# Dependency-Free Capture Tracking

Edward Lee
Computer Science
University of Waterloo
Waterloo, ON, Canada

Kavin Satheeskumar
Computer Science
University of Waterloo
Waterloo, ON, Canada

Ondřej Lhoták
Computer Science
University of Waterloo
Waterloo, ON, Canada

## ABSTRACT

Type systems usually characterize the shapes of values but not usually their free variables. Many desirable safety properties could be guaranteed by the type system if it knew exactly which variables were free in values.

There has been much recent work investigating such systems, with an eventual goal of incorporating a capture tracking system into Scala. These systems are unfortunately complicated by advanced features in Scala's type system, particularly dependent types. We explore what a capture tracking system could look like without the full complication of dependent types.

## CCS CONCEPTS

• **Software and its engineering** → **General programming languages**; **Compilers**.

## KEYWORDS

System $F_{<:}$, Capture Tracking, Type Systems

**ACM Reference Format:**
Edward Lee, Kavin Satheeskumar, and Ondřej Lhoták. 2023. Dependency-Free Capture Tracking. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3605156.3606454

## 1 INTRODUCTION

Type systems usually characterize the shapes of values but not their free variables. This is unfortunate, as many desirable safety properties could be guaranteed by the type system if it knew exactly which variables were free in values.

For example, using the *effects-as-capabilities* discipline, one can reduce the problem of tracking where potentially unsafe effects could happen to the problem of tracking where *capabilities* corresponding to those effects could flow to in the program. There is an ongoing effort to incorporate such a system into Scala, and a working prototype has been implemented in the Dotty compiler – see Odersky et al. [5].

Formalizing such a system has been difficult. The problem is that a free variable tracking system is inherently dependently typed, in that types track (term-level) variables. Consider an example Scala definition:

```
def delayedPrint(s: {*} String) = () => print(s)
```

Here, the `*` in the type of `s` specifies that capturing `s` is to be tracked by the type system. Thus, the return type of `delayedPrint` could be `{s} Unit => Unit` to indicate that the closure that is returned captures `s`. A type like `print (s : {*} String) => {s} Unit => Unit` will always unfortunately contain term variables.

One attempt to formalize a tracking system by Boruch-Gruszecki et al. [3] runs into complications concerning dependent types, which they solve by enforcing variance restrictions on where type variables can occur. More recently, Odersky et al. [6] sidestep these issues by working in administrative normal form (ANF), a technique previously used by Rapoport and Lhoták [7] and Amin et al. [1] to work around similar complications in formalizing type dependency in DOT, Scala's core calculus. However, the indirection of ANF makes it difficult to acquire an intuition for the properties of the system.

This does not need to be the case, though. As Brachthäuser et al. [4] show, a capture tracking system can be done with a simple core calculus with a simple mechanized proof soundness even in the presence of dependent types. While their system lacks subtyping, their system differs in one key way compared to Odersky et al. [6] and Boruch-Gruszecki et al. [3] which enables a simple soundness proof to go through while still keeping track of captured variables in types. They annotate their function applications with the capture information which should be substituted instead of relying on the computed capture information on the actual value being substituted in.

Inspired by Brachthäuser et al. [4], we show that a similar technique also works in systems like Boruch-Gruszecki et al. [3] and Odersky et al. [6]. We present a simple calculus System $F_{<:C}$ that extends System $F_{<:}$ with capture tracking without resorting to ANF nor to variance restrictions on type variables, for which we have mechanized a soundness proof, and we sketch out how System $F_{<:C}$ can be used as a basis of soundness proofs for a surface syntax in a practical programming language. Our development has been mechanized in Coq, and can be found at https://github.com/e45lee/simple-capture-proof.

We hope that with this contribution, capture tracking systems will be better understood, and that System $F_{<:C}$ can serve as the basis of a proof of soundness for capture tracking systems – not only in Scala, but for other languages as well.

## 2 CAPTURE TRACKING

At its core, capture tracking is concerned with:

(1) Tracking what free variables are present in values, and,
(2) For functions, tracking how arguments flow into results.


