Stoic: Effects as Capabilities

ANONYMOUS AUTHOR(S)

It is well-known that careless use of side effects in programming results in brittle code and causes subtle bugs, and an effect system can guard against abuse of effects. Unfortunately, effect systems introduce syntactic and cognitive overhead. The verbosity of checked exceptions in Java and the complexity of monad transformers in Haskell are two examples. To overcome such problems, we propose *effects as capabilities* as a paradigm for minimizing the overhead introduced by effect systems. The capability-based approach depends crucially on the fine-grained control over the capture of capabilities in higher-order functions. To this end, we propose a new abstraction: *stoic functions*. We prove that stoic functions enjoy non-interference of memory effects in a step-indexed model. Our system supports *effect polymorphism* with succinct syntax. Also, *effect masking* for local mutational effects works automatically without any special syntax or typing rule.

Additional Key Words and Phrases: stoic function, effect polymorphism, effect masking, capability

1 INTRODUCTION

How do you write an effect-polymorphic function `map` in your favorite programming language? Suppose the function `map` takes a function parameter `f`, a list `l`, and returns a list with `f` applied on each element of list `l`. The effects of the function `map` depends on the effects of the function `f` passed to it: if `f` is pure, then the call `map f l` is pure, and if `f` produces IO effects, then the call `map f l` produces IO effects as well.

In Java, which has an effect system for checking exceptions, we can implement an effect-polymorphic `map` as follows:

```
interface FunctionE<T, U, E extends Exception> {
  public U apply(T t) throws E;
}
interface List<T> {
  public <U, E extends Exception> List<U>
         mapE(FunctionE<T, U, E> f) throws E;
}
```

This is a lot of syntax, and rarely used in practice. In Haskell, the syntax is more concise:

```
mapM :: Monad m => (a -> m b) -> List a -> m (List b)
mapPure :: (a -> b) -> List a -> List b
```

If we choose the monad `m` to be the identity monad, we obtain a pure instance of `mapM`:

```
mapPure f xs = runIdentity (mapM (\x -> return (f x)) xs)
```

However, it is unsatisfactory that programmers need to use a different `map` function depending on whether the function `f` is pure or not. Lippmeier [2010] observes that Haskell has fractured into monadic and non-monadic sub-languages. In Haskell, almost every general purpose higher-order function needs both a monadic version and a non-monadic version.

In Koka [Leijn 2017], only one version of the function `map` is required. A polymorphic `map` has the following signature:

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

Note that the effect variable `e` expresses that the effect of the function `map` is the same as the effect of the parameter `f`. In functional programming, higher-order functions are ubiquitous, most

---

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

of them are effect-polymorphic. Introducing an additional effect variable makes the syntax less palatable and renders the type signature more complex.

It is well-known that the careless use of effectful computation in programming results in brittle code and causes subtle bugs. *Effect systems* track and restrict the use (and abuse) of computational effects [Lucassen and Gifford 1988]. Unfortunately, effect systems introduce syntactic and cognitive overhead. The verbosity of checked exceptions in Java and the complexity with monad transformers in Haskell are two examples.

One major overhead in effect systems is related to effect-polymorphic functions, as in functional programming most higher-order functions are effect-polymorphic. Can we write functions like `map` in a simple way that work both for pure functions and functions with arbitrary effects? This paper shows that with the ideas of *effects as capabilities* and *stoic functions*, this goal can be achieved; in a Scala-like syntax, we can write an effect-polymorphic `map` simply as follows:

```
val map =                             // (Int => Int) -> List[Int] => List[Int]
  (f: Int => Int) => (xs: List[Int]) =>
    xs match {
      case Nil => Nil
      case x :: xs => f(x) :: map(f)(xs)
    }
```

In our system the function `map` has the type $(\mathsf{Int} \Rightarrow \mathsf{Int}) \to \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$. The single arrow ($\to$) denotes a *stoic function*, while the double arrow ($\Rightarrow$) denotes a *free function* (Section 2.1). The function `map` accepts both a pure function and a function with arbitrary effects. There are no effect variables in the definition or the type signature of `map`. We will explain why `map` is effect-polymorphic in Section 2.2.

