Effects, Capabilities, and Boxes

From Scope-based Reasoning to Type-based Reasoning and Back

JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany
PHILIPP SCHUSTER, University of Tübingen, Germany
EDWARD LEE, University of Waterloo, Canada
ALEKSANDER BORUCH-GRUSZECKI, EPFL, Switzerland

Reasoning about the use of external resources is an important aspect of many practical applications. Effect systems enable tracking such information in types, but at the cost of complicating signatures of common functions. Capabilities coupled with escape analysis offer safety and natural signatures, but are often overly coarse grained and restrictive. We present System C, which builds on and generalizes ideas from type-based escape analysis and demonstrates that capabilities and effects can be reconciled harmoniously. By assuming that all functions are *second class*, we can admit natural signatures for many common programs. By introducing a notion of *boxed values*, we can lift the restrictions of second-class values at the cost of needing to track degree-of-impurity information in types. The system we present is expressive enough to support effect handlers in full capacity. We practically evaluate System C in an implementation and prove its soundness.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Abstraction, modeling and modularity**; • **Theory of computation** → **Type theory**.

ACM Reference Format:
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 (April 2022), 35 pages. https://doi.org/10.1145/3527320

1 INTRODUCTION

Programming languages have to provide the ability to communicate with the outside world. Moreover, programs often need to non-locally interact with other parts of the program, for instance via mutable state or exceptions. If a program depends on or modifies its context, it is *effectful*, otherwise, it is *pure*. We say that effectful programs *use an effect*. Unrestricted or undisciplined use of effects can lead to confusion and bugs [Coblenz et al. 2016]. To address this, language designers have sought to enable programmers to *statically* and *locally* reason about the use of effects.

1.1 Effect Systems and Type-based Reasoning

Effect systems extend the static guarantees of type systems to additionally track the use of effects [Lucassen and Gifford 1988; Nielson et al. 1999; Plotkin and Power 2003; Tofte and Talpin 1997]. Typically, this additional information of the effect system (on the left) is also reflected in the type of functions, which mention the set of effects a function might use (on the right).

$$\Gamma \vdash s : \tau \;/\; \{\, \mathsf{Exc},\ \mathsf{State} \,\} \qquad\qquad \tau \to \tau \;/\; \{\, \mathsf{Exc},\ \mathsf{State} \,\}$$

Based on the types, programmers can use this additional information to reason about programs. For example, functions with an empty effect set are pure and can be executed in parallel without causing data races. From a programmer's perspective, however, effect systems usually have a number of

2475-1421/2022/4-ART76
https://doi.org/10.1145/3527320

drawbacks, which inhibit a more widespread adoption. In particular, by enhancing function types with effects such systems often track *too much* information. Types quickly become verbose, difficult to understand, and difficult to reason about – especially in the presence of effect-polymorphic higher-order functions [Brachthäuser et al. 2020a; Rytz et al. 2012; Zhang et al. 2016]. Consequently, programmers avoid effect systems and some languages, such as Scala, avoid adding an effect system.

1.2 Effects as Capabilities and Scope-based Reasoning

Capabilities offer an alternative way to control the use of effects. In this model, one can use certain effects only through capabilities [Dennis and Van Horn 1966; Miller 2006]. Restricting access to capabilities restricts effects. A program, such as s below, can only perform effects of capabilities it has access to. Similarly, the function type in the middle requires the two capabilities as arguments.

$$\Gamma,\ \mathsf{ex} :\ \mathsf{Exc},\ \mathsf{st} :\ \mathsf{State}\ \vdash s\ :\ \tau \qquad\qquad (\tau,\ \mathsf{Exc},\ \mathsf{State}) \to \tau \qquad\qquad\qquad \tau \to \tau$$

From a language designer's perspective, capabilities offer an interesting alternative to traditional effect systems: programmers can reason about effects the same way they reason about bindings. Additionally, it has been shown that capabilities offer a lightweight alternative to traditional effect polymorphism: *contextual effect polymorphism* [Brachthäuser et al. 2020a; Osvald et al. 2016]. Functions can use effects by closing over capabilities. These are not visible in the type of the function (right column), simplifying signatures of effect polymorphic higher-order functions [Brachthäuser et al. 2020a]. However, since since closure over capabilities is not visible in a function's type, it often hinders reasoning about its purity.

Some capabilities have a limited lifetime, like when modeling checked exceptions, and should not leave a particular scope. The problem is non-trivial, since leaving a scope can also occur indirectly via functions that *close over* capabilities. In an attempt to rule this out and guarantee effect safety, type-based escape analysis [Hannan 1998; Osvald et al. 2016] distinguishes between first- and *second-class functions*. Capabilities and functions closing over them are second-class. They can be passed as arguments, but cannot be returned nor stored in data structures or mutable references. This restriction rules out a large class of programs, which are safe but not typable. For instance, since second-class functions cannot be returned, currying cannot be applied.

1.3 Explicit Boxing – From Scope-based to Type-based Reasoning and Back

In this paper, we set out to restore the expressivity of first-class functions and type-based reasoning about purity, without sacrificing the simplicity of contextual effect polymorphism.

As a starting point, we choose a core language with support for contextual effect polymorphism via second class capabilities – System Ξ – [Brachthäuser et al. 2020a] and extend it with support for first-class functions[1]. We require a possible solution to meet the following criteria:

- *Backwards compatibility.* Types assigned by System Ξ should not change in the extension. This entails that ergonomic advantages of lightweight effect polymorphism remain.
- *Pay-as-you-go.* Only when treating functions in a first-class way, programmers should be confronted with additional complexity in the involved types.

We present System C, which aims at striking the balance between ergonomics (we offer the same form of lexical reasoning and contextual effect polymorphism as System Ξ) and expressivity (we additionally allow returning functions which close over capabilities and support type-based reasoning). Our solution is based on the following design decisions:

[1] While effect inference might be desirable in the future, we believe that explicit capability-passing is a viable option for surface languages (as witnessed by languages like Wyvern [Melicher et al. 2017]).

Second-class values. Following Osvald et al. [2016], and like System Ξ, we distinguish between functions that can be treated as first-class values, and functions that are second-class (to highlight this difference, we follow Brachthäuser et al. and explicitly refer to second-class functions as *blocks*). Thus, we avoid confronting programmers with the ceremony associated with tracking capabilities in types as much as possible. In particular, blocks can freely close over capabilities and effectful computations can simply use all capabilities in their lexical scope, with no *visible* type-level machinery to keep track of either fact.

Capability sets. Based on the work by Osvald et al., we annotate each binding in the typing context with additional information. However, we do not only track whether a bound variable is first- or second-class, but track precisely over *which capabilities* it closes. That is, we augment bindings (*e.g.*, $f :^{C} \;\; \sigma$) in the typing context with *capability sets* (*e.g.* C). This information is *only* annotated at the binder and is *not* part of the type. This is important for ergonomics: the additional information, which is used to guarantee effect safety, is not visible to users unless explicitly requested.

Boxes. While blocks can freely close over capabilities and other blocks, they cannot be returned from a function or stored in a field. To recover these abilities, System C features explicit boxing and unboxing language constructs. They are inspired by equally named modal connectives and by the work of Choudhury and Krishnaswami [2020] on comonadic type systems. Boxing converts a second-class value to a first-class value, reifying the contextual information annotated on the binder into the boxed value's type (*e.g.*, $f :^{C} \;\; \sigma \vdash \;\; \textbf{box}\, f \;\; : \sigma \; \textbf{at} \; C$). That is, instead of completely preventing first-class values from closing over capabilities, the capabilities they close over are now faithfully represented in their types. To use a boxed block, we have to unbox it. We make sure to only perform this operation when the capabilities are still in scope, which guarantees effect safety (*e.g.*, $x : \;\; \sigma \; \textbf{at} \; C \vdash \;\; \textbf{unbox}\, x \;\; : \sigma \mid C$). The reader might find the following analogy helpful:

> Conceptually, we treat *mentioning* capabilities as an *effect.* In the terminology of call-by-push-value [Levy 1999], boxing corresponds to "thunking" and unboxing corresponds to "forcing" the effect of mentioning capabilities[2].

The **box** and **unbox** constructs allow programmers to freely move between tracking capabilities implicitly, via lexical scoping, or explicitly, in the types.

1.4 Contributions and Overview

This paper makes the following contributions:

- An example driven introduction to programming in System C, a calculus that reconciles scope-based and type-based reasoning in a language with advanced control effects (Section 3).
- A formal presentation of System C with static and dynamic semantics (Section 4). The typing context in System C is enhanced with information about block binders, which only becomes visible in types when explicitly boxing blocks.
- A proof of progress and preservation (Theorems 4.2 and 4.4), and effect safety (Corollary 4.8).
- A full mechanization of the calculus, as well as proofs of the progress and preservation in the Coq theorem prover (Section 4.5.4).
- An evaluation in terms of an implementation (Section 5) and several small case studies. This paper is accompanied by an artifact consisting of an interactive demonstration and Coq proofs, archived under https://doi.org/10.5281/zenodo.5833713.

Furthermore, Section 2 provides an in-depth presentation of the state-of-the-art and motivates our work. Section 6 offers a comparison with additional lines of related work.

[2] A more detailed comparision with call-by-push-value can be found in Section 6.8.

2 MOTIVATION

The motivation behind our work is to design a language that specifically features:

Lexical reasoning. Programmers can determine lexically where an exception (or effect) is handled.
Effect safety. The type system establishes that all exceptions (all effects) are eventually handled.
Ergonomics. The verbosity of effect tracking in types is limited to where it is necessary.
First-class functions. It should be possible to return effectful functions.

No prior work that we are aware of meets all of the above criteria. In the remainder of this section, we will motivate each criterion and point out limitations of existing work. Readers who want to first learn more about our proposed solution can skip to Section 3 and can come back if necessary.

2.1 Lexical Reasoning

Operationally, traditional implementations of (control) effects (such as exceptions or the more general algebraic effects) are *dynamically scoped* [Brachthäuser and Leijen 2019]. Consider, for instance, how exceptions behave in JavaScript:

```
function process(path) {①
  function abort() { throw("processing aborted") }
  try {② eachLine(open(path), line ⇒ {③ /*...*/ abort() }) }
  catch { msg ⇒ /*... handle IO exception, raised by open ...*/ }
}
```

We define a function process that processes the contents of a file. To do so, it defines a local function abort that raises an exception, signalling that processing failed. Since opening a file might throw an exception, we additionally install an exception handler to deal with this error condition. We then call a higher-order function eachLine with a function argument which uses abort.

The exception thrown by abort might be conceivably handled at three different source locations: Either ① by the call-site of process, ② by the handler inside process, or ③ by a handler inside of eachLine. Depending on the specific example and use case, all three are valid choices the programmer could make. Now, what actually happens is that the exception will be handled by ② *unless* it happens to be handled by ③. This is impossible to know without inspecting the source code of eachLine. Moreover, to propagate the exception to ①, we would have to explicitly *forward* it from ②, without accidentally forwarding any other exceptions. This behavior is common to most languages such as JavaScript, ML, Java, Ruby, and many more.

The underlying problem is that, traditionally, exceptions are *dynamically scoped*: the exception thrown by abort unwinds the call stack and the first catch clause relative to the *dynamic* call-site of abort handles it. As explained by Zhang et al. [2016], higher-order functions such as eachLine, make it difficult for programmers to statically reason about where an exception will be handled. This behavior is not limited to exceptions but also applies to more general control operators, such as algebraic effect handlers [Plotkin and Pretnar 2009].

2.1.1 Capabilities. To facilitate reasoning about exception handlers in the presence of higher-order functions, Zhang et al. [2016] argue for a different semantics based on *lexical scoping*. Recently lexically scoped exceptions have been generalized to lexically scoped effect handlers [Biernacki et al. 2019; Brachthäuser et al. 2020a; Zhang and Myers 2019]. One particular way to obtain lexically scoped effects is to model effects as capabilities [Gordon 2020] and perform *capability passing* [Brachthäuser et al. 2020a]. Consider the previous example in a hypothetical language with lexically scoped exceptions in explicit capability-passing style:

```
function process(path, exc1) {
  function abort() { exc1.throw("processing aborted") }
  try { exc2 ⇒ eachLine(open(path, exc2), (line, exc3) ⇒ { /*...*/ abort() }) }
  catch { msg ⇒ /*... handle IO exception ...*/ }
}
```

Every exception handler introduces a term-level *capability*. Each of the three capabilities (*e.g.*, exc$_1$, exc$_2$, exc$_3$) corresponds to one of the previously marked positions where the exception thrown by abort might be handled. When we want to throw an exception, we have to use one such capability and the exception will be handled by the handler that introduced it. In the function argument of eachLine, we call abort, which in turn calls exc$_1$.throw(...). By applying local reasoning it is immediately clear that the exception will be handled at the call-site of process. Moreover, it is directly possible to throw an exception to one of the other handlers, simply by using exc$_2$ or exc$_3$. For this to be safe it is necessary (but, as we will see, not sufficient) that the capabilities are in scope.

2.2 Effect Safety

The purpose of an effect system is to statically guarantee effect safety [Nielson et al. 1999]. In the special case of exceptions this means that all exceptions are eventually caught. Enriching function types with effects enables programmers to reason about the presence and absence of particular effects of interest. However, types inferred by traditional type-and-effect systems can be verbose and difficult to understand. This is in particular the case for higher-order functions, where the types not only accurately reflect which effects the function *uses*, but also which effects it *handles*. Consider the following example in Koka [Leijen 2014], a language with a Hindley-Milner style type system, featuring a row-based effect system, and dynamically scoped effects and handlers.

```
fun rethrow(func, prog) {
  handle ({ prog() }) except throw(msg) { throw(func(msg)) }
}
```

The example defines a useful helper function which catches all exceptions in prog and rethrows them after applying func to the message msg. Koka correctly infers the most general type:

```
forall<a,e> (func: (string) → <exc|e> string, prog: () → <exc,exc|e> a) → <exc|e> a
```

It abstracts over the result type a as well as effects e. The result type tells us that function rethrow itself uses effect exc and potentially other effects e to return a result of type a. Inspecting the inferred types of the argument functions sheds some light on how type-and-effect checking in Koka (and other languages based on row polymorphism) works. There are two aspects, which we believe are difficult for programmers who are learning the language:

(1) Maybe surprisingly, argument func is assigned effect <exc|e>, but why? Since the effect system is based on row-polymorphism, the effect of func has to unify with the effects of its calling context. So the effects of func(msg), of the handler body, and of the overall function have to unify. Operationally this is correct, since func may use exceptions.
(2) Even more surprisingly, the type of argument function prog mentions two copies of exc. Again, operationally this is correct since prog might itself either throw an exception that is handled by rethrow, or one that is handled at the call-site of rethrow. Allowing duplicate entries is also necessary for soundness [Leijen 2005; Xie et al. 2020].

