capture calculus and how we approached its mechanization. In particular, we compare the mechanizations of System $CF_{<:}$ with the one of System $CC_{<:}$ in the hope of motivating the concept of boxes, but this time from a metatheoretical perspective. We explain how the restriction on polymorphic types bounds to be pure leads to a much simpler theory.

3.1 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 $CC_{<:\square}$, 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.

Figure 2. Raw syntax for type in System $CF_{<:}$ (left) vs System $CC_{<:}$ (right)

In the mechanized soundness proof of System 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 was used in the mechanization of the soundness proof of System $CF_{<:}$ does not fit as well in System $CC_{<:}$. This duplicates a lot of the lemmas and requires mutual induction, for which Coq can be very slow to check for termination. 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 — see Figure 3 — 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.

3.2 Well-formedness

In System CF $_{<::}$, the well formedness judgement needs to take care of variance which was somewhat complicated to deal with in the mechanized proof. Due to the type/pretype distinction, it is split into two mutually inductive types, which are defined by four-place relations $\Gamma; A_+; A_- \vdash_T T$ wf and $\Gamma; A_+; A_- \vdash_R R$ wf for types and pretypes respectively, meaning "in context Γ with positive occurrences of variables in A_+ and negative occurrences of variables in A_- , type T or pretype R is well formed". See Figure 7 in the appendix for the complete set of rules, presented in a locally-nameless style as in the mechanization. Then, the usual well-formedness judgements for types and pretypes are defined as follows:

```
\Gamma \vdash_T T \mathbf{wf} := \Gamma; \mathbf{dom}(\Gamma); \mathbf{dom}(\Gamma) \vdash_T T \mathbf{wf}
\Gamma \vdash_R R \mathbf{wf} := \Gamma; \mathbf{dom}(\Gamma); \mathbf{dom}(\Gamma) \vdash_R R \mathbf{wf}
```

With the addition of boxes, the well-formedness judgement can be expressed in System $CC_{<:\square}$ in a way that is

$$\frac{\forall i \in \mathbb{N} \quad i \notin C}{C \text{ capt}} \text{ (CAPT)} \quad \frac{C \text{ capt}}{CR \text{ type}} \text{ (CAPT-TYPE)}$$

$$\frac{R \text{ pure}}{R \text{ type}} \text{ (PURE-TYPE)} \quad X \text{ pure} \text{ (TVAR-PURE)}$$

$$\top \text{ pure} \text{ (TOP-PURE)} \quad \frac{T \text{ type}}{\Box T \text{ pure}} \text{ (BOX-PURE)}$$

$$\frac{S \text{ type}}{\forall (S) \to T \text{ pure}} \text{ (ARR-PURE)}$$

$$\frac{R \text{ pure}}{\forall (R) \to T \text{ pure}} \text{ (ALL-PURE)}$$

Figure 3. type and **pure** judgements of System CC_{<:□}

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 Xwill be replaced with the captured variables of T, i.e. all variables contained in the capture sets in *T* that are not behind boxes. This feature is the reason why we need to parametrize the well-formedness judgement by atom sets. However, System CC_{<:□} 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 complete well-formedness judgement for System CC_{<:□} is defined in Figure 4. These design decision further cut down the length proof by a large amount, and simplified the overall mechanization process.

3.3 Subtyping

Another consequence of encoding the raw syntax of 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 8.

In the mechanized proof for System $CC_{\leq :\square}$, the new single subtyping judgement is described in Figure 9.

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, Cog cannot check that a naive mutually inductive proof

$$\frac{C \subseteq \mathbf{dom}(\Gamma) \cup \{\star\} \quad \Gamma \vdash R \text{ wf} \quad R \text{ pure}}{\Gamma \vdash CR \text{ wf}} \text{ (CAPT-WF)}$$

$$\frac{\Gamma \vdash T \text{ wf} \text{ (TOP-WF)}}{\Gamma \vdash X \text{ wf}} \frac{X <: T \in \Gamma}{\Gamma \vdash X \text{ wf}} \text{ (TVAR-WF)} \quad \frac{\Gamma \vdash CR \text{ wf}}{\Gamma \vdash \Box CR \text{ wf}} \text{ (BOX-WF)}$$

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

$$\frac{\Gamma \vdash R \text{ wf} \quad R \text{ pure} \quad (\forall X \notin L \quad \Gamma \vdash \{0 \to X\}T \text{ wf})}{\Gamma \vdash \forall [R] \to T \text{ wf}} \text{ (TFUN-WF)}$$
Figure 4. Locally-nameless presentation of the well-

formedness rules for types in System CC<:□

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 termination.