Contributions

The contributions of this paper are the following:

(1) We identify the concept of *stoic functions* as an abstraction for controlling and reasoning about capabilities (Section 2.1). We formalize stoic functions in λ^{cap}, an extension of STLC with stoic functions and mutations (Section 3.1).
(2) We study the meta-theory of λ^{cap} based on step-indexed models. We prove that stoic functions enjoy *non-interference* of memory effects (section 3.2).
(3) We demonstrate that our system supports a common form of *effect polymorphism* with succinct syntax. Also, *effect masking* for local mutational effects works automatically without any special syntax or typing rule (Section 4).

The benefits of usability are achieved by sacrificing some precision but not soundness of the effect system (Section 4.1).

2 EFFECTS AS CAPABILITIES

We follow a paradigm shift in designing effect systems as introduced in Marino and Millstein [2009]; Odersky [2015]; Osvald et al. [2016]: *instead of saying that a computation may produce some side effects, we say that some capabilities are required in order to carry out the computation.* For example, instead of saying that the function `println` produces input/output side effects, we say that `println` takes an `IO` capability. Capabilities are modeled as values of some capability type, e.g. `Undet` for non-determinism, `IO` for input/output,[1] `Ref T` for mutations. The following is a list of example primitive functions that require corresponding capabilities in order to produce side effect:

```
random  :  Undet -> Int
```

---

[1] Not to be confused with Haskell's `IO` side effects, since Haskell's `IO` allows arbitrary effects.

```
println :  String -> IO -> Unit
read    :  Ref T -> T
write   :  (Ref T, T) -> Unit
ref     :  T -> Ref T
```

Note that there exists one primitive function, i.e. `ref`, for allocating new reference capabilities (memory locations), while it is impossible to create capabilities for `IO` and `Undet`. This makes mutational references more complex than other capabilities, thus our formalization will focus on mutational references (Section 3). Strictly speaking, the types of memory operations (read/write/ref) should be polymorphic. But as we will introduce them as keywords in the calculus and give proper typing rules to them, we omit the universal type quantifier ∀T to simplify presentation.

Since capabilities are required to produce side effects, by tracking capabilities in the type system we can track effects in the program.

However, there is one fundamental difference between the usual notions of capabilities and effects: capabilities can be captured in closures. This means that a capability present at closure construction time can be preserved and accessed when the closure is applied. Effects, on the other hand, are temporal: it generally does make a difference whether an effect occurs when a closure is constructed or when it is used. This is where *stoic functions* come into play.

2.1 Stoic and Free Functions

Intuitively, *free functions* can freely capture capabilities from the environment, while *stoic functions* are more disciplined: they may only use capabilities or free functions provided to them *explicitly* as function arguments; *they never capture capabilities or free functions from the environment*. This is the *capability discipline* that all stoic functions must observe. In short, stoic functions are honest about their effects.

We illustrate stoic and free functions with the following example:

```
val main = (io: IO) => {                          // IO -> Unit
  val mult = (io: IO) => (a: Int) => (b: Int) => { // IO -> Int => Int => Int
    println(a)(io)
    a * b
  }
  val plus = (a: Int) => {             // Int => Int
    println(a)(io)
    a + a
  }
  val double = (a: Int) => plus(a, a)  // Int => Int
}
```

We present our examples in a Scala-like syntax. The syntax `val x = exp` defines a variable `x` bound to the expression `exp`. Braces are used for code blocks; the result of a block is given by its last expression. We write functions as `(x: T) => t`. The types of stoic functions are represented by `T -> R`, while the types of free functions are represented by `T => R`. Functions are inferred to be stoic whenever possible.[2] To avoid cluttering the presentation, we show type signatures of functions as comments instead of type annotations.

In the code above, the function `mult` is stoic, as it does not capture any capabilities or free functions from the environment. Instead, the other functions nested in `main` (that is, `plus` and

---

[2] While we expect such type inference to be unproblematic, we defer a formal study of decidability of type-checking to future work.

`double`) are non-stoic (or free). The function `plus` is non-stoic, as it captures the capability `io`. The function `double` is non-stoic, as it captures the free function `plus`.