The function rethrow in Koka is effect polymorphic. This is important because we want to pass effectful functions to it. Consider the following helper function in Koka, which prepends the current info string to all thrown exceptions.

```
fun prependInfo(prog) { rethrow(fun(str) { getInfo() ++ str }, prog) }
```

It calls `rethrow` with a function which uses the `info` effect to express that the current info depends on the context. Koka infers the following type:

```
forall<a,e> (prog : () → <exc,exc,info|e> a) → <exc,info|e> a
```

The argument to `rethrow` uses the info effect, which leaks into the inferred type of parameter `prog` – the problem of avoiding this issue is known as *effect encapsulation* [Lindley 2018]. Many other problems of existing type- and effect systems, accidental capture [Zhang et al. 2016], or effect parametricity [Zhang and Myers 2019], can be tracked down to the operational semantics of the underlying language. The above type is correct and most general, but certainly not easy to understand. We believe, the frequent use of functions with many function parameters can render explicit polymorphism impractical. This scenario is common in OOP, where almost every method is higher-order [Cook 2009].

2.3 Ergonomics

As an alternative to parametric effect polymorphism, second-class values admit a lightweight form of effect polymorphism. Consider the same function in Effekt, a language with lexical effect handlers [Brachthäuser et al. 2020a]:

```
def rethrow[A] { func: String ⇒ String / {} } { prog: () ⇒ A / {Exc} }: A / {Exc} =
  try { prog() } with Exc { def throw(msg) ⇒ throw(func(msg)) }
```

The signature of `rethrow` is polymorphic in the result type `A`, but does not abstract over any effects. No effect variables show up in types nor error messages. Yet, it features effect polymorphism and guarantees effect safety. This is because effect signatures in Effekt are *relative* to the calling context. The parameter `func`, enclosed in curly braces, denotes a so-called *block* – a second-class function. Blocks can use all effects from the context they were defined in; accordingly, `func` does not need to explicitly mention any effects in its type – it simply can use them. This form of polymorphism is called *contextual effect polymorphism* [Brachthäuser et al. 2020a]. To illustrate, let us consider the call-site in function `prependInfo`:

```
def prependInfo[A] { prog: () ⇒ A / {Exc} }: A / {Exc, Info} =
  rethrow { str ⇒ info() ++ str } { () ⇒ prog() }
```

The signature of `prependInfo` expresses that it can handle the exception effect used by `prog` and itself may use the exception and info effects. Notice how the first argument passed to `rethrow` is effectful (it uses `info`), even though the required type is `String ⇒ String / {}`. Guided by the types, Effekt translates to System Ξ, a core calculus in explicit capability-passing style.

$$
\begin{array}{l}
\mathbf{def}\ \mathsf{prependInfo}[\mathsf{A}](\mathsf{prog} : \ \mathsf{Exc} \Rightarrow \mathsf{A},\ \mathsf{exc}_1 : \ \mathsf{Exc},\ \mathsf{info} : \ \mathsf{Info}) : \ \mathsf{A}\ = \\
\quad \mathsf{rethrow}[\mathsf{A}](\{\ \mathsf{str} \Rightarrow \mathsf{info}() +\!\!+\ \mathsf{str}\},\ \{\ \mathsf{exc}_2 \Rightarrow \mathsf{prog}(\mathsf{exc}_2)\ \},\ \mathsf{exc}_1)
\end{array}
$$

The function argument *closes over* capability info, hiding it in its closure. While this leads to concise signatures, it means we cannot require function parameters to *not* use certain effects!

2.4 First-Class Functions

Effect safety for lexically scoped effect handlers means that a capability is only used while the corresponding handler is on the stack. In other words, capabilities shall not escape their handler. For example, the following program should be ruled out, since the capability `exc` leaks via closure:

```
try { exc ⇒ return (() ⇒ exc.throw("Unsound!")) } catch { ... }
```

Type-based escape analysis [Hannan 1998] can provide this static guarantee. One particular solution is based on *second-class* values, which can be passed as arguments, but never be returned [Osvald et al. 2016]. To establish effect safety, capabilities (like `exc`) need to be second class. But, as we have seen, functions can close over capabilities, hiding their use. In consequence, existing work either *(a)* distinguishes between first-class functions that cannot close over capabilities and second-class functions that can [Osvald et al. 2016] or *(b)* treats all functions as second-class [Brachthäuser et al. 2020a]. While many useful programs can still be written with such a restriction, both solutions come with a severe loss of expressivity.

2.5 The Best of Both Worlds

To summarize, capability-passing establishes lexical scoping between the binding-site of a capability and its use. Modeling effects as capabilities has multiple advantages. Firstly, programmers can re-apply their knowledge about variable binding to reason about effects. Secondly, combining it with a type-system based on second-class values results in a lightweight form of effect polymorphism, leads to simplified signatures, and avoids problems such as effect encapsulation. However, prior work imposes severe restrictions on the use of second-class functions resulting in a significant loss of expressivity. Furthermore, second-class functions silently close over capabilities, which enables contextual effect polymorphism but also prevents type-based reasoning about purity. In the following section, we introduce System *C*, a language that lifts many of the above mentioned restrictions while preserving all the advantages of capability passing and second-class values.

3 PROGRAMMING WITH SYSTEM C

In this section, we will introduce System *C* and the underlying concepts by example.

3.1 Capabilities

One important aspect of System *C* is that it uses *capabilities* for authority control [Dennis and Van Horn 1966; Melicher et al. 2017; Miller 2006]. Operationally, a capability is an ordinary object with effectful methods. Holders of the capability are entitled to perform the corresponding effects. What makes capabilities special is that we want to keep track of their use in a program, to indirectly track the use of effects. To control access to capabilities, our system uses second-class values in the style proposed by Osvald et al. [2016]—both capabilities and functions that close over them are second class. As we will see, our system allows transitioning back-and-forth between first- and second-class values. When converting to a first-class value, the (otherwise implicitly) captured capabilities become visible in its type (and only then). When transitioning back to second class, we use this information to decide whether the transition should be allowed.

3.1.1 Global capabilities. Consider the following program written in System *C*.

```
def sayTime(): Unit { console.println("Current time is: " + time.now()) }
```

It defines a *block* `sayTime` that prints the current time to the terminal. To do so, `sayTime` uses two capabilities: `console` and `time`. As expected of second-class values, this is not mentioned in the type, which is `sayTime: () ⇒ Unit`. Here, we rely on *scope-based reasoning*—we can reference both `console` and `time`, therefore we can use them. This intuition carries over to capability-polymorphic terms. Consider `repeat`, which takes a block parameter `f` and repeats it *n* times[3].

```
def repeat(n: Int) { f: () ⇒ Unit }: Unit
{ if (n == 0) { () } else { f(); repeat(n - 1) { f }} }
```

[3] We enclose value parameters (and arguments) with parenthesis and use curly braces for block parameters (and arguments).

Unlike traditional effect systems, in which `repeat` would need to be explicitly effect-polymorphic, we rely on scope-based reasoning—`repeat` receives `f` as second-class argument, therefore it can use it. Similarly, wherever we can use a capability, we can also use it with `repeat`.

```
repeat(3) { () ⇒ console.println("Hello!") }
repeat(3) { () ⇒ sayTime() }
```

3.2 Boxes

There are situations in which scope-based thinking fails us—we sometimes want to prevent a given term from being able to use some (or all) capabilities. For instance, consider a function `parallel` that takes two blocks and runs them in parallel:

```
def parallel { f: () ⇒ Unit } { g: () ⇒ Unit }: Unit
parallel { () ⇒ console.println("Hello, ") } { () ⇒ console.println("world!") }
```

In this example, argument blocks can capture *arbitrary* capabilities. Evaluating them in parallel could perform non-deterministic side-effects or introduce data races. But how can we express a version of `parallel` that requires the function arguments to be pure? The answer in System C is: we transition to *type-based* reasoning:

```
def parallel(f: () ⇒ Unit at {}, g: () ⇒ Unit at {}): Unit
```

In this version, `parallel` now expects *first-class functions* as arguments. First-class functions are blocks whose types keep track of what set of capabilities they might reference. The functions passed to `parallel` need to be pure—they cannot reference any capabilities. Our problematic call to `parallel` now look as follows:

```
parallel( box {console} { () ⇒ console.println("Hello, ") }, // ill-typed!
          box {console} { () ⇒ console.println("world!") })  // ill-typed!
```

The type of either argument is `() ⇒ Unit at {console}`, making the above ill-typed[4]. Note how `box` marks the transition from scope-based to type-based reasoning. It takes a block and turns it into a first-class value. The boxed block can only access capabilities admitted by the boxed type. In the following, we manually annotate the box with `{}` and thus `console` cannot be accessed:

```
box {} { () ⇒ console.println("Hello, ") } // ill-typed!
```

To complete the picture, consider what capability sets would be inferred in the following term:

```
box {?} { () ⇒ sayTime() }
```

Intuitively, we should allow sets no smaller than `{console,time}`, since `sayTime` itself uses those capabilities. But how can System C infer this information and refuse programs like the ill-typed example above? The answer is that this information is kept at the binders itself. Which is to say, our system annotates the following blocks with capability sets:

```
def {console, time} sayTime() : Unit
def {} repeat(n: Int) { f: () ⇒ Unit }: Unit
def {console, time} sayTimeThrice(): Unit { repeat(3) { () ⇒ sayTime() } }
```

[4] We use the notation {...} to display capability sets, which are inferred by the type checker and displayed by the IDE.

3.2.1 Local Capabilities. So far we have only discussed global capabilities, which prevented us from highlighting one important aspect of our approach to capabilities. In System C, neither capabilities nor blocks can be returned. Why do we want such a restriction? Consider the following term:

```
withFile("a.txt") { file ⇒ file.readByte(0) }
```

Function `withFile` creates a capability to access a file, and passes it to a block. After the block terminates, `withFile` closes the handle and returns the result of the block. If we let the handle outlive the block, using it afterwards results in an error—this is precisely what we want to prevent. We could follow Osvald et al. [2016] and Brachthäuser et al. [2020a] and forbid to return *any* capabilities or functions that close over them. However, this is overly restrictive since sometimes we might want to return a capability from some scope, other than its own.

Example 3.1. Consider that we may want to do the following: open file A.txt, open file B.txt, read B's contents to define a block that then continues to read from A, return the block from the scope of file B so that we can use it. Naturally, our block will need to use the handle to A, so how can we return it? We *box* the block into first-class value, at which point we can see (based on its type) that returning it is safe. The above scenario can be modeled in System C as follows:

```
withFile("A.txt") { fileA ⇒
  val offsetReader : Int ⇒ Byte at {fileA} =
    withFile("B.txt") { fileB ⇒
      val offset = fileB.readByte(0);
      return box {fileA} { pos ⇒ fileA.readByte(pos + offset) }
    };
  (unbox offsetReader)(10)
}
```

Note how in order to use `offsetReader`, we first need to *unbox* it. In System C, first-class functions cannot be used at all—they first need to be unboxed, which turns them back into second-class blocks[5]. We only allow unboxing when all the capabilities mentioned in the box's type are in scope. The reason for why this is sound becomes apparent if we consider the previous sentence—since unboxing turns boxes back into second-class values, we can only unbox blocks in environments that anyway have access to no less than what the block has access to!

3.2.2 From Scope-based Reasoning to Type-based Reasoning and Back. Our notion of scope-based reasoning comes from the idea of second-class values [Osvald et al. 2016]. The familiar concept of lexical scoping enables convenient and flexible reasoning about the use of effects [Brachthäuser et al. 2020a; Zhang and Myers 2019]. As already pointed out, not being able to return second-class values at all is an overly harsh restriction. Other than the example we have already seen, it immediately rules out the common technique of *currying* functions with second-class arguments.

Our notion of type-based reasoning is inspired by an approach to reasoning about effects with capabilities introduced by Choudhury and Krishnaswami [2020]. They demonstrate how to recover a notion of pure functions in a language that does not otherwise keep track of effects. The idea is to have a special type of values that are guaranteed to not have access to *any* capabilities. We take this idea and generalize it to keep track of *which* capabilities a value has access to. A function of type `S ⇒ T at {}` is known to be pure, but we are not limited to using the empty set in function types. An example is the value `box sayTime`, which has an inferred capability set of $\{\mathit{console}, \mathit{time}\}$. That is, we not only know that it is impure, but also which capabilities it closes over.

[5] In our implementation of System C, we infer almost all necessary boxing and unboxing operations. However, in the paper, for exposition we refrain from doing so.

System *C* harmoniously combines these two ways of reasoning about effects via capabilities and allows programmers to move between them. We mediate between blocks and functions by explicitly converting them with box and unbox, respectively. As long as blocks are used in a strictly second-class manner, by design, closing over capabilities is not visible to the programmer. However, as soon as a function is used as a first-class value, the capabilities come to light.

3.2.3 Capability Polymorphism. Effect systems based on capabilities give rise to a new notion of *contextual* effect polymorphism [Brachthäuser et al. 2020a], as observed in the repeat example. Blocks passed to repeat can simply use all capabilities in their lexical scope. Since System *C* supports boxing blocks, this (so far invisible) polymorphism now can manifest itself in types:

```
def repeater { f: () ⇒ Unit }: Int ⇒ Unit at { f }
{ return box { n ⇒ repeat(n) { f } } }
```

The return type of repeater uses a limited form of term dependent types to express *capability polymorphism*: intuitively, the returned function closes over any capabilities that f closes over. This becomes visible when calling repeater with sayTime, which closes over console and time:

```
val repeatTime : Int ⇒ Unit at { console, time } = repeater { sayTime }
```

By design, block arguments, such as f are always capability polymorphic. In contrast, block definitions, such as sayTime are always *capability monomorphic*. Only capabilities and polymorphic block variables are allowed to occur in capability sets.