*FTfJP '23, July 18, 2023, Seattle, WA, USA*

ACM ISBN 979-8-4007-0246-4/23/07…$15.00
https://doi.org/10.1145/3605156.3606454

For example, in our earlier example:

```
def delayedPrint(s: {*} String) = () => print(s)
```

The above term would receive the following type:

```
{print} (s : {*} String) => {s} Unit => Unit
```

This type describes that `delayedPrint` is a function that returns a thunk which captures the parameter `s` passed to `delayedPrint`.

Applying delayedPrint to a string returns a value whose type reflects that it captures that string. For example, in the following listing, `delayedPrint(hello)` has type `{hello} Unit => Unit`.

```
val hello: {*} String = "Hello␣World".
delayedPrint(hello) // returns thunk with type {hello}
    Unit => Unit
```

## 2.1 Dependent Types

Dependent types pose a problem. Consider the following example.

```
val goodbye: {*} String = "Goodbye␣World".
def wantsEqual = (s: {*} () => String) =>
                 (t: {s} () => String) => { ... }
wantsEqual
   (if true then () => hello else () => goodbye)
   (() => goodbye)
```

Here, we declare that goodbye is a `String` which should be tracked, and `wantsEqual` is a curried function that expects a value `s` that can capture anything and a value `t` that captures the same variables as `s` does. We then apply `wantsEqual` to a conditional expression that returns two thunks: one returning `hello` and one returning goodbye:

```
if true then () => hello else () => goodbye
```

Now, what should the type of this conditional term be? Both branches return a thunk returning a `String`, but one captures `hello` and the other goodbye. All we can say is that the conditional could possibly capture `hello` or goodbye, giving the conditional the type `{hello, goodbye} () => String`. Thus, `wantsEqual(if true then () => hello else () => goodbye)` has type `({hello, goodbye}() => String) => T` for some `T`. Accordingly, we widen the type of the second argument of `wantsEqual` from `{goodbye}() => String` to `{hello, goodbye}() => String`. The overall program is well typed.

Now, consider what happens after a single step of reduction, which reduces the conditional term to just the thunk `() => hello`. The overall program reduces to:

```
wantsEqual(() => hello)(() => goodbye) // what type does
    this function have?
```

The next step will be to reduce the application `wantsEqual(() => hello)`. As is standard for function application, we substitute the argument `() => hello` for the parameter `s` in the body of `wantsEqual`. But `s` occurs not only in the body of the function, but also in the type of the second parameter `t`, which is `{s} () => String`. What should be substituted for `s` in the type of `t`?

There are a few options. One option is to substitute with the actual set of variables that are captured by the value, namely the free variables of `() => hello`. This is given by the substitution rule:

$$(\lambda(x : T).t)v \longrightarrow t\ [x \mapsto_{\texttt{type}} fv(v)]\ [x \mapsto_{\texttt{term}} v]\ \text{(BETA-V-BAD)}$$

However, this cannot be done so easily. In our example, if we replaced `s` with `hello` in the types of the body of `wantsEqual`, we end up with the term:

```
((t: {hello} () => String) => ...)(() => goodbye)
```

This is obviously ill-typed, since the argument captures goodbye but is required by the parameter type to capture `hello`. The reduction in this example violates type preservation. What went wrong? In this example two things went wrong for this to transpire:

(1) The capture set that we had "assigned" to `s` shrank under reduction from `{hello, goodbye}` to just `{hello}`, and,
(2) We used `s` in a negative, contravariant position – namely, in the type annotation for what we expect to be passed in the second parameter `t`.

This breaks preservation under reduction, as contravariance flips the subtyping hierarchy. It would be unsound for `s` to shrink in this manner. It must either stay static or expand. In order for preservation to hold, we need to prevent this from happening...but how?

One possibility is to simply disallow contravariant occurrences of term variables in types, which is the approach taken by Boruch-Gruszecki et al. [3]. This is restrictive, however, and complicates the proof of soundness, which needs to enforce and check that term variables occur only covariantly in types, even under reduction.