Stoic functions can produce free functions, as the following code shows:

```
val main = (io: IO) => {                        // IO -> Unit
  val incStoic = (io: IO) => (a: Int) => {    // IO -> Int => Int
    println(a)(io)
    a + 1
  }
  val incFree = incStoic(io)                     // Int => Int
}
```

The function `incStoic` has the type $\mathsf{IO} \to \mathsf{Int} \Rightarrow \mathsf{Int}$. It is not a surprise that the inner function is non-stoic, as it captures the capability `io` from the environment. Thus the function call `incStoic(io)` creates a free function from a stoic function.

A stoic function can also take a free function as parameter, as shown in the code below:

```
val twice = (f: Int => Int) => (x: Int) => f(f(x))     // (Int => Int) -> Int => Int
```

The function `twice` will accept, as its first argument, both a stoic function and a free function. If we call `twice` with a stoic function, no capabilities will be used directly or indirectly in the execution of `twice`. In general, our type system enables using a stoic function in place of a free function.

2.2 Effect Polymorphism

Let's look again at the function `map`:

```
val map =                                 // (Int => Int) -> List[Int] => List[Int]
  (f: Int => Int) => (xs: List[Int]) =>
    xs match {
      case Nil => Nil
      case x :: xs => f(x) :: map(f)(xs)
    }
```

The function `map` has the type signature $(\mathsf{Int} \Rightarrow \mathsf{Int}) \to \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$. The outer function is stoic, as it does not capture capabilities nor free functions from the environment. The inner function captures the free function `f`, thus it is non-stoic. In a function call `map(f)(l)`, the inner function may only use capabilities carried by `f`. The function `f` can be either a stoic function $(\mathsf{Int} \to \mathsf{Int})$ or a free function $(\mathsf{Int} \Rightarrow \mathsf{Int})$ that produces effects. In this sense, the function `map` is effect-polymorphic. The following example demonstrates the usage:

```
  val f = (xs: List[Int]) => {                  // List[Int] -> List[Int]
    val sum = ref 0
    map { x => sum := (read sum) + x; x * x } xs
    map { x -> x * x } xs
  }
```

Sometimes, when we partially apply the function `map` with a stoic function f of the type $\mathsf{Int} \to \mathsf{Int}$, we expect the result type to be $\mathsf{List[Int]} \to \mathsf{List[Int]}$. This is achieved by η-expansion in our system (Section 4.1), as the following code shows:

```
val mapEta = (xs: List[Int]) => map { x -> x * x } xs // List[Int] -> List[Int]
```

In the above snippet, the function `mapEta` is stoic because it captures from its environment neither capabilities nor free functions. If `map` is instead applied to a free function `f` of the type $\mathsf{Int} \Rightarrow \mathsf{Int}$, then neither `map f` nor its η-expansion `(xs: List[Int]) => map f xs` will be stoic, and they will both have the type $\mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$. Similarly, if the function `map` had the signature

(Int ⇒ Int) ⇒ List[Int] ⇒ List[Int], the call `map(f)(l)` may use more capabilities than what is provided by `f`, as the function `map` may capture capabilities from the environment itself. Trying to call such a function `map` from a stoic function will result in a typing error, as it violates the capability discipline of stoic functions.

2.3 Effect Propagation

If a function `f` calls another function `g` inside its body, the effects produced by the function `g` should be propagated to the function `f`. In contrast to type-and-effect systems [Lucassen and Gifford 1988], in capability-based effect systems capabilities propagate from the caller to the callee, which makes sense because capabilities are *permissions* to perform effects.

However, there is another way to propagate effects in capability-based effect systems: *capturing capabilities*. This can be demonstrated by the following example:

```
val complex = (x: Int) => (io: IO) => {           // Int -> IO -> Int
  val f = (a: Int) => { println(a)(io); a * a }    // Int => Int
  val g = (a: Int) => { println(a)(io); a + a }    // Int => Int
  f(x) + g(x)
}
```

The function `complex` is stoic, as it does not capture any capabilities except the explicitly given capability `io`. However, the implementation of `complex` is based on the non-stoic functions `f` and `g`, which capture `io` from the environment. Note that `f` and `g` cannot capture any capabilities beyond those explicitly given to `complex`, otherwise `complex` could not be stoic. This saves boilerplate for threading the capabilities through function calls. Otherwise, we would have to write this code more verbosely:

```
val complex = (x: Int) => (io: IO) => {                     // Int -> IO -> Int
  val f = (a: Int) => (io: IO) => { println(a)(io); a * a } // Int -> IO -> Int
  val g = (a: Int) => (io: IO) => { println(a)(io); a + a } // Int -> IO -> Int
  f(x)(io) + g(x)(io)
}
```