3.3 Effect Handlers in System *C*

System *C* combines the notion of second-class values with a particularly general and challenging language feature (already present in System Ξ): effect handlers [Plotkin and Pretnar 2009, 2013]. One potentially uncommon aspect of our effect handlers is that we use lexical effect handling in capability-passing style [Biernacki et al. 2019; Brachthäuser et al. 2020a]. We briefly introduce effect handlers and refer the interested reader to other introductions [Pretnar 2015]—the work by Zhang et al. [2020] and Brachthäuser et al. [2020a] is particularly similar in syntax and semantics to our approach. Potentially the simplest and most familiar application of effect handlers are exceptions.

```
try { console.println("hello"); exc.throw("world"); console.println("done") }
with exc: Exc { def throw(msg: String) { console.println(msg + "!") } }
```

After printing the string "hello", by invoking exc.throw, control flow is transferred to the handler, which simply prints the string "world!". The final call to println is unreachable. Handlers introduce capabilities, such as exc, which here has type Exc. The attentive reader will notice a potential problem—if capabilities are terms, what happens if we perform exc.throw outside of the enclosing try? The answer is: exc is a block and cannot leave the enclosing scope. As such, exc.throw can only be performed when it is handled. Trying to return it will yield a type error:

```
try { return (box {exc} exc) } with exc: Exc { ... } // type error
```

The type of the boxed capability is Exc at {exc}, which is not well-formed outside of the corresponding handler that binds it. Unlike exceptions, effects handlers in our system are not limited to aborting the computation—they can continue it at the original call to the capability.

```
val before = time.now();
try { console.println(watch.elapsed()) } with watch: Stopwatch {
  def elapsed() { resume(time.now() - before) }
}
```

Again, the handler introduces a capability of type `Stopwatch`. However, this time the handler implementation *resumes* the computation by passing a value of type `Int`, the *return type* of the effect operation. Interestingly, the continuation `resume` closes over both the capabilities used by the handled program, as well as the capabilities used by the handler itself. In this case, we have `box` $\{\text{console}, \text{time}\}$ `resume` since the handled program uses `console` and the handler uses `time`.

3.4 Conclusion

System C combines two approaches to effects via capabilities: scoped-based reasoning (which admits lightweight polymorphism) and type-based reasoning (which enables reasoning about absence). We can move between the two styles with `box` and `unbox`.

4 FORMAL PRESENTATION

In this section, we formally present the syntax, static and dynamic semantics of System C, and highlight meta-theoretic properties. The presentation follows the one of Brachthäuser et al. [2020a]. For clarity, and to focus on the novel aspects of System C, we omit type polymorphism from our presentation of System C, which is largely orthogonal to the rest of our calculus (Section 5.1). We highlight some important aspects of the calculus, which we will discuss later in full detail.

Computation and values. Since the calculus supports control effects via effect handlers, it is presented in fine-grain call-by-value [Levy et al. 2003]. We syntactically distinguish statements, which may perform effectful computation (that is, they are *serious* in the terminology of Reynolds [1972]), from expressions and blocks, which are pure (that is, *trivial*) and cannot perform effects.

Values and blocks. Following Brachthäuser et al. [2020a], we separate the universe of values into *expression values* that are considered first-class [Osvald et al. 2016] and *block values*, which we consider second-class. To emphasize the first-class nature of expression values, we often speak of *values* and *blocks*. Importantly, blocks may implicitly close over capabilities, whereas values are explicit and reveal captured capabilities in their type. Syntactically, we distinguish between variables that stand for expression values (x, y, ...) and variables that stand for block values (f, g, ...). The stratification can also be observed on the level of types, where we introduce value types τ and block types σ, correspondingly.

Boxing and unboxing. Blocks can be lifted into values by boxing—reifying contextual information in the type; (function) values can be lowered into blocks by explicit unboxing—making capture information contextually available.

4.1 Syntax

Figure 1 defines the syntax of System C. We have syntactic categories for expressions, blocks, and statements. Only statements can perform effectful computation. As usual, we follow Barendregt [1992] and require that all variable names are globally unique.

4.1.1 Expressions.

Expressions are either variables, primitives, or boxed blocks. The evaluation of expressions never has side effects. We could add, for example, integer addition to the syntactic category of expressions. Boxing a block (*i.e.*, **box** b) performs no side effects either, and only reifies the information about its captured capabilities from the typing context into the type of the resulting boxed block. The ability to box blocks presents a significant extension to other calculi with first- and second-class values [Brachthäuser et al. 2020a; Osvald et al. 2016], because it allows a second-class block b to be lifted to become a first-class value v.

Syntax:				
Expressions	e	$::=$	x	expression variables
		$\mid$	$() \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \ldots$	primitives
		$\mid$	$\mathbf{box}\ b$	box introduction
Blocks	b	$::=$	f	block variables
		$\mid$	$\{\ (\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s\ \}$	block implementation
		$\mid$	$\mathbf{unbox}\ e$	box elimination
Statements	s	$::=$	$\mathbf{def}\ f = b;\ s$	block definition
		$\mid$	$b(\overrightarrow{e_i}, \overrightarrow{b_j})$	block application
		$\mid$	$\mathbf{val}\ x = s;\ s$	sequencing
		$\mid$	$\mathbf{return}\ e$	returning
		$\mid$	$\mathbf{try}\ \{\ f \Rightarrow s\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x_i}, k) \Rightarrow s\ \}$	handlers
Types:				
Value Types	τ	$::=$	$\mathsf{Int} \mid \mathsf{Boolean} \mid \ldots$	base types
		$\mid$	$\sigma\ \mathbf{at}\ C$	boxed block types
Block Types	σ	$::=$	$(\overrightarrow{\tau_i}, \overrightarrow{f_j : \sigma_j}) \to \tau$	
Capabilities	C	$::=$	$\emptyset \mid \{f\} \mid C \cup C$	
Environments:				
Environments	Γ	$::=$	$\emptyset$	empty environment
		$\mid$	$\Gamma,\ x : \tau$	value bindings
		$\mid$	$\Gamma,\ f :^{*} \sigma$	tracked bindings
		$\mid$	$\Gamma,\ f :^{C} \sigma$	transparent bindings

Fig. 1. Syntax of the language System C – differences to System Ξ highlighted in *grey*.

4.1.2 Blocks. Blocks in System C play the role of functions in other languages. In contrast to traditional functions in other lambda calculi, our blocks are multi-arity to avoid the complexity of currying in effectful languages. Blocks come in two forms: block literals and unboxed values. Block literals are of the form $\{\ (\overrightarrow{x_i : \tau_i}, \overrightarrow{f_j : \sigma_j}) \Rightarrow s\ \}$. They simultaneously abstract over multiple value parameters $x_i : \tau_i$ as well as multiple block parameters $f_j : \sigma_j$. The body of a block literal is a (potentially effectful) statement. Unboxing an expression with ($\mathbf{unbox}\ e$) re-embeds the first-class (function) value e into the universe of blocks. Boxing and unboxing are inverse operations of each other and we have that $\mathbf{box}\ (\mathbf{unbox}\ e) \equiv e$ as well as $\mathbf{unbox}\ (\mathbf{box}\ b) \equiv b$.

4.1.3 Statements. Finally, statements represent potentially effectful computation in System C. Block definitions $\mathbf{def}\ f = b;\ s$ and statement sequencing operations $\mathbf{val}\ x = s_1;\ s_2$ evaluate blocks and statements to block and expression values respectively and bind them to names before evaluating the remaining portion of the program. Multi-arity block application takes multiple expressions as well as multiple blocks. Note that only blocks can be applied, and in particular, boxed blocks must first be *unboxed* before they can be called.

Effect handlers. System C supports effect handlers in capability-passing style [Brachthäuser et al. 2020a]. A handling statement of the form $\mathbf{try}\ \{\ f \Rightarrow s_1\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x_i}, k) \Rightarrow s_2\ \}$ introduces a fresh capability f in the scope of the handled program s_1. When the capability is invoked, control is passed to the handler s_2 with arguments bound to $\overrightarrow{x_i}$ and the continuation bound to k. Calling the

Surface Language	*Core Calculus*	
def f(x: T_1) { f : T_2 ⇒ T_3 } = ...	**def** f = { (x : T_1, f : $T_2 \to T_3$) ⇒ ... }	Block abstr.
f { x ⇒ ... }	f({ x ⇒ ... })	Block app.
{ T_1 ⇒ T_2 } ⇒ T_3	(f : $T_1 \to T_2$) → T_3	Block types
try { ... } f : S with { ... }	**try** { f ⇒ ... } **with** { (x, resume) ⇒ ... }	Resumptions
f(g())	**val** x = g(); f(x)	Fine-grain CBV
s_1; s_2	**val** x = s_1; s_2	Sequencing

Table 1. Mapping between the informal surface syntax and formal presentation of System C.

continuation transfers control back to original call-site of the capability. Note that only expression values can be passed to the capability, which is important for effect safety, as otherwise a capability introduced in the body of the handled program may leave its defining scope.

4.1.4 Types. System C differentiates between value types τ and block types σ, just like how it distinguishes expression values and block values; we assign value types to expression values, and block types to blocks. Analogously to term-level boxing, a block type σ can be annotated (or "boxed") with a capability set C to form a value type (that is, σ **at** C). Grammatically, *capability sets* C are sets of block variable names f – however, well-formed types and terms can only mentioned a subset of bindings, which we explain in Section 4.1.5. Block types take multiple value types τ_i and multiple block types $f_j : \sigma_j$ to a single value type τ. In particular, the return type τ can mention any of the bound f_j within a capability set. Block types add a limited form of term dependency to System C. One example is a capability-polymorphic identity function: { (f : σ) ⇒ **return box** f }. Here, the term-level boxing is reflected in the return type of (f : σ) → σ **at** {f}, which mentions f.

4.1.5 Environments. Contexts Γ can bind first-class values $x : \tau$ and second-class blocks. Based on different annotations on the binder, we distinguish between two different kinds of block bindings. Firstly, a binding of the form $f :^{*} \sigma$ is *tracked*. That is, the use of the block f will be tracked by the type system. We also refer to these tracked block variables as *capabilities*. Only tracked bindings can be mentioned in capability sets (Appendix A.1). Secondly, a binding of the form $f :^{C} \sigma$ is *transparent*. In order to use block f, all capabilities C are required to be in scope. We refer to those bindings as transparent, since using f by itself will not be tracked in the type system. Furthermore, they will never occur in capability sets and consequently do not occur in types.

4.2 Surface Syntax

There are differences between our formal calculus and the surface language we have used in our motivating examples. To facilitate mapping between the two languages, Table 1 relates the syntax and summarizes a few syntactical abbreviations. In System C, block definitions have separate lists of block and value parameters separated by a comma. Our informal syntax distinguishes between value parameters and block parameters, by enclosing value parameters in parenthesis and block parameters in braces. In the examples, we also use additional features such as type polymorphism, algebraic data types, or mutable variables. Those extensions will be discussed in Section 5.

4.3 Typing

The static semantics of System C is defined in terms of three typing judgements for expressions, blocks, and statements (Figure 2). We present the (meta-level) syntax of the judgements itself in grey. We start with block typing as it features the most relevant ideas in System C.

Block Typing. $\boxed{\Gamma \vdash b : \sigma \mid C}$

$$\frac{f :^{C} \sigma \in \Gamma}{\Gamma \vdash f : \sigma \mid C}\,[\textsc{Transparent}] \qquad \frac{f :^{*} \sigma \in \Gamma}{\Gamma \vdash f : \sigma \mid \{f\}}\,[\textsc{Tracked}]$$

$$\frac{\Gamma,\ \overrightarrow{x_i : \tau_i},\ \overrightarrow{g_j :^{*} \sigma_j} \vdash s : \tau \mid C \cup \overrightarrow{g_j}}{\Gamma \vdash \{\ (\overrightarrow{x_i : \tau_i},\ \overrightarrow{g_j : \sigma_j}) \Rightarrow s\ \} : (\overrightarrow{\tau_i},\ \overrightarrow{g_j : \sigma_j}) \to \tau \mid C}\,[\textsc{Block}]$$

$$\frac{\Gamma \vdash e : \sigma\ \mathbf{at}\ C}{\Gamma \vdash \mathbf{unbox}\ e : \sigma \mid C}\,[\textsc{BoxElim}] \qquad \frac{\Gamma \vdash b : \sigma \mid C' \qquad C' \subseteq C}{\Gamma \vdash b : \sigma \mid C}\,[\textsc{BSub}]$$

Expression Typing. $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}\,[\textsc{Lit}] \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,[\textsc{Var}] \qquad \frac{\Gamma \vdash b : \sigma \mid C}{\Gamma \vdash \mathbf{box}\ b : \sigma\ \mathbf{at}\ C}\,[\textsc{BoxIntro}]$$

Statement Typing. $\boxed{\Gamma \vdash s : \tau \mid C}$

$$\frac{\Gamma \vdash s_0 : \tau_0 \mid C_0 \qquad \Gamma,\ x : \tau_0 \vdash s_1 : \tau_1 \mid C_1}{\Gamma \vdash \mathbf{val}\ x = s_0;\ s_1 : \tau_1 \mid C_0 \cup C_1}\,[\textsc{Val}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{return}\ e : \tau \mid \emptyset}\,[\textsc{Ret}]$$

$$\frac{\Gamma \vdash b : (\overrightarrow{\tau_i},\ \overrightarrow{f_j : \sigma_j}) \to \tau \mid C \qquad \overrightarrow{\Gamma \vdash e_i : \tau_i} \qquad \overrightarrow{\Gamma \vdash b_j : \sigma_j \mid C_j}}{\Gamma \vdash b(\overrightarrow{e_i},\ \overrightarrow{b_j}) : \tau[\overrightarrow{f_j \mapsto C_j}] \mid C \cup \overrightarrow{C_j}}\,[\textsc{App}]$$

$$\frac{\Gamma \vdash b : \sigma \mid C' \qquad \Gamma,\ f :^{C'} \sigma \vdash s : \tau \mid C}{\Gamma \vdash \mathbf{def}\ f = b;\ s : \tau \mid C}\,[\textsc{Def}] \qquad \frac{\Gamma \vdash s : \tau \mid C' \qquad C' \subseteq C}{\Gamma \vdash s : \tau \mid C}\,[\textsc{SSub}]$$

$$\frac{\begin{array}{c}\Gamma,\ f :^{*} \overrightarrow{\tau_i} \to \tau_0 \vdash s_1 : \tau \mid C \cup \{f\} \\ \Gamma,\ \overrightarrow{x_i : \tau_i},\ k :^{C} \tau_0 \to \tau \vdash s_2 : \tau \mid C\end{array}}{\Gamma \vdash \mathbf{try}\ \{\ f \Rightarrow s_1\ \}\ \mathbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s_2\ \} : \tau \mid C}\,[\textsc{Try}]$$

Fig. 2. Static semantics of System C.