Another possibility is to make what we "assign" to `s` static, by putting programs in administrative normal form, in effect assigning a name to every (sub)expression in a program:

```
val tmp =
    if true then () => hello else () => goodbye
wantsEqual(tmp) // expects {tmp} String
```

Now, since we have named the conditional term `tmp`, we can give `wantsEqual(tmp)` the type `({tmp} () => String) => T`. Since `tmp` is named, we can use the capture set `{tmp}` to represent exactly what `tmp` captures, even as what is stored in `tmp` shrinks under reduction from something capturing `{hello, goodbye}` to just `hello`.

```
val tmp = () => hello // reduced
wantsEqual(tmp) // still expects {tmp} String.
```

This is the approach taken by Odersky et al. [6]. Similar approaches have been used to handle variance issues in other contexts – for example, Rapoport and Lhoták [7] and Amin et al. [1]. However, administrative normal form adds complexity to the reduction rules. Moreover, it does not even improve expressiveness for this example. While `wantsEqual` can be typed, the only variable that can be passed to `wantsEqual(tmp)` is either `tmp` itself or a variable that does not capture any other tracked variables. In particular, the ANF-converted version of the original term `wantsEqual(tmp)(() => goodbye)` remains ill-typed.

In this paper, in Section 3, we explore a third approach, which is to annotate each term application in the original program with the *capture set* that will be substituted in place of that term variable in capture set position. That set is fixed in the original program and does not shrink with reduction. Formally, we replace (BETA-V-BAD) with the following:

$$(\lambda(x : T).t)\ [C]\ (v) \longrightarrow t\ [x \mapsto_{\texttt{type}} C]\ [x \mapsto_{\texttt{term}} v]\ \text{(BETA-V)}$$

This neatly sidesteps the issue of shrinking variables and variance, at the cost of annotating each term application with a capture set. The resulting system satisfies type preservation and allows us to type the `wantsEqual` example with the capture set annotation `{hello,goodbye}`:

```
wantsEqual
  {hello, goodbye}
    (if true then () => hello else () => goodbye)
  {hello, goodbye}(() => goodbye)
-->
wantsEqual
  {hello, goodbye}(() => hello)
  {hello, goodbye}(() => goodbye)
-->
(t: {hello, goodbye} () => String)
  {hello, goodbye}(() => goodbye)
```

The System C calculus of Brachthäuser et al. [4] also uses this approach, but in the context of a rather different calculus.

## 2.2 Types, Shapes, and Polymorphism

Another issue that arises is that of parametric polymorphism. What should a type variable `X` quantify over, now that types contain both information describing the *shape* of a value, as well as what that value *captures*. The seemingly obvious answer is that `X` should stand for both the shape and the capture information of a value, but this imposes some restrictions. For one, if `X` is a type variable that stands for a full type, how do we interpret a type like `{hello} X` that adds a capture set to a type variable? This is not so clear, especially after `X` has been substituted away for a type like `{goodbye} () => String`. What does the type `{hello}({goodbye} () => String)` even mean? Do we somehow combine the two capture sets `{hello}` and `{goodbye}`?

One solution is to prevent type variables from further being annotated by capture sets. This has the advantage of being simple conceptually at first glance. However, it comes at the cost of complexity in the calculus: Boruch-Gruszecki et al. [3] require two very distinct typing rules for term variables depending on what sort of type they are bound to. Furthermore, complications arise in the presence of generic data types. Consider an arbitrary list type `List[X]` describing a list containing elements of type `X`, and the generic `map` function over said lists. What should the type of map be – in particular, what should the return type of map be?

```
def map[X,Y](l: {*} List[X], f: {*} X => Y)
  : {???} List[Y]
```

Assuming map does not capture anything else, the values that could be captured in the resulting list must come from `f` and the elements of `l`. Naming the former is easy in a capture set, but not the latter, as the result does not capture `l` directly but the elements of `l`, which do not have any name in this scope. We could have the capture set `l` stand in for all its elements when `l` is given a `List` type, but that requires special support in the calculus for every data structure.

To sidestep this issue, Odersky et al. [6] require generic data structures to hold only pure elements with empty capture sets. This would be impractically restrictive, however, so they add a special mechanism called *boxing* to "tunnel" captures, by temporarily making impure values pure so they can be stored in a generic data structure, but marking their type to ensure that the necessary captures are covered by the type when the element is later retrieved from the data structure.

In this paper, as we discuss in Section 3, we use a more familiar mechanism, for which soundness proofs are well understood: an additional binding form for variables that stand for capture sets. Thus, we separate the binders for the shape part and the capture part of a type. In this setting, we could give `map` the following signature:

```
def map[C,X,Y](l: {*} List[{C} X], f: {*} X => Y)
  : {C, f} List[{C, f} Y]
```

Here, the capture set binder allows us to introduce a name `C` for the capture set of the elements of `l`.