3.4 Reduction rules

The small-step semantics of System F_{<:} in the proof of Aydemir et al. [Aydemir et al. 2008] are specified using a standard binary relation on terms. However, due to the MNF structure of System $CC_{\leq:\Box}$, we can not substitute a term for another in general, we instead rely on a three-state abstract machine, similar to the CEK machine [Felleisen and Friedman 1986]. 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_{<:□} semantics illustrated in Figure 10.

Working with holed contexts as in Figure 10 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 CC<:□ is given in Figure 5. Note that the only additional rule compared to the holed-context-based reductions in Figure 10 is the (Let) rule which simply builds up the evaluation context if the current focused expression is a let-binding.

Conclusion

The mechanization of the soundness proof of System CC_{<:□} 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, as

$$\frac{x = \lambda(T)e \in S \qquad y = v \in S}{\langle S \mid E \mid xy \rangle \to \langle S \mid E \mid [0 \to y]e \rangle} \text{ (APP)}$$

$$\frac{x = \Lambda[R]e \in S \qquad R \text{ pure}}{\langle S \mid E \mid x[R] \rangle \to \langle S \mid E \mid \{0 \to R\}e \rangle} \text{ (TAPP)}$$

$$\frac{x = \Box y \in S}{\langle S \mid E \mid C \hookrightarrow x \rangle \to \langle S \mid E \mid y \rangle} \text{ (OPEN)}$$

$$\frac{x = v \in S}{\langle S \mid e :: E \mid x \rangle \to \langle S \mid E \mid [0 \to x]e \rangle} \text{ (RENAME)}$$

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

$$\langle S \mid E \mid \text{let } e_1 \text{ in } e_2 \rangle \to \langle S \mid e_2 :: E \mid e_1 \rangle \text{ (LET)}$$

Figure 5. Reduction rules for the System $CC_{<:\Box}$ abstract machine used in the mechanization

described in [Xu and Odersky 2023]. Nevertheless, the mechanization of System CC_{<:□} 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.

5 Future Work

5.1 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 somehow 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 [Aydemir et al. 2008] 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 $x \notin ...$ 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.

5.2 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 [Schäfer et al. 2015]. AutoSubst is a Coq library tailored for automating the proof of substitution lemmas, which account for a large portion of the overall proof.

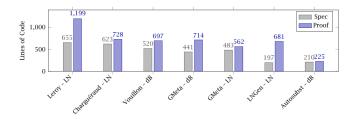


Figure 6. Comparison of POPLMARK challenge submissions (image taken from [Schäfer et al. 2015])

Our current mechanization is based on Arthur Charguéraud's locally nameless proof [Charguéraud 2012], 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.

References

Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack, and Stephanie Weirich. 2008. "Engineering formal metatheory." *Acm sigplan notices*, 43, 1, 3–15.

Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. "Tracking Captured Variables in Types." arXiv preprint arXiv:2105.11896.

Jonathan Immanuel Brachthäuser, Aleksander Slawomir Boruch-Gruszecki, and Martin Odersky. 2021. "Representing Monads with Capabilities." In: HOPE 2021 Workshop POST_TALK.

Arthur Charguéraud. 2012. "The locally nameless representation." Journal of automated reasoning, 49, 3, 363–408.

Matthias Felleisen and Daniel P Friedman. 1986. Control Operators, the SECD-machine, and the [1]-calculus. Indiana University, Computer Science Department.

Andrzej Filinski. 1999. "Representing Layered Monads." In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99). Association for Computing Machinery, San Antonio, Texas, USA, 175–188. ISBN: 1581130953. DOI: 10.1145/292540.292557.

Andrzej Filinski. 1994. "Representing Monads." In: (POPL '94), 446–457.
John Hatcliff and Olivier Danvy. 1994. "A generic account of continuation-passing styles." In: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 458–471.

Oleg Kiselyov and Hiromi Ishii. 2015. "Freer monads, more extensible effects." ACM SIGPLAN Notices, 50, 12, 94–105.

Daan Leijen. 2014. "Koka: Programming with row polymorphic effect types." arXiv preprint arXiv:1406.2061.

Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. 2021. "Safer exceptions for Scala." In: Proceedings of the 12th ACM SIGPLAN International Symposium on Scala, 1–11. Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, A and Ondřej Lhoták. 2022. "Scoped capabilities for polymorphic effects." arXiv preprint arXiv:2207.03402.

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

Marianna Rapoport and Ondřej Lhoták. 2019. "A path to DOT: formalizing fully path-dependent types." arXiv preprint arXiv:1904.07298.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. "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, 359–374.

Mads Tofte and Jean-Pierre Talpin. 1997. "Region-based memory management." *Information and computation*, 132, 2, 109–176.

Yichen Xu and Martin Odersky. 2023. "Formalizing Box Inference for Capture Calculus." arXiv preprint arXiv:2306.06496.

Appendix

$$C \subseteq A_{+} \qquad \Gamma; A_{+}; A_{-} \vdash_{R} U \text{ wf}$$

$$\forall x_{i} \in C \text{ } x_{i} : S_{i} \in \Gamma$$

$$\Gamma; A_{+}; A_{-} \vdash_{T} CU \text{ wf}$$

$$\frac{\Gamma; A_{+}; A_{-} \vdash_{R} U \text{ wf}}{\Gamma; A_{+}; A_{-} \vdash_{T} \{ \star \} U \text{ wf}} \text{ (universe-wf)}$$

$$\frac{X <: T \in \Gamma}{\Gamma; A_{+}; A_{-} \vdash_{R} X \text{ wf}} \text{ (TVAR-WF)}$$

$$\frac{\Gamma; A_{-}; A_{+} \vdash_{T} S \text{ wf}}{\Gamma; A_{+}; A_{-} \vdash_{R} X \text{ vf}} \text{ (FUN-WF)}$$

$$\frac{(\forall x \notin L \quad \Gamma; A_{+} \cup \{x\}; A_{-} \vdash_{T} [0 \to x] T \text{ wf})}{\Gamma; A_{+}; A_{-} \vdash_{R} \forall (S) \to T \text{ wf}} \text{ (FUN-WF)}$$

$$\frac{\Gamma; A_{-}; A_{+} \vdash_{T} S \text{ wf}}{\Gamma; A_{+}; A_{-} \vdash_{R} \forall [S] \to T \text{ wf}} \text{ (TFUN-WF)}$$

$$\Gamma; A_{+}; A_{-} \vdash_{R} \forall [S] \to T \text{ wf}$$

Figure 7. Locally-nameless presentation of well-formedness rules for types (\vdash_T) and pretypes (\vdash_R) in System CF<:

Received 12 July 2023

$$\frac{\Gamma \text{ wf} \qquad \Gamma \vdash_{T} X \text{ wf}}{\Gamma \vdash_{T} X <: X} \text{ (sub-refl-tvar)} \qquad \frac{X <: S \in \Gamma \qquad \Gamma \vdash_{T} S <: T}{\Gamma \vdash_{T} X <: T} \text{ (sub-trans-tvar)}$$

$$\frac{\Gamma \vdash_{C} C_{2} \qquad \Gamma \vdash_{R} R_{1} <: R_{2}}{\Gamma \vdash_{T} C_{1}R_{1} <: C_{2}R_{2}} \text{ (sub-capt)} \qquad \frac{\Gamma \text{ wf} \qquad \Gamma \vdash_{R} T \text{ wf}}{\Gamma \vdash_{R} T <: T} \text{ (sub-top)}$$

$$\frac{\Gamma \vdash_{T} S_{2} <: S_{1} \qquad \Gamma \vdash_{T} S_{1} \text{ wf} \qquad \Gamma \vdash_{T} S_{2} \text{ wf}}{(\forall x \notin L \quad \Gamma, x : S_{1}; \text{dom}(E) \cup \{x\}; \text{dom}(E) \vdash_{T} \{0 \to \{x\}\}T_{1} \text{ wf})} \text{ (}\forall x \notin L \quad \Gamma, x : S_{2}; \text{dom}(E) \cup \{x\}; \text{dom}(E) \vdash_{T} \{0 \to \{x\}\}T_{2} \text{ wf})}$$

$$\frac{(\forall x \notin L \quad \Gamma, x : S_{2} \vdash_{T} \{0 \to \{x\}\}T_{1} <: \{0 \to \{x\}\}T_{2})}{\Gamma \vdash_{R} \forall (S_{1}) \to T_{1} <: \forall (S_{2}) \to T_{2}} \text{ (sub-fun)}$$