4.3.1 Block Typing. Typing judgements for blocks and statements have the form $\Gamma \vdash b : \sigma \mid C$. In these judgements, C is a subset of Γ and tracks the *effect of mentioning* capabilities. We can read it in two ways: first, as an input, which describes a *context restriction*; only those capabilities mentioned in C will be available. second, as an output, which describes a *context requirement*; typing b requires all tracked capabilities in C to be in scope. As usual, we require that all components b, σ, and C are well-formed with respect to the typing context Γ. Typing rules Transparent and Tracked check block variables and express the requirements on the context. Referencing tracked variables requires the variable itself to be in the context. For transparent bindings, we require that the annotated capability set C. This is important, as this constraint enforces the restriction that blocks may only be invoked, and hence effectful computation are only performed, in a context where the corresponding capabilities are in scope. A boxed block value can be unboxed through rule BoxElim only when the annotated capability set is compatible with the requirements in the current context C. Again, this ensures that effectful computations can only be performed in a context where its capabilities are in scope. Finally, rule Block types block literals. As usual, the body of the block literal s is checked in a context extended with the bindings for values x_j and blocks g_j where the latter are marked as *tracked capabilities*. As we will see in rule App, this is to support *capability polymorphism*. In contrast, all blocks bound by **def** are *capability monomorphic*.

4.3.2 Expression Typing. The judgement form $\Gamma \vdash e : \tau$ assigns a value type τ to an expression e, in a typing environment Γ. Expression typing is completely independent of any requirements on the context. This highlights a central aspect of System C: expressions are *first class* and can be freely used without any limitations. This is safe, as capabilities that are implicitly captured by an expression can only be used by *unboxing* the expression, which checks if the capabilities mentioned on the boxed type are present in the lexical context. Most rules are completely standard; the only interesting rule is the rule for boxing blocks – BoxIntro, which reifies the requirement C under which we check the enclosed block b into the type σ **at** C, making it visible to the programmer.

4.3.3 Statement Typing. Typing rules Val and Ret are completely standard. Val simply collects the requirements for the binding and the body. Rule Ret types and expression and thus does not have any requirements. Similarly to block typing, statement typing includes a rule SSub to shrink the current requirement to a subset. Let us now explain the other three rules in detail.

Typing applications. Rule App, is used to check an application $b(\overrightarrow{e_i},\ \overrightarrow{b_j})$. The callee b has to be checked against a block type. The value arguments e_i need to conform to value types τ_i. Typing each block argument b_j, however, can result in different requirements C_j. The resulting type of checking the application is $\tau[\overrightarrow{f_j \mapsto C_j}]$. That is, occurrences of block variable names f_j in the return type τ are substituted with the concrete requirement the arguments could be type checked in. Where Block serves the dual purpose of abstracting over terms (expressions and blocks) and (implicitly) over capability sets, rule App now applies the block b to terms as well as (implicitly) to capability sets.

Typing block definitions. Rule Def checks the bound block b under an arbitrary restriction C' and annotates the binder with this restriction to type check the rest of the program s. Block definitions are *transparent*, that is, f itself will not show up in any capability set. Notably, the restriction C' is independent of C and thus does not necessarily need to be a subset of C. In this regard, rule Def is very similar to rule BoxIntro as it delays the requirements C' to the use site of f.

Typing effect handlers. Rule Try checks statements of form **try** $\{\ f \Rightarrow s_1\ \}$ **with** $\{\ (\overrightarrow{x_i},\ k) \Rightarrow s_2\ \}$ in a context Γ under a context requirement C. We first discuss typing of the body s_1 and typing of the handler s_2. Handling brings a fresh capability f into the scope of the handled program s_1. The capability has block type $\overrightarrow{\tau_i} \to \tau_0$, which we also refer to as the *effect signature*. That is, given a list of value arguments it returns a value of type τ_0. We refer to τ_i as the types of the *arguments* of the effect operation, to τ_0 to the *return type* of the effect operation, and to τ as the *answer type* of the handler. Like in the rule Block, the capability binding for f is marked as being tracked. However, unlike rule App we *do not* substitute for f in the answer type τ. This is essential to guarantee effect safety. By marking f as tracked, it cannot leave the scope of the corresponding handler that introduced it. In particular, if a first-class function closes over f, then f will necessarily appear in its type. That is the following example program does not type check, since **box** f has type σ **at** {f}:

try { (f : σ) ⇒ **return box** {f} f } **with** { ... }

The return type σ **at** $\{\mathsf{f}\}$ is not well-formed outside of the handler, since the block variable f is not bound. In consequence, effect safety indirectly follows from *(1)* tracking capabilities and *(2)* well-formedness of types. Finally, the handler implementation s_2 itself is type checked in a context extended with the parameters of the effect operation $x_i : \tau_i$ and the continuation k. The continuation expects a value of type τ_0 as an argument (the return type of the effect operation), and will itself return τ (the answer type of the handled statement). Most importantly, the continuation is marked as transparent and annotated with the capability set C. As witnessed by the operational semantics, the continuation closes over both the handled statement as well as the handler statement and thus is annotated with C, the restriction which both statements are type checked under.

4.4 Operational Semantics

We define the semantics of System C as a small-step operational semantics using evaluation contexts [Wright and Felleisen 1994]. To allow capturing and resuming continuations, the semantics of System C closely follows the generative semantics presented by Brachthäuser et al. [2020a], who in turn present a variant of multi-prompt delimited control [Biernacki et al. 2019; Gunter et al. 1995]. Figure 3 extends the syntax with runtime constructs that only appear during reduction:

Labels. All runtime constructs refer to unique runtime labels l. We only require that labels can be compared for equality and that we are able to generate fresh labels at runtime. We represent concrete labels as hexadecimal hashes (*e.g.*, `@a5f`) to highlight that they are created at runtime.

Delimiters. The additional statement $\#_l$ $\{\ s\ \}$ **with** $\{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$ represents a *delimiter* that delimits a statement s at a given label l (or *prompt* in the terminology of Felleisen [1988], Sitaram [1993], and Gunter et al. [1995]). It also contains the original handler implementation $\{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$, which we sometimes abbreviate with the meta variable h.

Capabilities. The additional block value $\mathbf{cap}_l$ represents a *capability*. Calling a capability captures the stack segment up to the next dynamically enclosing delimiter for the label l, and transfers control to the corresponding handler. While we could also attach the handler implementation to the capability and pass it, alongside the label, to the call-site [Brachthäuser and Schuster 2017; Zhang and Myers 2019], we choose to locate the handler with the delimiter, because it simplifies proofs.

Label sets. Capability sets C are extended with an additional production (*e.g.*, $\{l\}$), which effectively turns them into heterogeneous sets of block variables and labels. Source programs start with variable sets only—reduction then replaces free block variables with runtime labels.

Label contexts. The global label context Ξ behaves like a store and maps runtime labels to effect signatures $\overrightarrow{\tau_i} \longrightarrow \tau_0$. The label context is merely a proof device necessary to prove type preservation.

4.4.1 Reduction Rules. The presentation of the operational semantics in Figure 3b follows Gunter et al. [1995] and is based on *delimited evaluation contexts* H_l where the label l does not appear in any delimiters in H_l. This is necessary to establish that captured continuations are always delimited by the dynamically closest delimiter for a label. Most reduction rules are standard and we only point out significant differences to previous presentations. The full operational semantics, as well as definition of expression values v and block values w can be found in Appendix A.2. We overload the notation for substitution in the following way: we use $f \mapsto w$ to refer to a substitution of block variable f by block value w in terms. Additionally, we use $f \mapsto C$ to refer to a substitution of block variable f by capability set C in both terms (that is, in type annotations) and types. This substitution replaces all occurrences of f in capability sets by C. The result of substitution is then flattened.

Extended Syntax for Operational Semantics:

Runtime Labels	l	$::=$	`@a5f` $\mid$ `@4ba` $\mid \ldots$	labels
Runtime Statements	s	$::=$	$\ldots \mid \#_l \{ s \}$ **with** $\{ (\overrightarrow{x_i},\ k) \Rightarrow s \}$	delimiters
Runtime Blocks	b	$::=$	$\ldots \mid$ **cap**$_l$	capabilities
Runtime Capabilities	C	$::=$	$\ldots \mid \{l\}$	label sets
Runtime Signatures	Ξ	$::=$	$\emptyset \mid \Xi,\ l : \overrightarrow{\tau_i} \to \tau_0$	label context

(a) Extended Syntax of System C.

Evaluation Contexts:

Contexts	E	$::=$	$\square \mid$ **val** $x = \mathsf{E};\ s$	$\mid \#_l \{ \mathsf{E} \}$ **with** $\{ (\overrightarrow{x_i},\ k) \Rightarrow s \}$	
Delimited Contexts	H_l	$::=$	$\square \mid$ **val** $x = \mathsf{H}_l;\ s$	$\mid \#_{l'} \{ \mathsf{H}_l \}$ **with** $\{ (\overrightarrow{x_i},\ k) \Rightarrow s \}$	where $l \neq l'$

Reduction Rules:

(box)	**unbox** (**box** b)	$\longrightarrow$	b
(val)	**val** $x =$ **return** $v;\ s$	$\longrightarrow$	$s[x \mapsto v]$
(def)	**def** $f = w;\ s$	$\longrightarrow$	$s[f \mapsto w]$
(ret)	$\#_l \{$ **return** $v \}$ **with** h	$\longrightarrow$	v
(app)	$(\{ (\overrightarrow{x_i},\ \overrightarrow{f_j}) \Rightarrow s \})(\overrightarrow{v_i},\ \overrightarrow{w_j})$ where $\emptyset \vdash w_j : \sigma_j \mid C_j$	$\longrightarrow$	$s[\overrightarrow{x_i \mapsto v_i},\ \overrightarrow{f_j \mapsto C_j},\ \overrightarrow{f_j \mapsto w_j}]$
(try)	**try** $\{ f \Rightarrow s \}$ **with** $\{ (\overrightarrow{x_i},\ k) \Rightarrow s' \}$ where $l \notin dom\, \Xi$, and $\vdash f : \overrightarrow{\tau_i} \to \tau_0$, then $\Xi(l) := \overrightarrow{\tau_i} \to \tau_0$	$\longrightarrow$	$\#_l \{ s[f \mapsto \{l\},\ f \mapsto$ **cap**$_l] \}$ **with** $\{ (\overrightarrow{x_i},\ k) \Rightarrow s' \}$
(cap)	$\#_l \{ \mathsf{H}_l[$ **cap**$_l(\overrightarrow{v_i})\ ] \}$ **with** h where $h = \{ (\overrightarrow{x_i},\ k) \Rightarrow s \}$	$\longrightarrow$	$s[\overrightarrow{x_i \mapsto v_i},\ k \mapsto \{ y \Rightarrow \#_l \{ \mathsf{H}_l[$**return** $y] \}$ **with** $h \}]$

(b) Operational semantics of System C – we omit congruences.

Fig. 3. Additional runtime constructs and operational semantics of the language System C. The global context Ξ maps labels to effect signatures at runtime – it is extended by rule *(try)*.

Reducing values and blocks. The only reduction of expressions and blocks is **box**/**unbox** elimination as defined in rule *(box)*[6]. To keep the presentation concise, we omit congruences.

Value and block binders. The reduction of value *(val)* and block binders *(def)* is completely standard. Since blocks bound by **def** are capability-set monomorphic, reducing block binders only performs term-level substitution $f \mapsto w$ and does not need to substitute a capability set for f.

Application substitutes capability sets. In contrast, in reduction rule *(app)*, we simultaneously substitute f_j with a block value w_j in terms and a capability set C_j in types. Like in typing rule App, C_j denotes the context requirement the argument block w_j was checked in. Interestingly, all redexes (including application) can be type checked in the empty typing context Γ. This implies that the substituted capability sets C_j do not contain any block variables, but only runtime labels.

[6] One could imagine that blocks are stack-allocated, while boxed blocks are heap-allocated. Unboxing then could copy the closure back to the stack. We leave working out the details of this observation to future work.

Handling introduces delimiters. Rule *(try)* creates a fresh runtime label l, delimits the handled statement s with this label, and substitutes a capability that refers to l for the block variable f. Similarly, on the type-level, we substitute the singleton label set $\{l\}$ for block variable f. As a side-effect, we record the effect signature of f in the global label context Ξ. As already pointed out, the global context is only necessary to prove type preservation—when handling an effect operation, we need to establish that the type of the capability and the type at the handler still agree.

Capabilities capture the continuation. The most interesting rule *(cap)* captures part of the context H_l. The application of a capability with label l is only meaningful in a context, which is delimited at label l. This becomes visible in *(cap)*, where the delimiter $\#_l$, the delimited context H_l, and the capability application form a redex. We reify this context as a continuation and substitute it (as well as the argument v) in the body of the handler implementation. Effect safety means that applications of a capability with label l only occur in a context that contains a delimiter at l (Theorem 4.2).

Only boxed values can leave delimiters. Once a statement is reduced to a value, rule *(ret)* discards the delimiter. This is the very point where effect safety could be violated. So why is this reduction safe? As already pointed out in the discussion of typing rule Try, since only values can be returned, blocks that could potentially close over labels will have to be boxed. Boxing in turn reifies captured capabilities, and therefore labels, into the type of an expression. Wellformedness of type of v guarantees that no reference of l can occur in the type and thus v cannot close over l.