# 3 A SIMPLER CALCULUS

To this end, we develop a simpler capture tracking calculus System $\mathsf{F}_{<:\mathtt{C}}$ inspired by these observations. As discussed in Section 2.3, we explicitly annotate term applications with the capture sets which should be substituted in. Specifically, we define application reduction to be as below:

$$(\lambda(x : T).t)\ [C]\ (v) \longrightarrow t\ [x \mapsto_{\mathtt{type}} C]\ [x \mapsto_{\mathtt{term}} v]\ \text{(BETA-V)}$$

Obviously, though, `C` needs to bound to what could be carried in `v` – this is handled through the typing rule (APP).

$$\frac{\Gamma \vdash t : \{D\}\ (x : \{C\}\ S) \to T \qquad \Gamma \vdash s : \{C\}\ S}{\Gamma \vdash t[C](s) : T[x \mapsto C]}\ \text{(APP)}$$

To handle parametric polymorphism, as discussed in Section 2.2, we adopt a split shape-type system with polymorphism on shape variables. We support capture polymorphism directly using a third binder $\Lambda(Y <: C).t$ for capture set variables.

This allows for straightforward mechanized proofs of Progress and Preservation, without requiring variance restrictions or administrative normal form.

Theorem 3.1 (Preservation). *Suppose* $\Gamma \vdash s : T$, *and* $s \longrightarrow t$. *Then* $\Gamma \vdash t : T$ *as well.*

Theorem 3.2 (Progress). *Suppose* $\varnothing \vdash s : T$. *Either* $s$ *is a value, or* $s \longrightarrow t$ *for some term* $t$.

Our development has been mechanized in Coq, based on the mechanization System $\mathsf{F}_{<:}$ of Aydemir et al. [2]. While there are are many lemmas in our mechanization, due to our three binding forms, all of those proofs were straightforward and were not difficult to discharge. The only proofs that were difficult were those that dealt with sets, which is mainly due to lack of good automation for working with sets in Coq.

## 3.1 The Burden of Annotations?

One might object to the extra annotation required on term application. After all, one has to find a `C` – a capture set – to fill in there. However, this is not such an onerous requirement. As Brachthäuser et al. [4] observe, one infers the appropriate annotation anyways while type checking. Concretely, with the same typing rules, we could devise a calculus System $\mathsf{F}_{<:\mathtt{CO}}$, without reduction rules, but with almost the same syntax and typing rules, with the following two changes to syntax and typing:

| | | | |
|---|---|---|---|
| $s, t$ | ::= | | **Terms** |
| | \| | $\lambda(x : T).t$ | term abstraction |
| | \| | $\Lambda(X <: S).t$ | shape abstraction |
| | \| | $\Lambda(Y <: C).t$ | capture abstraction |
| | \| | $x$ | term variable |
| | \| | $s[C](t)$ | application |
| | \| | $s[S]$ | shape application |
| | \| | $s[C]$ | capture application |
| $\Gamma$ | ::= | | **Environment** |
| | \| | $\cdot$ | empty |
| | \| | $\Gamma,\ x : T$ | term binding |
| | \| | $\Gamma,\ X <: S$ | shape binding |
| | \| | $\Gamma,\ Y <: C$ | capture binding |

| | | | |
|---|---|---|---|
| $S$ | ::= | | **Shapes** |
| | \| | $X$ | shape variable |
| | \| | $(x : T_1) \to T_2$ | function shapes |
| | \| | $\forall(X <: S).T$ | for-all shapes |
| | \| | $\forall(Y <: C).T$ | capture for-all shapes |
| $T$ | ::= | | **Types** |
| | \| | $\{C\}\ S$ | capturing type |
| $C, D$ | ::= | | **Capture Sets** |
| | \| | $\{x_1, x_2, \ldots, Y_1, Y_2, \ldots\}$ | proper set |
| | \| | $\top$ | universal set |
| | where | $Y$ | capture variable |