For programming languages that support Scala-like implicits or implicit function types [Odersky et al. 2018], the syntax can be cut even further:

```
type IO[T] = implicit IO -> T
def complex(x: Int): IO[Int] = {
  def f(a: Int): Int = { println(a); a * a }
  def g(a: Int): Int = { println(a); a + a }
  f(x) + g(x)
}
```

2.4 Combining Effects

Suppose we want to write a function to print the content of a memory reference. This task requires combining two effects: memory access and I/O. By treating effects as capabilities, combining multiple effects requires simply abstracting over multiple capabilities:

```
val inspect = (r: Ref[Int]) => (io: IO) =>       // Ref[Int] -> IO => Unit
  print(read(r))(io)
```

2.5 Incremental Adoption

An effect system is like an armor that protects programmers from tricky bugs caused by abuse of effects. However, the merits of such an armor does not justify that programmers should carry its weight in all development scenarios, at all stages and for all components of a program. In a quick prototype, programmers may choose to ignore checked effects completely. In a larger project, the choice of which components should be effect-disciplined may evolve over time.

With both stoic and free functions, our system supports easily *incremental adoption* of effects. If programmers decide to not track effects, they can just use free functions throughout in the program. During software development, if programmers want to make more components effect-disciplined, it suffices to change some free functions to stoic functions and making their effects explicit. We believe enabling programmers to incrementally make code effect-disciplined is another key factor in the adoption of effect systems.

3 CALCULUS

We formalize the concept of *stoic functions* in call-by-value simply typed lambda calculus extended with mutation, taking heap references as capabilities. We study the meta-theory of the system following a semantic approach based on step-indexed models as in Ahmed [2004].

On a first reading, readers can safely ignore the meta-theory based on step-indexed models and come back to it later.

3.1 Definition

The calculus is presented in Figure 1; the syntax is mostly standard. Types are separated into two groups: *pure types* ($\mathsf{T_{pu}}$) and *impure types* ($\mathsf{T_{im}}$). Impure types include capabilities ($\mathsf{Ref\ T}$) and free function types ($\mathsf{T} \Rightarrow \mathsf{T}$). All other types are pure, including unit type, naturals and stoic function types ($\mathsf{T} \to \mathsf{T}$).

The small-step semantics is presented using evaluation contexts. We let S range over *stores*, which are finite maps from locations to values. We write one-step reduction as $(\mathsf{S}, \mathsf{t}) \longrightarrow (\mathsf{S}', \mathsf{t}')$, which means the term t with the store S takes one step to t' with the updated store S'.

The typing judgments are of the form $\Gamma \vdash \mathsf{t} : \mathsf{T}$, which means the term t can be typed as T under the environment Γ. Instead of proving soundness through progress and preservation, we will take a semantic approach to soundness: we define a semantics of types and typing judgements, and then prove typing rules as theorems (Section 3.2). The semantic approach allows us to restrict source programs to contain no locations, thus the typing judgments need not mention store typing and we can omit the usual typing rule for locations [Pierce 2002, Chap. 13].

The most important change in typing rules is the introduction of the typing rule T-Stoic, which assigns type to stoic functions. In contrast to the standard typing rule T-Abs for functions, it purifies the environment in typing stoic functions. This is how the *capability discipline* is enforced in the type system. The capability discipline is implemented with the helper function pure, which removes all variables of impure types from the typing environment.

Note also that in the typing rule T-Stoic, we restrict the term to be a value, which can only be a lambda in this context. This restriction is important, we will discuss it in Section 4.2.

The rule T-Degen says that a stoic function can be used as a free function, it is a dual of the rule T-Stoic.

3.2 Semantic Typing

On a first reading, readers can jump to Section 4 and come back later.