Example 4.1. In fact, a returned value *can* close over a label—but only if the corresponding capability is never used. Take the following example reduction:

$$\textbf{try}\ \{\ \mathsf{f} \Rightarrow \textbf{return box}\ \{\}\ \{\ () \Rightarrow \textbf{val}\ \mathsf{g}\ =\ \textbf{box}\ \mathsf{f};\ \textbf{return}\ 42\ \}\ \}\ \textbf{with}\ \{\ \dots\ \}$$

The example type checks and the returned value has type $() \longrightarrow \mathsf{Int}\ \textbf{at}\ \{\}$. By rule *(try)*, we obtain

$$\#_{\mathtt{@a31}}\ \{\ \textbf{return box}\ \{\ () \Rightarrow \textbf{val}\ \mathsf{g}\ =\ \textbf{box cap}_{\mathtt{@a31}};\ \textbf{return}\ 42\ \}\ \}\ \textbf{with}\ \{\ \dots\ \}$$

which then (by rule *(ret)*) steps to:

$$\textbf{box}\ \{\ () \Rightarrow \textbf{val}\ \mathsf{g}\ =\ \textbf{box cap}_{\mathtt{@a31}};\ \textbf{return}\ 42\ \}$$

Notably, the returned value *does* contain a reference to label `@a31`. However, the value itself cannot be unboxed, as we show in Section 4.5, disarming the capability it contains.

4.5 Safety

We state progress and preservation and point out important aspects of our proof.

4.5.1 Progress. Given an arbitrary label context Ξ, closed and well-typed System C programs either return a value or can take a step. Here, the relation $\longmapsto$ (defined in Appendix A.2, Figure 5b) describes congruence, that is, statement reduction under an evaluation context E.

Theorem 4.2 (Progress of System C). *If $\emptyset \vdash s : \tau \mid \emptyset$, then s is **return** v or $s \longmapsto s'$.*

The proof of progress is mostly straightforward. Only the case of capability application requires special precautions. In particular, we state the following auxiliary lemma.

Lemma 4.3 (Labels are delimited). *If $\emptyset \vdash E[\textbf{cap}_l(\overrightarrow{v_i})] : \tau \mid \emptyset$ and $\Gamma \vdash \textbf{cap}_l(\overrightarrow{v_i}) : \tau' \mid C$ and $\vdash_{\mathsf{ctx}} E : \tau' \rightsquigarrow \tau \mid C$, then $E = E'[\ \#_l\ \{\ H_l\ \}\ \textbf{with}\ h\ ]$.*

This lemma uses the judgement $\vdash_{\mathsf{ctx}} \mathsf{E} : \tau_1 \rightsquigarrow \tau_2 \mid C$ to type contexts, which is defined in Appendix A.3. Here, τ_1 is the type at the hole and τ_2 is the return type of the resulting statement. Importantly, capability set C can be thought of as an output of the relation. It represents the restriction under which the hole can be type checked, that is, all labels delimited by the context.

Proof of Progress. The proof of progress simply amounts to splitting s into an evaluation context and a redex, such that $s = \mathsf{E}[s_{red}]$. If s_{red} is a redex, other than $\mathbf{cap}_l(\overrightarrow{v_i})$, we can simply invoke rule *cong*, otherwise we use Lemma 4.3 followed by rule *(cap)*. □

4.5.2 Preservation. Performing a reduction step on a statement preserves its type:

Theorem 4.4 (Preservation of System C). *If $\emptyset \vdash s : \tau \mid \emptyset$ and $s \longmapsto s'$ then $\emptyset \vdash s' : \tau \mid \emptyset$.*

Proving preservation requires proving of substitution lemmas. In particular, it requires a variant taking simultaneous substitution of blocks and capability sets into account:

Lemma 4.5 (Substitution of blocks and capability sets). *Given a well-typed statement $\Gamma_1,\ f :^{*} \sigma,\ \Gamma_2 \vdash s : \tau \mid C_1$ and a block $E \vdash b : \sigma \mid C_2$ that can be checked under restriction $E \vdash C_2$ wf, then $\Gamma_1,\ \Gamma_2[f \mapsto C_2] \vdash s[f \mapsto b,\ f \mapsto C_2] : \tau[f \mapsto C_2] \mid C_1[f \mapsto C_2]$.*

The corresponding lemmas for expressions and blocks are similar.

Proof. The proof proceeds by mutual induction over the typing derivation. Due to context restrictions and capability sets, proving substitution requires reasoning about subset inclusion, but is straightforward otherwise. Notably, by construction all entries $l : \overrightarrow{\tau_i} \rightarrow \tau_0$ in the signature environment Ξ are typable in the empty context Γ and thus substitution is idempotent on them. □

Furthermore, we need to make sure that capturing the continuation preserves types.

Lemma 4.6. *Given a well-typed effect call $\emptyset \vdash H_l[\ \mathbf{cap}_l(\overrightarrow{v_i})\ ] : \tau \mid C \cup \{l\}$ with effect signature $l : \overrightarrow{\tau_i} \rightarrow \tau_0 \in \Xi$, it follows that $y : \tau_0 \vdash H_l[\ \mathbf{return}\ y\ ] : \tau \mid C \cup \{l\}$.*

Proof of Preservation. By induction over the typing derivation, followed by inversion on the step taken. Steps *(app)* and *(try)* both require the lemma for simultaneous substitution (Lemma 4.5). The only other interesting case is the application of rule *(cap)*. Here, we need to construct a typing derivation for $s[\overrightarrow{x_i \mapsto v_i},\ k \mapsto \{\ y \Rightarrow \#_l\ \{\ \mathsf{H}_l[\mathbf{return}\ y]\ \}\ \mathbf{with}\ h\ \}]$. Assuming the label typing $l : \overrightarrow{\tau_i} \rightarrow \tau_0 \in \Xi$, in order to apply the substitution lemma on the continuation k, we need to show that $\emptyset \vdash \{\ y \Rightarrow \#_l\ \{\ \mathsf{H}_l[\mathbf{return}\ y]\ \}\ \mathbf{with}\ h\ \} : \tau_0 \rightarrow \tau \mid C$. After applying rules Block and Reset, we finally use Lemma 4.6 to conclude the proof. □

4.5.3 Effect Safety. We characterize effect safety as ruling out a particular class of stuck terms: capability applications without a corresponding delimiter [Brachthäuser et al. 2020a].

Definition 4.7 (Undelimited Label). A statement s contains an undelimited label l, if it has the form $\mathsf{H}_l[(\mathbf{cap}_l(\overrightarrow{v_i})]$.

Corollary 4.8. *Starting from an empty context reducing a well-typed program $\emptyset \vdash s : \tau \mid \emptyset$ never results in an undelimited label.*

This corollary directly follows from progress and preservation. It further relates to Lemma 4.3, which guarantees that labels are always delimited.

4.5.4 Mechanization. This paper is accompanied by a mechanization of System C in the Coq theorem prover [Bertot and Castéran 2004], as well as proofs of the usual progress and preservation theorems. To facilitate mechanization, we chose to diverge from the presentation in the paper:

Representing names. We base our mechanization efforts on the proofs by Aydemir et al. [2008], who in turn use a locally nameless representation to distinguish free from bound variables. Consequently, we represent capability sets as triples of free variables (opaque atoms), bound variables (natural numbers), and labels. The universes of labels and atoms are disjoint.

Explicit annotations. Instead of assuming capability sets from the context, as is done in our presentation of System C, our mechanized formulation requires that some terms are explicitly annotated with that capability set. This includes application (*i.e.*, $b(\overrightarrow{e_i},\ \overrightarrow{b_j \mathbin{@} C_j})$), block definition (*i.e.*, $\textbf{def}\ f \mathbin{@} C\ =\ b;\ s$), and handlers (*i.e.*, $\textbf{try} \mathbin{@} C\ \{\ f \Rightarrow s_1\ \}\ \textbf{with}\ h$). This way, in the mechanization we never have to infer the capability sets and restrictions.

Abstract machine semantics. We model the semantics of statements with control effects in terms of a state machine (Appendix A.4). This way, to search for the correct delimiter unwinding the stack takes place frame by frame. For each unwinding step, we can easily establish the invariant that all free labels in the captured continuation are delimited in the remaining stack, and that the effect call can be type checked in the composition of the continuation and the stack.

Label safety. To ensure that an effect handler associated with a label is invoked with the right arguments, we extend the state of the abstract machine with a field for runtime effect signatures Ξ, acting as a source for fresh labels [Dybvig et al. 2007] and to guarantee that the handler itself is invoked with arguments of the proper type [Brachthäuser et al. 2020a].

Type polymorphism. Finally, to ensure that System C can be used as a basis for modeling effect safety for practical languages, we formalized support for value-type polymorphism in our mechanization of System C, as described in Section 5.1. As value types τ are orthogonal to the effect system in System C, our proof terms for dealing with value-type polymorphism are mainly straightforward extensions of the proof terms one would obtain in a mechanization of System F. In particular, one can never unbox a term which is typed with a value-type variable—BoxElim expects a expression typed with a concrete boxed block type.

5 DISCUSSION OF LANGUAGE EXTENSIONS

We have implemented the static and dynamic semantics as a compiler from System C to JavaScript. We submit the implementation, all the code examples in this paper, as well as additional small case studies as supplementary material. In this section, we further evaluate the design of System C and the involved concepts by discussing several implemented language extensions.

5.1 Parametric Type Polymorphism

$$\frac{\Gamma, X \vdash b : \sigma \mid C}{\Gamma \vdash [X] \Rightarrow b : \forall X.\, \sigma \mid C}\ [\textsc{TAbs}] \qquad \frac{\Gamma \vdash b : \forall X.\, \sigma \mid C}{\Gamma \vdash b[\tau] : \sigma[X \mapsto \tau] \mid C}\ [\textsc{TApp}]$$

Type polymorphism is largely orthogonal to tracking capture in capability sets. To support type polymorphism, we extended the syntax of blocks with support for type abstraction and type application, together with the above two standard typing rules. Importantly, type variables X range over value types, not block types. That is, values of type X cannot silently close over capabilities. A function like **def** f[X](x : X) cannot perform unboxing on x since it is parametric in its type X. We extended System C with type polymorphism in our implementation and proved its soundness in our mechanization. Proving the extension did not impose any interesting challenges.

5.2 Mutable State

In languages with support for control effects, the implementation of local mutable variables requires some care in order to obtain the correct backtracking behavior [Brachthäuser et al. 2018; Kiselyov et al. 2006]. Effect handlers are general enough to express mutable state, but rely on first-class functions to do so. Where System Ξ was unable to express local state as an effect handler, System C with its support for first-class functions now makes it possible.

$$\frac{\Gamma, f :^{*} \mathrm{Reg} \vdash s : \tau \mid C \cup \{f\}}{\Gamma \vdash \textbf{region}\ \{ f \Rightarrow s \} : \tau \mid C}\ [\textsc{Region}] \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash b : \mathrm{Reg} \mid C}{\Gamma \vdash \textbf{new}\ b(e) : \mathrm{Ref}[\tau] \mid C}\ [\textsc{New}]$$

$$\frac{\Gamma \vdash b : \mathrm{Ref}[\tau] \mid C}{\Gamma \vdash\ !b : \tau \mid C}\ [\textsc{Get}] \qquad \frac{\Gamma \vdash b : \mathrm{Ref}[\tau] \mid C \qquad \Gamma \vdash e : \tau \mid C}{\Gamma \vdash b := e : \mathrm{Unit} \mid C}\ [\textsc{Put}]$$

Fig. 4. Extension with local backtrackable state.

```
def handleState[S, R](init: S) { prog: {State[S]} ⇒ R }: R {
  val stateFun: S ⇒ R at {prog} =
    try { val res = prog { state }; return box {prog} { (s: S) ⇒ res } }
    with state: State[S] {
      def get() { box {prog} { (s: S) ⇒ (unbox resume(s))(s) } }
      def set(v: S) { box {prog} { (_: S) ⇒ (unbox resume(()))(v) } }
    };
  return (unbox stateFun)(init)
}
```

This example in System C type checks and exhibits the correct behavior. The boxed block uses the capabilities that `prog` uses. While it is possible to emulate local mutable state with effect handlers, for efficiency and flexibility it is worthwhile to investigate a direct implementation.

Scoped State. To support state, Figure 4 extends System C with two new block types: Reg to describe dynamic regions and Ref[τ] to represent reference cells of type τ. We also extend the language with three new statement forms. **region** $\{ f \Rightarrow s \}$ delimits a fresh region and introduces a region capability Reg that can be used to create fresh references. References of type Ref[τ] can be accessed (*i.e.*, $!b$) and written to (*i.e.*, $b := e$) using the new statement forms. Finally, **new** $b(e)$ is a block, which given a region initializes a fresh reference and returns a capability to access that reference. The example on the left presents a simple example using the state extension[7].

```
region r {
  var x in r = 42;
  val t = x;
  x = (t + 1)
}
```

```
region r {
  var x in r = 42;
  val f: () ⇒ Int at {r} = box {r} { () ⇒ x };
  (unbox f)()
}
```

We create a region, allocate a reference initialized to 42, and increment its content. The example on the right illustrates that access to mutable references becomes visible in boxed blocks. The box is typechecked under {r}, since dereferencing x requires r to be in scope. In our implementation, every block definitions and effect handlers implicitly creates a new region; for example, function definitions `def myFun() { ... }` automatically introduces a fresh (equally named) region `def myFun() { region myFun { ... } }`. When allocating a variable, omitting the region will default to the closest lexical region. This allows us to express the above example as `region r { var x: Int = 42; val t = x; x = t + 1 }`. At the same time, however, we still guarantee capability safety. It is interesting to see how references that are used in a second-class

[7] The surface language differs slightly from the calculus and we write `x` for $!x$, `x = e` for $x := e$, `region r { ... }` for **region** $\{ r \Rightarrow \ldots \}$, and `var x in r = e; s` for **def** x = **new** $r(e);\ s$.

way, and therefore naturally follow a stack discipline, do not need any special precautions to prevent them from escaping. It is only when we want to return a reference, or a closure that uses it, that the region becomes visible in the type. We believe that even for languages without effect handlers this design for region-based resource management would be worth investigating.