**Figure 1: The syntax of System $\mathsf{F}_{<:\mathsf{C}}$.**

**Subcapturing** $\boxed{\Gamma \vdash C <: D}$

$$\frac{}{\Gamma \vdash C <: \top} \quad \text{(SC-UNIV)}$$

$$\frac{x \in C}{\Gamma \vdash \{x\} <: C} \quad \text{(SC-IN)}$$

$$\frac{\forall x \in C,\ \Gamma \vdash \{x\} <: D}{\Gamma \vdash C <: D} \quad \text{(SC-EXPAND)}$$

$$\frac{x : \{C\}\ S \in \Gamma \qquad \Gamma \vdash C <: D}{\Gamma \vdash \{x\} <: D} \quad \text{(SC-VAR)}$$

$$\frac{x <: C \in \Gamma \qquad \Gamma \vdash C <: D}{\Gamma \vdash \{x\} <: D} \quad \text{(SC-CVAR)}$$

**Figure 2: Subcapturing rules of System $\mathsf{F}_{<:\mathsf{C}}$.**

(1) Term application omits the capture set annotation – $f(v)$ instead of $f[C](v)$.
(2) The typing judgment is updated to reflect this:

**Subtyping** $\boxed{\Gamma \vdash S_1 <: S_1 \text{ and } \Gamma \vdash T_1 <: T_2}$

$$\Gamma \vdash S <: \top \quad \text{(SUB-TOP)}$$

$$\frac{X \in \Gamma}{\Gamma \vdash X <: X} \quad \text{(SUB-REFL-SVAR)}$$

$$\frac{X <: S_1 \in \Gamma \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash X <: S_2} \quad \text{(SUB-TVAR)}$$

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash \{C_1\}\ S_1 <: \{C_2\}\ S_2} \quad \text{(SUB-TYPE)}$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma, x : T_1 \vdash T_3 <: T_4}{\Gamma \vdash (x : T_1) \to T_3 <: (x : T_2) \to T_4} \quad \text{(SUB-ARROW)}$$

$$\frac{\Gamma \vdash S_1 <: S_2 \qquad \Gamma, X <: S_1 \vdash T_1 <: T_2}{\Gamma \vdash \forall(X <: S_1).T_1 <: \forall(X <: S_2).T_2} \quad \text{(SUB-ALL)}$$

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma, Y <: C_1 \vdash T_1 <: T_2}{\Gamma \vdash \forall(Y <: C_1).T_1 <: \forall(X <: C_2).T_2} \quad \text{(SUB-CALL)}$$

**Figure 3: Subtyping rules of System $\mathsf{F}_{<:\mathsf{C}}$.**

**Typing** $\boxed{\Gamma \vdash t : T}$

$$\frac{x : \{C\}\ S \in \Gamma}{\Gamma \vdash x : \{x\}\ S} \quad \text{(VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x : T_1).t : \{fv(t) - x\}\ (x : T_1) \to T_2} \quad \text{(ABS)}$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(X <: S).T : \{fv(t)\}\ \forall(X <: S).T} \quad \text{(S-ABS)}$$

$$\frac{\Gamma, Y <: C \vdash t : T}{\Gamma \vdash \Lambda(Y <: C).T : \{fv(t)\}\ \forall(Y <: C).T} \quad \text{(C-ABS)}$$

$$\frac{\Gamma \vdash t : \{D\}\ (x : \{C\}\ S) \to T \qquad \Gamma \vdash s : \{C\}\ S}{\Gamma \vdash t[C](s) : T[x \mapsto C]} \quad \text{(APP)}$$

$$\frac{\Gamma \vdash t : \{D\}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \vdash t[S'] : T[X \mapsto S']} \quad \text{(S-APP)}$$

$$\frac{\Gamma \vdash t : \{D\}\ \forall(Y <: C).T \qquad \Gamma \vdash C' <: C}{\Gamma \vdash t[C'] : T[Y \mapsto C']} \quad \text{(C-APP)}$$

**Figure 4: Typing rules for System $\mathsf{F}_{<:\mathsf{C}}$**

**Evaluation** $\boxed{s \longrightarrow t}$

$$(\lambda(x : T).t)[C](v) \longrightarrow t\ [x \mapsto_{\texttt{type}} C]\ [x \mapsto_{\texttt{term}} v] \quad \text{(BETA-V)}$$

$$(\Lambda(X <: S_b).t)[S] \longrightarrow t[X \mapsto S] \quad \text{(BETA-S)}$$

$$(\Lambda(Y <: C_b).t)[C] \longrightarrow t[Y \mapsto C] \quad \text{(BETA-C)}$$