$$\Gamma \vdash_{T} S_{2} <: S_{1} \qquad \Gamma \vdash_{T} S_{1} \text{ wf} \qquad \Gamma \vdash_{T} S_{2} \text{ wf}} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{1}; \text{dom}(E) \cup \{X\}; \text{dom}(E) \cup \{X\} \vdash_{T} \{0 \to \{X\}\}T_{1} \text{ wf})} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{2}; \text{dom}(E) \cup \{X\}; \text{dom}(E) \cup \{X\} \vdash_{T} \{0 \to \{X\}\}T_{2} \text{ wf})} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{2}; \text{dom}(E) \cup \{X\}; \text{dom}(E) \cup \{X\} \vdash_{T} \{0 \to \{X\}\}T_{2} \text{ wf})} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{2}; \text{dom}(E) \cup \{X\}; \text{dom}(E) \cup \{X\} \vdash_{T} \{0 \to \{X\}\}T_{2} \text{ wf})} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{2}; \text{dom}(E) \cup \{X\}; \text{dom}(E) \cup \{X\}\}T_{1} <: \{0 \to \{X\}\}T_{2} \text{ wf})} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{2} \vdash_{T} \{0 \to \{X\}\}T_{1} <: \{0 \to \{X\}\}T_{2} \text{ wf})} \text{ (}\forall X \notin L \quad \Gamma, X <: S_{2}; \text{dom}(E) \cup \{X\}; \text{dom}(E) \cup \{X\};$$

Figure 8. Locally-nameless presentation of the subtyping rules for types (\vdash_T) and pretypes (\vdash_R) in System CF $_{<:}$

$$\frac{\Gamma \text{ wf} \qquad \Gamma \vdash X \text{ wf}}{\Gamma \vdash X <: X} \text{ (sub-refl-tvar)} \qquad \frac{X <: R \in \Gamma \qquad \Gamma \vdash R <: S}{\Gamma \vdash X <: S} \text{ (sub-trans-tvar)} \qquad \frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \Box T_1 <: \Box T_2} \text{ (sub-box)}$$

$$\frac{\Gamma \vdash C_1 <:_C C_2 \qquad \Gamma \vdash R_1 <: R_2 \qquad R_1 \text{ pure} \qquad R_2 \text{ pure}}{\Gamma \vdash C_1 R_1 <: C_2 R_2} \text{ (sub-capt)} \qquad \frac{\Gamma \text{ wf} \qquad \Gamma \vdash R \text{ wf} \qquad R \text{ pure}}{\Gamma \vdash R <: T} \text{ (sub-top)}$$

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

$$\frac{\Gamma \vdash R_2 <: R_1 \qquad (\forall X \notin L \quad \Gamma, X <: R_2 \vdash \{0 \to \{X\}\}T_1 <: \{0 \to \{X\}\}T_2)}{\Gamma \vdash \forall [R_1] \to T_1 <: \forall [R_2] \to T_2} \text{ (sub-tfun)}$$

Figure 9. Locally-nameless presentation of the subtyping rules in System $CC_{<:\Box}$

```
Store context: S := [] \mid \text{let } x = v \text{ in } S

Evaluation context: E := [] \mid \text{let } x = E \text{ in } e

S[E[xy]] \longrightarrow S[E[[z \to y]e]] \quad \text{if } S(x) = \lambda(z : T). \ e \quad \text{(APP)}

S[E[x[T]]] \longrightarrow S[E[X \to T]e] \quad \text{if } S(x) = \Lambda[X <: T]. \ e \quad \text{(TAPP)}

S[E[C \hookrightarrow x] \longrightarrow S[E[y]] \quad \text{if } S(x) = \Box y \quad \text{(OPEN)}

S[E[\text{let } x = y \text{ in } e]] \longrightarrow S[E[[x \to y]e]] \quad \text{(RENAME)}

S[E[\text{let } x = y \text{ in } v]] \longrightarrow S[\text{let } x = v \text{ in } E[e]] \quad \text{if } E \neq [] \quad \text{(LIFT)}
```

Figure 10. Reduction rules for System CC_{<:□} as defined in [Odersky, Boruch-Gruszecki, Lee, et al. 2022]