Example 5.1. As a more advanced example of mutable state and effects, we demonstrate one of the original motivations of supporting first-class functions: Being able to express cooperative multitasking using effect handlers [Ahman and Pretnar 2021; Dolan et al. 2017; Leijen 2017a].

```
interface Proc { def fork(): Boolean }
def schedule { p: { Proc } ⇒ Unit }: Unit {
  var q: Queue[() ⇒ Unit at {p, schedule}] = emptyQueue();
  try { p {proc} } with proc: Proc {
    def fork() { q = enqueue(q, box {p, schedule} { () ⇒ resume(true) });
                 q = enqueue(q, box {p, schedule} { () ⇒ resume(false) }) }
  };
  while (nonEmpty(q)) { val (q2, k) = dequeue(q); q = q2; (unbox k)() }
}
```

The above handler implementation assumes the presence of a Queue datatype along standard operations such as enqueue, dequeue, and nonEmpty. When fork is invoked, it pushes two continuations to the queue, once resuming with true and once resuming with false. In order to be able to store a second-class continuation in a Queue, we need to explicitly box it. Boxing reifies the capability set of the continuation into the type, which is () ⇒ Unit at {p, schedule}. The handled program closes over p and the handler itself uses state allocated in the region named schedule, hence the whole **try** statement can only be typechecked under a restriction allowing for both capabilities. As discussed in Section 4.3, the continuation is also annotated with this restriction.

5.3 Type- and Capability Inference

While we leave a full formal treatment of inference to future work, here we want to report on our experiences in implementing System C. Reading the context restriction C of a typing judgement as an *output*, the type system of System C can be thought of as tracking the effect of *referencing* a block variable. This can be seen in typing rule Tracked, which “introduces” the variable f into the restriction. However, the typing rules presented in Section 4.3 are not fully algorithmic. There are four rules that require some adjustments to facilitate type and capability inference.

Subsumption. As usual, subsumption rules BSub and SSub present difficulties for type inference. Since System C only supports subtyping on capability sets (as subset inclusion) but not on types, in our implementation, we simply defer all applications of subsumption to one rule: BoxIntro. If a box is annotated with an *expected* capability set **box** $C\ b$ and the block b can be checked under C', we then assert that $C' \subseteq C$. In all other cases, we either compute capability set requirements via set union (like rule Val) or collect equality constraints (as in Try below).

Abstraction. We rephrase rule for block abstraction Block as follows:

$$\frac{\Gamma,\ \overrightarrow{x_i : \tau_i},\ \overrightarrow{g_j :^{*} \sigma_j} \vdash s : \tau \mid C}{\Gamma \vdash \{\, (\overrightarrow{x_i : \tau_i},\ \overrightarrow{g_j : \sigma_j}) \Rightarrow s \,\} : (\overrightarrow{\tau_i},\ \overrightarrow{g_j : \sigma_j}) \to \tau \mid C - \overrightarrow{g_j}}\ [\textsc{Block}]$$

Inspecting the conclusion, we see that block abstraction conceptually *handles* (that is, removes) bound block parameters $\overrightarrow{g_j}$, while application introduces the corresponding capability sets C_j by means of set union in the conclusion.

Capability Set on the Continuation. Maybe the most challenging rule is Try, which *masks*, that is *removes*, a tracked variable. We can rephrase it as:

$$\frac{\begin{array}{c}\Gamma,\ f :^{*}\ \overrightarrow{\tau_i} \to \tau_0 \ \vdash\ s_1 \ :\ \tau \mid C_1 \qquad C \ =\ (C_1 \setminus \{\, f \,\}) \cup C_2 \\ \Gamma,\ \overrightarrow{x_i :\ \tau_i},\ k :^{C}\ \tau_0 \to \tau \ \vdash\ s_2 \ :\ \tau \mid C_2\end{array}}{\Gamma \ \vdash\ \textbf{try}\ \{\, f \Rightarrow s_1 \,\}\ \textbf{with}\ \{\, (\overrightarrow{x_i},\ k) \Rightarrow s_2 \,\} \ :\ \tau \mid C}\ [\textsc{TryEff}]$$

While the rule above is more algorithmic, the astute reader might have noticed that on the premise checking s_2, the "output" requirement C_2 also indirectly appears as part of the annotation on binder of the continuation k. This cyclic definition makes it difficult to derive a fully algorithmic variant that assigns C_2 to the minimal capability set. In our implementation, we annotate the continuation binder k with a fresh unification variable for the capability set. After checking the handler s_2, we might have gathered cyclic constraints that would require fixed point computation to be solved. Our implementation can infer the correct solution for all examples and case studies submitted, we leave solving the constraints in the general case to future work. We want to note that this potential complication only arises since we offer support for first-class continuations. Languages that only support exception handlers and regions would not encounter this difficulty.

5.4 Effect Handlers and Object-Oriented Programming

In the introduction, we used capabilities like `console` assuming they have multiple member *methods* (*e.g.*, `println`, and `readln`). This is not reflected in the description of our core calculus, which only formalizes blocks, but no objects or methods. The following example uses an extension with interfaces and objects:

```
interface Counter {
  def inc(): Unit
  def get(): Int
}
```

```
def makeCounter { pool: Region }: Counter at {pool,console} {
  var count in pool = 0;
  def {console, pool} c = new Counter {
    def inc() { console.println(count); count = count + 1 }
    def get() { count }
  };
  return box {console, pool} c
}
```

Perhaps unintuitively we treat objects as a generalization of blocks—that is, they are second-class by default! This implies that objects, such as `c`, can simply close over arbitrary capabilities. In this case, `c` closes over `console` and the region `pool`, which (as with blocks) is not visible in its type. Only if and when we want to return `c` do we box it, making its capabilities explicit in its type. As it has been pointed out earlier [Brachthäuser et al. 2020b; Zhang et al. 2020], it is very natural to unify the notion of effect signatures and interfaces, capabilities and objects, as well as handlers and classes. The only difference is that objects created with `new` do not have a continuation to capture.

6 RELATED WORK

The calculus presented in this paper builds on different lines of work, centered around capabilities. In this section, we offer a discussion of this work, before comparing System C to other approaches based on effects and coeffects.

6.1 Capability-based Systems

Osvald et al. [2016] present a calculus $\lambda^{1/2}$, implementing a type-based escape analysis [Hannan 1998] that distinguishes between first-class and second-class values. They demonstrate that second-class values provide a lightweight alternative to effect systems, which can even express borrowing [Osvald and Rompf 2017]. Our work is closely connected to $\lambda^{1/2}$: tracked blocks in System C correspond to second-class values, while expression values correspond to first-class values. They propose a generalization to arbitrary lattices. Similarly, in System C, we precisely track the capability sets of transparent block bindings, which form a lattice. In their calculus only pure functions can be treated first-class, while in System C arbitrary blocks can be boxed.

Brachthäuser et al. [2020a] build on $\lambda^{1/2}$, develop it into a full language with support for effect handlers, and explore the novel and lightweight form of effect polymorphism offered by treating effects as capabilities. Their core calculus System Ξ is the basis for System C. It divides the universe of types into value and block types, and distinguishes between expressions, blocks, and statements. However, their calculus does not offer explicit boxing and unboxing, neither on the term level, nor on the type level. In consequence, blocks can never be returned or stored as values. This greatly simplifies the type system of System Ξ. System C is designed to be backwards compatible with System Ξ: Programs only using blocks as second-class need no changes or additional annotations.

6.2 Representing Closure in Types

Boxing in System C makes the requirements a block imposes on the calling context explicit in its type. There have been various lines of work to enrich the types of closures with information about its context. Hannan [1998] proposes a type-based escape analysis with the goal of facilitating stack allocation. The analysis tracks variable reference using a type-and-effect system and annotates every function type with the set of free variables it captures. The authors leave the treatment of effect polymorphism to future work. In a similar spirit, Scherer and Hoffmann [2013] present *Open Closure Types* to facilitate reasoning about data flow properties such as non-interference. They present an extension of the simply typed lambda calculus that enhances function types with the lexical environment used to type the closure. Odersky et al. [2021] propose to gradually establish exception safety in Scala with capabilities. To ensure capability safety, they track captured capabilities by enhancing types with capture sets. Similarly to System C, their calculus supports a closely related concept of boxing, as well as lightweight dependent types. Unlike their calculus, System C distinguishes between first-class values (where capture is tracked in types) and second-class values (where capture is tracked on binders), improving the ergonomics. System C also tracks mention of capabilities with an effect-like system, making it more precise. For example, a statement $g :^{*} \sigma \vdash \mathbf{def}\ f = g;\ \mathbf{return}\ () : \mathsf{Unit} \mid \emptyset$, has the empty restriction. Finally, while their system features full subtyping, System C only features subeffecting on blocks and statements.

6.3 Comonadic Type Systems

Comonadic type systems, as presented by Choudhury and Krishnaswami [2020], allow programmers to reason about *purity* in an impure language. A special type constructor `Safe` witnesses the fact that its values are constructed without using any (impure) capabilities. Importantly, explicit box introductions and box eliminations mark the transition between scope-based reasoning, and type-based reasoning about effects (that is, *impurity*). The concept of boxing and unboxing in System C is inspired by their work. They annotate each entry in the typing context with additional information about whether it is pure (or *safe*, *e.g.*, $x : A^{s}$) or impure (*e.g.*, $x : A^{i}$), similar to our annotations on block binders. Their notion of purity is related to our notion of expression values: a pure value is constructed only by accessing other pure values. Similarly, an expression value in System C can only (immediately) consist of other expression values. There are two important differences, though.

Generalizing the notion of impurity. They only distinguish pure from impure values. Since it is their goal to create isolated islands of purity in an otherwise impure language, making this distinction suffices. They already point out that,

> [A] direction for future work lies in the observation that our $\Box$-comonad [. . .] takes away all capabilities [. . .]. However, we could consider a graded or indexed version of the same [. . .], i.e., $\Box_C$, which only takes away a set of capabilities C [. . .].

In this paper, we do almost exactly this. However, in System C the boxed type σ **at** C does not witness which capabilities C are "taken away", but instead, which capabilities might have been used to construct this boxed value. This generalization is significant for our use case of establishing effect safety. Effect handlers locally introduce capabilities, that we want to subtract (or *mask*), because their effects are delimited and cannot be observed outside of the handler. This would not be possible in a system that only distinguishes between pure and impure computation.

Context purification. Another interesting difference is our notion of restricting the typing context. The context of typing judgements for statements $\Gamma \vdash s : \tau \mid C$ consists of two parts: the typing context Γ and the restriction C. Together, they enable restricting the use of block variables, as witnessed by rules Tracked and Transparent. In System C this context restriction does not necessarily have to become smaller as we nest boxes. This is illustrated in the following example on the left, which does type check. Here, we write **box** C b to refer to a type ascription **box** $b : \sigma$ **at** C for some block type σ.

```
{ (f : () ⇒ Int) ⇒
  return box {} { () ⇒
    return box {f} { () ⇒ f() }
  }
}
```

$$\begin{array}{llll}
(\Gamma,\ f :^{*}\ \sigma)^C & = \Gamma^C,\ f :^{*}\ \sigma & \textbf{if} & f \in C \\
(\Gamma,\ f :^{C'}\ \sigma)^C & = \Gamma^C,\ f :^{C'}\ \sigma & \textbf{if} & C' \subseteq C \\
(\Gamma,\ f :^{C'}\ \sigma)^C & = \Gamma^C & \textbf{otherwise} & \\
(\Gamma,\ x : \tau)^C & = \Gamma^C,\ x : \tau & &
\end{array}$$

On the left, the nested box imposes the restriction $\{f\}$, while the outer box imposes a stronger restriction $\{\}$. This is different in the work of Choudhury and Krishnaswami [2020] and also in the work of Osvald et al. [2016]. Both restrict contexts by filtering the typing context, written Γ^C. In our setting, this restriction could be implemented as sketched on the right. That is, only those bindings which are compatible with C remain in Γ^C[8]. This eager filtering of the context is a significant difference which would make the language less expressive.

6.4 Contextual Modal Types

Effectful Contextual Modal Type Theory [Zyuzin and Nanevski 2021] aims to relate algebraic effects and contextual modal logic [Nanevski et al. 2008]. Like System C, it syntactically distinguishes pure expressions from effectful computation. The judgement for typing computation (*i.e.*, $\Delta; \Gamma \vdash c \div T$) takes two contexts, Δ to bind expressions and Γ to bind effect operations. Computations can be embedded into expressions by boxing, which delays the computation. The type of boxes is indexed by an *algebraic effect theory* Ψ, reifying the context of effect operations Γ at box creation into the type. While conceptually related, there are a few important differences. Lambda abstractions in ECMTT can only abstract over expressions, not effect operations. They also only close over the value context Δ. In contrast, blocks in System C can both take expressions and capabilities as

[8] Furthermore, to maintain well-formedness also all bindings which refer to other filtered block variables need to be removed.

arguments and also close over both. In ECMTT the only way to force a boxed computation is by immediately handling it. Unhandled effects need to be forwarded explicitly and ECMTT does not support effect polymorphism. Importantly, ECMTT is much closer to classical effect systems, while System C implements lexical effects and models capabilities on the term level, including closure.

6.5 Coeffect-based Systems

Dually to how effect systems annotate the output of the typing judgement with additional information, coeffect systems enrich the *input* of a typing judgement, that is, the typing context [Petricek et al. 2014]. Petricek et al. differentiate between two forms of coeffects: structural and flat. Structural coeffects annotate each bound variable, while flat coeffects annotate the context as a whole. Our annotations on block variable bindings roughly correspond to structural coeffects, whereas the context restriction C roughly corresponds to a flat coeffect. While coeffects served as a source of inspiration for System C, the precise connection is unclear. It would be interesting to to see whether we could instantiate Petricek et al.'s framework. It is not immediately clear to us how to combine structural and flat coeffects to at the same time annotate individual bindings and restrict the context as a whole. Furthermore, System C supports some limited form of term dependency in types to support capability polymorphism. It would be interesting to see how the coeffect framework could be extended to support coeffect polymorphism in this way. Our type of boxed blocks σ **at** C is reminiscent of the graded box modality of Gaboardi et al. [2016], but we do not know how to instantiate their system to accommodate our use case. Our use of box seems to be closer to the box introduction of Nanevski et al. [2008], but again the precise connection is not clear to us.