Syntax

$$
\begin{array}{lllr}
\mathsf{t} & ::= & & \text{terms:} \\
 & & \mathsf{x} & \text{variable} \\
 & & \lambda \mathsf{x}{:}\mathsf{T}.\,\mathsf{t} & \text{abstraction} \\
 & & \mathsf{t}\ \mathsf{t} & \text{application} \\
 & & \mathsf{l} & \text{locations} \\
 & & \mathsf{ref}\ \mathsf{t} & \text{new memory} \\
 & & \mathsf{t} := \mathsf{t} & \text{assignment} \\
 & & !\mathsf{t} & \text{dereference} \\
 & & \mathsf{unit} & \text{unit value} \\
 & & \mathsf{n} & \text{naturals} \\
 & & & \\
\mathsf{v} & ::= & & \text{values:} \\
 & & \lambda \mathsf{x}{:}\mathsf{T}.\,\mathsf{t} & \text{abstraction value} \\
 & & \mathsf{unit} & \text{unit value} \\
 & & \mathsf{n} & \text{naturals} \\
 & & \mathsf{l} & \text{location values} \\
 & & & \\
\mathsf{T}_{\mathsf{pu}} & ::= & & \text{pure types:} \\
 & & \mathsf{Nat} & \text{naturals} \\
 & & \mathsf{Unit} & \text{unit type} \\
 & & \mathsf{T} \rightarrow \mathsf{T} & \text{stoic funs} \\
 & & & \\
\mathsf{T}_{\mathsf{im}} & ::= & & \text{impure types:} \\
 & & \mathsf{Ref}\ \mathsf{T} & \text{references} \\
 & & \mathsf{T} \Rightarrow \mathsf{T} & \text{free funs} \\
 & & & \\
\mathsf{T} & ::= & \mathsf{T}_{\mathsf{pu}} \mid \mathsf{T}_{\mathsf{im}} & \text{types}
\end{array}
$$

Evaluation $\boxed{(\mathsf{S}, \mathsf{t}) \longrightarrow (\mathsf{S}, \mathsf{t}')}$

$$\mathsf{E} ::= [\cdot] \mid \mathsf{E}\ \mathsf{t} \mid \mathsf{v}\ \mathsf{E} \mid \mathsf{ref}\ \mathsf{E} \mid !\mathsf{E} \mid \mathsf{E} := \mathsf{t} \mid \mathsf{v} := \mathsf{E}$$

$$\frac{\mathsf{t} \longrightarrow \mathsf{t}'}{\mathsf{E}[\mathsf{t}] \longrightarrow \mathsf{E}[\mathsf{t}']} \quad (\textsc{E-Context})$$

$$(\lambda \mathsf{x}{:}\mathsf{T}.\,\mathsf{t}_1)\ \mathsf{v}_2 \longrightarrow [\mathsf{x} \mapsto \mathsf{v}_2]\mathsf{t}_1 \quad (\textsc{E-Beta})$$

$$\frac{\mathsf{l} \notin \mathsf{dom}(\mathsf{S})}{(\mathsf{S}, \mathsf{ref}\ \mathsf{v}) \longrightarrow (\mathsf{S}[\mathsf{l} \mapsto \mathsf{v}], \mathsf{l})} \quad (\textsc{E-Ref})$$

$$\frac{\mathsf{l} \in \mathsf{dom}(\mathsf{S})}{(\mathsf{S}, !\mathsf{l}) \longrightarrow (\mathsf{S}, \mathsf{S}(\mathsf{l}))} \quad (\textsc{E-Deref})$$

$$\frac{\mathsf{l} \in \mathsf{dom}(\mathsf{S})}{(\mathsf{S}, \mathsf{l} := \mathsf{v}) \longrightarrow (\mathsf{S}[\mathsf{l} \mapsto \mathsf{v}], \mathsf{unit})} \quad (\textsc{E-Assign})$$