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \quad \text{(CONTEXT)}$$

$$\begin{array}{rcll} E & ::= & [] \mid E[C](t) \mid v[C](E) & \textbf{Evaluation Context} \\ & \mid & E[S] & \\ & \mid & E[C] & \end{array}$$

**Figure 5: Reduction rules for System $\mathsf{F}_{<:\mathtt{C}}$**

$$\frac{\Gamma \vdash t : \{D\}\ (x : \{C\}\ S) \to T \qquad \Gamma \vdash s : \{C\}\ S}{\Gamma \vdash t(s) : T[x \mapsto C]} \quad \text{(APP-OMIT)}$$

At the surface level, System $\mathsf{F}_{<:\mathtt{CO}}$ looks almost like a dependently-typed calculus without capture annotations in the same style as Boruch-Gruszecki et al. [3]. This allows System $\mathsf{F}_{<:\mathtt{CO}}$ to serve as the basis for the surface-level syntax of a capture tracking system in a programming language, like Scala. It does not contain annotations like System $\mathsf{F}_{<:\mathtt{C}}$. But System $\mathsf{F}_{<:\mathtt{C}}$ can be used to show that well-typed System $\mathsf{F}_{<:\mathtt{CO}}$ programs behave properly as well. A typing derivation for a System $\mathsf{F}_{<:\mathtt{CO}}$ program is in one-to-one correspondence with a typing derivation for a System $\mathsf{F}_{<:\mathtt{C}}$ program – simply lift out the Cs that appear in the (APP-OMIT) rule into the program itself. Progress and preservation in System $\mathsf{F}_{<:\mathtt{C}}$ guarantee that the resulting System $\mathsf{F}_{<:\mathtt{C}}$ program will reduce properly and not get stuck. Note, however, that the program will reduce according to the *inferred capture sets* on term application, unlike previous dependently-typed capture tracking calculi. This avoids the restrictions that Odersky et al. [6] and Boruch-Gruszecki et al. [3] imposed on their systems.

## 4 FUTURE WORK

System $\mathsf{F}_{<:\mathtt{C}}$ contributes a simple and sound basis for an expressive capture tracking system in a System $\mathsf{F}_{<:}$-based programming language. It does so by adding capture set annotations at every term application. Some existing systems share a similar property, in that the capture set to be substituted when the term is applied can be inferred by simply looking at the term. Notably, this is true of Odersky et al. [6]'s System $\mathsf{CC}_{<:\Box}$. Term application in their ANF-based calculus reduces as follows:

$$\frac{f \text{ is bound to } \lambda x.t \text{ in the enclosing ANF context}}{f(y) \longrightarrow t[x \mapsto_{\texttt{type}} \{y\}][x \mapsto_{\texttt{term}} y]} \quad \text{(BETA-V-VAR)}$$

It would be interesting to explore whether our simpler System $\mathsf{F}_{<:\mathtt{C}}$ could serve as a basis for showing the soundness of System $\mathsf{CC}_{<:\Box}$. More generally, it would be interesting to see if System $\mathsf{F}_{<:\mathtt{C}}$ could serve as the basis of soundness proofs for other capture tracking systems which may come about in the future.

## 5 CONCLUSION

We contributed a *simple* and *sound* treatment of capture tracking as an extension of System $\mathsf{F}_{<:}$ without the complications involving dependent types that arise in existing attempts. We show that simply annotating term applications with static capture sets allows for a simpler treatment of soundness in a core calculus, without the need for variance restrictions or ANF. We hope that this will contribute to better intuitive understanding of capture tracking systems and that System $\mathsf{F}_{<:\mathtt{C}}$ can serve as the basis of a proof of soundness for capture tracking systems – not only for Scala, but for other languages as well.

## REFERENCES


[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 249–272.

[2] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

[3] Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondřej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. arXiv:2105.11896 [cs.PL]

[4] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-Based Reasoning to Type-Based Reasoning and Back. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 76 (apr 2022), 30 pages. https://doi.org/10.1145/3527320

[5] 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* (Chicago, IL, USA) *(SCALA 2021)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3486610.3486893

[6] Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. 2022. Scoped Capabilities for Polymorphic Effects. arXiv:2207.03402 [cs.PL]

[7] Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-Dependent Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 145 (oct 2019), 29 pages. https://doi.org/10.1145/3360571


Received 2023-05-26; accepted 2023-06-23