6.6 Effect-based Systems with Capabilities

Zhang and Myers [2019] present a language with effect handlers, where effect operations are used by invoking methods on capabilities. In a similar vein, Biernacki et al. [2019] present a language with effect handlers where they track the use of effect instances in the type of functions. Their type- and effect systems guarantee effect safety by tracking the use of capabilities. However, they establish effect safety by means of traditional effect systems and do not have a notion of second-class values. Instead they directly support parametric effect polymorphism. Every function, even if it is only used in a second-class way, carries effect information whereas in our language System C this information only becomes visible when functions are made first-class.

Crary et al. [1999] present a language with a type system that statically tracks capabilities. Their motivation is to make region-based memory management safe. The underlying problem they solve is similar to ours: we want to make sure that capabilities are only used when they are still valid. They have a separate notion of *regions*, while in System C tracked variables serve the dual purpose of regions and handles, depending on whether they appear as terms or in types. When viewed like this, their system becomes similar to ours. Again, the key benefit of System C is that no type-level region information is needed for variables that follow a stack discipline.

6.7 Effect Systems and Type-Based Reasoning

In Sections 1.1 and 2.2 we compared with various related work on effect systems that feature type-based reasoning. Those systems [Biernacki et al. 2019; Leijen 2017b; Lindley et al. 2017; Zhang and Myers 2019] typically support effect polymorphism in terms of abstraction and application of effect variables. In contrast, System C features two *modes of operation*, first- and second-class, mediated by boxing and unboxing. In second-class mode, we fully avoid the ceremony of parametric effect polymorphism. However, System C does not improve the verbosity of type-based reasoning with first-class values, which is comparable to existing solutions. Importantly, transitioning to type-based reasoning is performed *selectively* and details are only exposed when necessary.

6.8 Call-by-Push-Value

Levy [1999] presents call-by-push-value (CBPV) as a paradigm that subsumes call-by-value and call-by-name. CBPV distinguishes between values (denoted A) and computations (denoted $\underline{B}$) on the type-level. Value types can be embedded into computation types with a type constructor $F\ A$ and computation types can be embedded into value types with a type constructor $U\ \underline{B}$.

Similarities. By presenting System C in fine-grain call-by-value, we also distinguish pure expressions from side-effecting computations. We furthermore separate computation into the separate syntactic categories of statements and blocks. Figure 7 in Appendix A.5 should serve as a guide for readers familiar with CBPV. On the type level all System C statements can be thought of as typed against a computation type $F \lceil \tau \rceil$. Ignoring our addition of capabilities C, the typing rules for returning and sequencing computation align with those of CBPV. The typing judgements of boxing and unboxing, which correspond to thunking and forcing in CBPV, also align.

Differences. Despite some similarity, there are important differences between System C and CBPV. The mapping of System C to CBVP is not complete. In particular, CBPV does not support abstraction over computation and its functions can only be applied to values, not computations. That is, there are no equivalents of block variables, block abstraction, and block application in CBPV. Furthermore, there are CBPV programs which are not expressible in System C. For example, CBPV functions have the general shape of $A \rightarrow \underline{B}$ and can result in further computation, which is not possible in System C. We accommodate the typical use case of this feature by generalizing to multi-arity functions. In CBPV term abstraction by itself *does not delay computation*, but merely requires an argument to be on top of the stack. For example, in CBPV, the following two programs are semantically equivalent:

$$M\ \mathbf{to}\ x\ \mathbf{in}\ (\lambda y\ N) \quad \equiv \quad \lambda y\ (M\ \mathbf{to}\ x\ \mathbf{in}\ N)\ \} \qquad \textit{where } y\ \notin \textit{fv}(M)$$

Both programs are computations of function type $A \rightarrow \underline{B}$ and expect a value $(y :\ A)$ to be on the top of the stack when executed. In System C a program corresponding to the first one would not be syntactically correct, as blocks (computations of function type) cannot be returned without boxing. Since in System C, blocks already delay computation, we repurpose the thunking / boxing construct to delay the effect of *"mentioning a tracked variable"*. We leave a full formal comparison of System C and CBPV to future work.

7 CONCLUSION

In this paper, we presented System C, in which natural scope-based reasoning and precise type-based reasoning can co-exist and programmers can switch between the two. Capabilities and blocks let us assign simple types to common definitions, while boxed blocks allow us to circumvent typical limitations of second-class values and let us be precise in signatures where necessary. System C integrates well with languages with advanced control flow, as witnessed by our implementation of effect handlers. Our system is sound as well, as evidenced by the proof mechanized in Coq. We studied System C as an alternative effect system for capability-based lexical effects. However, the design might also be interesting for languages with simple control effects (like exceptions) or region-based resource management. In the future, to remove the burden of explicitly passing capabilities we would like to investigate effect inference.

ACKNOWLEDGMENTS

The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

REFERENCES

Danel Ahman and Matija Pretnar. 2021. Asynchronous Effects. *Proc. ACM Program. Lang.* 5, POPL, Article 24 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434305

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the Symposium on Principles of Programming Languages* (San Francisco, California, USA). ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

Henk P. Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures*. Oxford University Press, New York, NY, USA, 117–309.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development, Coq'Art:The Calculus of Inductive Constructions*. Springer-Verlag.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. *Proc. ACM Program. Lang.* 4, POPL, Article 48 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371116

Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *Proceedings of the International Symposium on Scala* (Vancouver, BC, Canada). ACM, New York, NY, USA. https://doi.org/10.1145/3136000.3136007

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect Handlers for the Masses. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 111 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276481

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020). https://doi.org/10.1145/3428194

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. Effekt: Capability-Passing Style for Type- and Effect-safe, Extensible Effect Handlers in Scala. *Journal of Functional Programming* (2020). https://doi.org/10.1017/S0956796820000027

Jonathan Immanuel Brachthäuser and Daan Leijen. 2019. *Programming with Implicit Values, Functions, and Control.* Technical Report MSR-TR-2019-7. Microsoft Research.

Vikraman Choudhury and Neel Krishnaswami. 2020. Recovering Purity with Comonads and Capabilities. *Proc. ACM Program. Lang.* 4, ICFP, Article 111 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408993

Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. 2016. Exploring language support for immutability. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 736–747.

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications* (Orlando, Florida, USA). ACM, New York, NY, USA, 557–572.

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. https://doi.org/10.1145/292540.292564

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Effectively Tackling the Awkward Squad. In *ML Workshop*.

R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17, 6 (2007), 687–730.

Matthias Felleisen. 1988. The Theory and Practice of First-class Prompts. In *Proceedings of the Symposium on Principles of Programming Languages* (San Diego, California, USA). ACM, New York, NY, USA, 180–190.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/2951913.2951939

Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:25. https://doi.org/10.4230/LIPIcs.ECOOP.2020.10

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA). ACM, New York, NY, USA, 12–23.

John Hannan. 1998. A Type-based Escape Analysis for Functional Languages. *Journal of Functional Programming* 8, 3 (May 1998), 239–273.

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *Journal of Functional Programming* 30 (2020), e5. https://doi.org/10.1017/S0956796820000040

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *Proceedings of the International Conference on Functional Programming* (Portland, Oregon, USA). ACM, New York, NY, USA, 26–37.

Daan Leijen. 2005. Extensible records with scoped labels. In *Proceedings of the Symposium on Trends in Functional Programming*. 297–312.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*.

Daan Leijen. 2017a. Structured Asynchrony with Algebraic Effects. In *Proceedings of the Workshop on Type-Driven Development* (Oxford, UK). ACM, New York, NY, USA, 16–29.

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499.

Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–243.

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210.

Sam Lindley. 2018. Encapsulating effects. *Dagstuhl Reports* 8, 4 (2018).

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the Symposium on Principles of Programming Languages* (Paris, France). ACM, New York, NY, USA, 500–514.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A capability-based module system for authority control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph. D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA. Advisor(s) Shapiro, Jonathan S. AAI3245526.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. Type and effect systems. In *Principles of Program Analysis*. Springer, 283–363.

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

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, New York, NY, USA, 234–251.

Leo Osvald and Tiark Rompf. 2017. Rust-like Borrowing with 2nd-Class Values (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 13–17. https://doi.org/10.1145/3136000.3136010

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the International Conference on Functional Programming* (Gothenburg, Sweden). ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

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

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference* (Boston, Massachusetts, USA). ACM, New York, NY, USA, 717–740.

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *Proceedings of the European Conference on Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–282.

Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 710–726.

Dorai Sitaram. 1993. Handling Control. In *Proceedings of the Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA). ACM, New York, NY, USA, 147–155.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. *Proc. ACM Program. Lang.* 4, ICFP, Article 99 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408981

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages.

Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting Blame for Safe Tunneled Exceptions. In *Proceedings of the Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA). ACM, New York, NY, USA, 281–295.

Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling Bidirectional Control Flow. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 139 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428207

Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual Modal Types for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 5, ICFP, Article 75 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473580

A APPENDIX

A.1 Well-formedness of Capability Sets

Capability sets can only contain tracked block variables, which is captured by the following well-formedness rules.

$$\frac{}{\Gamma \vdash \emptyset\ \mathit{wf}} \qquad\qquad \frac{f :^{*} \sigma \in \Gamma}{\Gamma \vdash \{f\}\ \mathit{wf}} \qquad\qquad \frac{\Gamma \vdash C_1\ \mathit{wf} \qquad \Gamma \vdash C_2\ \mathit{wf}}{\Gamma \vdash C_1 \cup C_2\ \mathit{wf}}$$

A.2 Full Operational Semantics

Here, we fill in missing details of our description of the operational semantics. For easier reference, Figure 5a repeats the extended syntax.

A.2.1 Definition of Values, Evaluation Contexts, and Statement Congruence. Figure 5b repeats the reduction rules and defines evaluation contexts E and H_l as well as the congruence rule for statements $\sigma \longmapsto \sigma'$. Expression values v include primitive constants and boxed block values w. The latter is a significant difference to the presentation by Brachthäuser et al. [2020a], where blocks were second class and could never be returned or stored in data structures. In contrast, in System C, blocks can be boxed and then treated as first-class expression values. Block values w are either block literals or capabilities.

A.2.2 Additional Typing Rules. We also extend the static typing rules to cover the new additional runtime constructs introduced by the operational semantics and prove progress and preservation for the extended calculus.

Figure 5c extends the typing rules for System C with two additional rules. Rule Cap is very similar to rule Tracked and checks capabilities simply by looking up their signature in the global signature context Ξ. Similarly to rule Tracked, we require that f is compatible with the current context restriction. Rule Reset is very similar to rule Try. Since the capability has been substituted by rule *(try)*, we only have to establish that the handled statement type checks under restriction $C \cup \{l\}$. The premise for checking the handler is exactly the one of Try. Finally, we should explicitly note that the static typing rules for boxing and unboxing—BoxIntro and BoxElim—now additionally reifiy runtime labels into the type system and check to ensure that any runtime labels are present in the current environment restriction when unboxing values to ensure that the value itself may be safely unboxed.

A.3 Proof that Labels are Delimited

The following rules define the auxiliary typing of evaluation contexts $\vdash_{\mathrm{ctx}} \mathsf{E} : \tau_1 \rightsquigarrow \tau_2 \mid C$ as used in Lemma 4.3.

$$\frac{}{\vdash_{\mathrm{ctx}} \square : \tau \rightsquigarrow \tau \mid \emptyset}\ [\textsc{CTop}] \qquad \frac{\vdash_{\mathrm{ctx}} \mathsf{E} : \tau_2 \rightsquigarrow \tau \mid C \qquad x : \tau_1 \vdash s : \tau_2 \mid C}{\vdash_{\mathrm{ctx}} \mathsf{E}[\mathbf{val}\ x = \square;\ s] : \tau_1 \rightsquigarrow \tau \mid C}\ [\textsc{CVal}]$$

$$\frac{\begin{array}{l} l : \overrightarrow{\tau_i} \rightarrow \tau_0 \in \Xi \qquad \vdash_{\mathrm{ctx}} \mathsf{E} : \tau_1 \rightsquigarrow \tau_2 \mid C \qquad C' \subseteq C \\ \overrightarrow{x_i : \tau_i},\ k :^{C'} \tau_0 \rightarrow \tau_1 \vdash s : \tau_1 \mid C' \end{array}}{\vdash_{\mathrm{ctx}} \mathsf{E}[\ \#_l\ \{\ \square\}\ \mathbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \})] : \tau_1 \rightsquigarrow \tau_2 \mid C' \cup \{l\}}\ [\textsc{CReset}]$$

Here, τ_1 is the type at the hole and τ_2 is the overall return type of the resulting statement. Importantly, the capability set C can be thought of as an output of the relation. It represents the restriction under which the hole can be type checked, that is, it contains all labels that are delimited by the context.

Proof of Lemma 4.3. Similar to how an abstract machine would unwind the stack frame-by-frame, the proof proceeds by induction over the typing derivation of the context. The cases for **val**-frames, and delimiters are straightforward. Only the case for the empty context is interesting. From $\Gamma \vdash \textbf{cap}_l(\overrightarrow{v_i}) : \tau' \mid C$, we know that $l \in C$. However, context typing of the empty context only admits an empty restriction (that is, $C = \emptyset$), which leads to a contradiction. □

A.4 Abstract Machine Semantics

Our mechanization in Coq deviates from the presentation in Section 4.4. Instead of building the operational semantics on evaluation contexts, we model the semantics in terms of an substitution-based abstract machine.

Figure 6 presents this version of the operational semantics in terms of an abstract machine, very similar to the presentation by Hillerström et al. [2020]. Machine states are triples of the form $\langle\ s \mid \mathsf{E} \mid \Xi\ \rangle$, where E are stacks. We reuse the definition of evaluation contexts in Figure 3b but, for pushing stack frames, write $\textbf{val}\ x = \square;\ s :: \mathsf{E}$ instead of $\mathsf{E}[\textbf{val}\ x = \square;\ s]$ to highlight the nature of the stack. The machine can be in one of two states:

(1) Reduction Mode. States of the form $\langle\ s \mid \mathsf{E} \mid \Xi\ \rangle$ are used to perform standard machine reductions. Similar to the presentation by Dybvig et al. [2007], the last component of the machine state Ξ is used as a source for fresh labels. However, like in our presentation of the semantics in Section 4.4, it also maps labels to effect signatures at runtime, necessary to establish type preservation. Standard reductions include reductions that can be performed without affecting the machine state (rule *(cong)*). Again, we omit the trivial congruence rules for blocks and expressions. Rules *(pop)*, *(ret)*, *(push)*, and *(reset)* perform the standard reductions of pushing and popping frames of the stack.

(2) Search Mode. In contrast, states of the form $\langle\ s \mid \mathsf{E} \circ \mathsf{E}' \mid \Xi\ \rangle$ are used to model the search for a delimiter. Here E' is the captured continuation and E is the remainder of the stack that the search continues on. The notation $\mathsf{E} \circ \mathsf{E}'$ suggests that we can compose the two stacks $\mathsf{E}[\mathsf{E}']$ to obtain the original stack before the search for a delimiter started. Rule *(cap)* marks the transition from standard machine reductions to the search for a delimiter for l. Rules *(unwind)* and *(forward)* stepwise unwind the stack if the top-most frame is either a **val**-frame or the wrong delimiter, correspondingly. Finally, rule *(handle)* marks the transition back to standard machine reductions.

Extented Syntax for Operational Semantics:

Runtime Labels	l	$::=$	`@a5f` $\mid$ `@4ba` $\mid \ldots$	labels
Runtime Statements	s	$::=$	$\ldots \mid \#_l\ \{\ s\ \}\ \textbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$	delimiters
Runtime Blocks	b	$::=$	$\ldots \mid \textbf{cap}_l$	capabilities
Runtime Capabilities	C	$::=$	$\ldots \mid \{l\}$	label sets
Runtime Signatures	Ξ	$::=$	$\emptyset \mid \Xi,\ l:\ \overrightarrow{\tau_i} \rightarrow \tau_0$	label context

(a) Extended Syntax of System C.

Definition of Values:

Expression Values	v	$::=$	$() \mid 0 \mid 1 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \ldots \mid \textbf{box}\ w$
Block Values	w	$::=$	$\{\ (\overrightarrow{x_i:\ \tau_i},\ \overrightarrow{f_j:\ \sigma_j}) \Rightarrow s\ \} \mid \textbf{cap}_l$

Evaluation Contexts:

Contexts	E	$::=$	$\square \mid \textbf{val}\ x\ =\ \mathsf{E};\ s \mid \#_l\ \{\ \mathsf{E}\ \}\ \textbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$
Delimited Contexts	H_l	$::=$	$\square \mid \textbf{val}\ x\ =\ \mathsf{H}_l;\ s \mid \#_{l'}\ \{\ \mathsf{H}_l\ \}\ \textbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$ where $l \neq l'$

Reduction Rules:

(box)	$\textbf{unbox}\ (\textbf{box}\ b)$	$\longrightarrow$	b
(val)	$\textbf{val}\ x\ =\ \textbf{return}\ v;\ s$	$\longrightarrow$	$s[x \mapsto v]$
(def)	$\textbf{def}\ f\ =\ w;\ s$	$\longrightarrow$	$s[f \mapsto w]$
(ret)	$\#_l\ \{\ \textbf{return}\ v\ \}\ \textbf{with}\ h$	$\longrightarrow$	v
(app)	$(\{\ (\overrightarrow{x_i},\ \overrightarrow{f_j}) \Rightarrow s\ \})(\overrightarrow{v_i},\ \overrightarrow{w_j})$ where $\emptyset \vdash\ w_j\ :\ \sigma_j \mid C_j$	$\longrightarrow$	$s[\overrightarrow{x_i \mapsto v_i},\ \overrightarrow{f_j \mapsto C_j},\ \overrightarrow{f_j \mapsto w_j}]$
(try)	$\textbf{try}\ \{\ f \Rightarrow s\ \}\ \textbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s'\ \}$ where $l\ \notin\ \mathit{dom}\,\Xi$, and $\vdash\ f\ :\ \overrightarrow{\tau_i} \rightarrow \tau_0$, then $\Xi(l)\ :=\ \overrightarrow{\tau_i} \rightarrow \tau_0$	$\longrightarrow$	$\#_l\ \{\ s[f \mapsto \{l\},\ f \mapsto \textbf{cap}_l]\ \}\ \textbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s'\ \}$
(cap)	$\#_l\ \{\ \mathsf{H}_l[\ \textbf{cap}_l(\overrightarrow{v_i})\ ]\ \}\ \textbf{with}\ h$ where $h\ =\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s\ \}$	$\longrightarrow$	$s[\overrightarrow{x_i \mapsto v_i},\ k \mapsto \{\ y \Rightarrow \#_l\ \{\ \mathsf{H}_l[\textbf{return}\ y]\ \}\ \textbf{with}\ h\ \}]$

Congruences:

(cong) If $s \longrightarrow s'$ then $\mathsf{E}[s]\ \longmapsto\ \mathsf{E}[s']$

(b) Operational semantics of System C – we omit trivial congruences for expressions and blocks.

$$\frac{l:\ \overrightarrow{\tau_i} \rightarrow \tau_0\ \in \Xi}{\Gamma\ \vdash\ \textbf{cap}_l\ :\ \overrightarrow{\tau_i} \rightarrow \tau_0 \mid \{l\}}\ [\textsc{Cap}] \qquad \frac{\begin{array}{l} l:\ \overrightarrow{\tau_i} \rightarrow \tau_0\ \in \Xi \qquad \Gamma\ \vdash\ s_1\ :\ \tau \mid C\ \cup \{l\} \\ \Gamma,\ \overrightarrow{x_i:\ \tau_i},\ k :^{C}\ \tau_0 \rightarrow \tau\ \vdash\ s_2\ :\ \tau \mid C \end{array}}{\Gamma\ \vdash\ \#_l\ \{\ s_1\ \}\ \textbf{with}\ \{\ (\overrightarrow{x_i},\ k) \Rightarrow s_2\ \}\ :\ \tau \mid C}\ [\textsc{Reset}]$$

(c) Static semantics of runtime constructs.

Fig. 5. Full description of the operational semantics of the language System C.

Reduction without context:

$$\textit{(def)}\quad \mathbf{def}\ \underline{f}\ =\ w;\ s \quad\longrightarrow\quad s[f \mapsto w]$$

$$\textit{(app)}\quad (\{\ (\overrightarrow{x_i},\ \overrightarrow{f_j}) \Rightarrow s\ \})(\overrightarrow{v_i},\ \overrightarrow{w_j}) \quad\longrightarrow\quad s[\overrightarrow{x_i \mapsto v_i},\ \overrightarrow{f_j \mapsto C_j},\ \overrightarrow{f_j \mapsto w_j}] \quad \text{where } \emptyset \vdash\ w_j\ :\ \sigma_j \mid C_j$$

$$\textit{(box)}\quad \mathbf{unbox}\ (\mathbf{box}\ b) \quad\longrightarrow\quad b$$

Machine reductions:

Standard Machine Reductions

$$\textit{(cong)}\quad \langle\ s \mid \mathsf{E} \mid \Xi\ \rangle \longrightarrow \langle\ s' \mid \mathsf{E} \mid \Xi\ \rangle \qquad \text{if } s \longrightarrow s'$$

$$\textit{(pop)}\quad \langle\ \mathbf{return}\ v \mid \mathbf{val}\ x\ =\ \square;\ s :: \mathsf{E} \mid \Xi\ \rangle \longrightarrow \langle\ s[x \mapsto v] \mid \mathsf{E} \mid \Xi\ \rangle$$

$$\textit{(ret)}\quad \langle\ \mathbf{return}\ v \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ h :: \mathsf{E} \mid \Xi\ \rangle \longrightarrow \langle\ \mathbf{return}\ v \mid \mathsf{E} \mid \Xi\ \rangle$$

$$\textit{(push)}\quad \langle\ \mathbf{val}\ x\ =\ s_1;\ s_2 \mid \mathsf{E} \mid \Xi\ \rangle \longrightarrow \langle\ s_1 \mid \mathbf{val}\ x\ =\ \square;\ s_2 :: \mathsf{E} \mid \Xi\ \rangle$$

$$\textit{(reset)}\quad \langle\ \#_l\ \{\ s\ \}\ \mathbf{with}\ h \mid \mathsf{E} \mid \Xi\ \rangle \longrightarrow \langle\ s \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ h :: \mathsf{E} \mid \Xi\ \rangle$$

Installing Effect Handlers

$$\begin{aligned}\textit{(try)}\quad & \langle\ \mathbf{try}\ \{\ f \Rightarrow s_1\ \}\ \mathbf{with}\ h \mid \mathsf{E} \mid \Xi\ \rangle \longrightarrow \\ & \langle\ s_1[f \mapsto \mathbf{cap}_l,\ f \mapsto \{l\}] \mid \#_l\ \{\ \square\ \}\ \mathbf{with}\ h :: \mathsf{E} \mid \Xi,\ l\ :\ \overrightarrow{\tau_i} \to \tau\ \rangle \\ & \qquad \text{where } l\ \notin \mathit{dom}\ \Xi \quad \text{and} \quad \vdash f\ :\ \overrightarrow{\tau_i} \to \tau\end{aligned}$$

Handling Effect Operations

$$\textit{(cap)}\quad \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid \mathsf{E} \mid \Xi\ \rangle \longrightarrow \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid \mathsf{E} \circ \square \mid \Xi\ \rangle$$

$$\textit{(unwind)}\quad \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid (\mathbf{val}\ x\ =\ \square;\ s :: \mathsf{E}) \circ \mathsf{E}' \mid \Xi\ \rangle \longrightarrow \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid \mathsf{E} \circ (\mathbf{val}\ x\ =\ \mathsf{E}';\ s) \mid \Xi\ \rangle$$

$$\begin{aligned}\textit{(forward)}\quad & \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid (\#_{l'}\ \{\ \square\ \}\ \mathbf{with}\ h :: \mathsf{E}) \circ \mathsf{E}' \mid \Xi\ \rangle \longrightarrow \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid \mathsf{E} \circ (\#_{l'}\ \{\ \mathsf{E}'\ \}\ \mathbf{with}\ h) \mid \Xi\ \rangle \\ & \qquad \text{where } l \neq l'\end{aligned}$$

$$\begin{aligned}\textit{(handle)}\quad & \langle\ \mathbf{cap}_l(\overrightarrow{v}) \mid (\#_l\ \{\ \square\ \}\ \mathbf{with}\ h :: \mathsf{E}) \circ \mathsf{E}' \mid \Xi\ \rangle \longrightarrow \\ & \langle\ s[\overrightarrow{x_i \mapsto v_i},\ k \mapsto \{\ y \Rightarrow \#_l\ \{\ \mathsf{E}'[y]\ \}\ \mathbf{with}\ h\ \}] \mid \mathsf{E} \mid \Xi\ \rangle \\ & \qquad \text{where } h\ =\ \{\ (\overrightarrow{x},\ k) \Rightarrow s\ \}\end{aligned}$$

Fig. 6. Mechanized abstract machine semantics of System C.

Mapping Types:

Value Types

$$\begin{array}{lcl} \lceil \mathsf{Int} \rceil & \approx & \mathsf{Int} \\ \lceil \sigma \,\mathbf{at}\, C \rceil & \approx & U \lceil \sigma \rceil \end{array}$$

Computation Types

$$\begin{array}{lcl} \lceil (\overrightarrow{\tau_i}) \to \tau \rceil & \approx & \lceil \tau_0 \rceil \to \ldots \to \lceil \tau_n \rceil \to F \lceil \tau \rceil \end{array}$$

Value Typing

$$\begin{array}{lcl} \lceil \Gamma \vdash e : \tau \rceil & \approx & \lceil \Gamma \rceil \vdash^{\mathsf{v}} \lceil e \rceil : \lceil \tau \rceil \end{array}$$

Computation Typing

$$\begin{array}{lcl} \lceil \Gamma \vdash s : \tau \mid C \rceil & \approx & \lceil \Gamma \rceil \vdash^{\mathsf{c}} \lceil s \rceil : F \lceil \tau \rceil \\ \lceil \Gamma \vdash b : \sigma \mid C \rceil & \approx & \lceil \Gamma \rceil \vdash^{\mathsf{c}} \lceil b \rceil : \lceil \sigma \rceil \end{array}$$

Mapping Terms:

Value Terms

$$\begin{array}{lcl} \lceil \mathbf{box}\; b \rceil & \approx & \mathbf{thunk} \lceil b \rceil \\ \lceil n \rceil & \approx & n \\ \lceil x \rceil & \approx & x \end{array}$$

Computation Terms

$$\begin{array}{lcl} \mathbf{val}\; x = s_0;\; s_1 \rceil & \approx & \lceil s_0 \rceil \,\mathbf{to}\; x \;\mathbf{in}\, \lceil s_1 \rceil \\ \mathbf{return}\; e & \approx & \mathbf{produce} \lceil e \rceil \\ \lceil b(\overrightarrow{e_i}) \rceil & \approx & \lceil e_n \rceil \text{`} (\ldots (\lceil e_0 \rceil \text{`} \lceil b \rceil)) \\ & & \\ \lceil \mathbf{unbox}\; e \rceil & \approx & \mathbf{force} \lceil e \rceil \\ \{ (\overrightarrow{x_i : \tau_i}) \Rightarrow s \} & \approx & \lambda x_0 \ldots \lambda x_n \lceil s \rceil \end{array}$$

Fig. 7. Mapping concepts from System C to CBPV. The correspondence should only serve as an intuition and still needs to be established formally.

A.5 Call-By-Push-Value

For easier comparison, here we repeat some typing rules for CBPV [Levy 1999]. Value types can be embedded into computation types by means of a type constructor $F\,A$. On the term-level, CBPV features the following rules to embed values in computation and to sequence computation:

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{c}} \mathbf{produce}\; V : F\,A} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : F\,A \qquad \Gamma,\, x : A \vdash^{\mathsf{c}} N : \underline{B}}{\Gamma \vdash^{\mathsf{c}} M \;\mathbf{to}\; x \;\mathbf{in}\; N : \underline{B}}$$

Symmetrically, computation types can be embedded into value types using type constructor $U\,\underline{B}$. Again, on the term-level CBPV features the following two constructs:

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{v}} \mathbf{thunk}\; V : U\,\underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : U\,\underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathbf{force}\; V : \underline{B}}$$

Figure 7 serves as a guide for readers familiar with call-by-push-value.