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

$$\Gamma \vdash \mathsf{unit} : \mathsf{Unit} \quad (\textsc{T-Unit})$$

$$\Gamma \vdash \mathsf{n} : \mathsf{Nat} \quad (\textsc{T-Nat})$$

$$\frac{\mathsf{x} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}} \quad (\textsc{T-Var})$$

$$\frac{\Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{T}_1.\,\mathsf{t}_2 : \mathsf{T}_1 \Rightarrow \mathsf{T}_2} \quad (\textsc{T-Abs})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \Rightarrow \mathsf{T}_2 \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_1}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}_2} \quad (\textsc{T-App})$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \mathsf{ref}\ \mathsf{t} : \mathsf{Ref}\ \mathsf{T}} \quad (\textsc{T-Ref})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{Ref}\ \mathsf{T} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \mathsf{t}_1 := \mathsf{t}_2 : \mathsf{Unit}} \quad (\textsc{T-Assign})$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{Ref}\ \mathsf{T}}{\Gamma \vdash !\mathsf{t} : \mathsf{T}} \quad (\textsc{T-Deref})$$

$$\frac{\mathsf{pure}(\Gamma) \vdash \mathsf{v} : \mathsf{T}_1 \Rightarrow \mathsf{T}_2}{\Gamma \vdash \mathsf{v} : \mathsf{T}_1 \rightarrow \mathsf{T}_2} \quad (\textsc{T-Stoic})$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{T}_1 \rightarrow \mathsf{T}_2}{\Gamma \vdash \mathsf{t} : \mathsf{T}_2 \Rightarrow \mathsf{T}_2} \quad (\textsc{T-Degen})$$

Pure Environment

$$
\begin{array}{lcl}
\mathsf{pure}(\varnothing) & = & \varnothing \\
\mathsf{pure}(\Gamma, \mathsf{x}{:}\mathsf{T}_{\mathsf{im}}) & = & \mathsf{pure}(\Gamma) \\
\mathsf{pure}(\Gamma, \mathsf{x}{:}\mathsf{T}_{\mathsf{pu}}) & = & \mathsf{pure}(\Gamma), \mathsf{x}{:}\mathsf{T}_{\mathsf{pu}}
\end{array}
$$

Fig. 1. Syntax and Syntactic Typing for λ^{cap}

To prove soundness of the system, we follow the step-indexed approach as demonstrated in Ahmed [2004]. Actually, we will reuse most of the definitions and proofs in section 3.3 of the thesis, thanks to composability of semantic typing.

Step-indexes (written as j or k) are natural numbers used both to count evaluation steps and to avoid circularities in the definition of store typings and semantic types.

Motivating step-indexed models. Step-indexed models interpret syntactic types T as semantic types τ, which are predicates on values and store typings. In turn, store typings Ψ map locations to semantic types. Roughly, semantic type $[\![T_1 \Rightarrow T_2]\!]$ is satisfied by $\langle v, \Psi \rangle$ if the value v, when run in a store matching store typing Ψ, runs *safely* (without getting stuck) and maps argument values in $[\![T_1]\!]$ to result expressions in $[\![T_2]\!]^*$.

The definitions of semantic typings and store typings have a problematic circularity, so instead of performing these definitions in one go, a semantic type is defined to be a *step-indexed* family of sets, which serves as a sequence of approximations of the "correct" semantic type. When defining the k-th approximation of a semantic type, any circularity can be resolved by referring to approximations at step-indexes j smaller than k.

Moreover, general references allow constructing recursive functions v; showing that recursive functions are safe also has circularity problems, because v can only be shown safe if recursive calls to v are also safe. To fix this circularity, the k-th approximation of a semantic type only constrains the behavior of a value when observed for up to k steps; to show a recursive function v safe for up to k steps, we only need to assume recursive calls to v safe for fewer steps.

Step-indexed models. Because of the reasons explained, a semantic type τ is a set of triples $\langle k, \Psi, v \rangle$. Roughly speaking, $\langle k, \Psi, v \rangle \in [\![T]\!]$ means that, in any store that matches store typing Ψ, the value v behaves as a value of type T, when tested for up to k evaluation steps. For example, if the value v satisfies $[\![T_1 \Rightarrow T_2]\!]$, then v must be a function value, and the result of applying this function value to an input in $[\![T_1]\!]$ must satisfy $[\![T_2]\!]$ (up to a certain number of steps).

A key insight on the connection between step-indexed models and capabilities, alluded in the footnote of [Ahmed 2004, P. 55], is that the store typing Ψ in the tuple $\langle k, \Psi, t \rangle$ can be read as the resources (or capabilities from our perspective) that are sufficient for the safe evaluation of t for k steps.

The semantic approach requires us to first give meanings to types and typing judgments, and then prove that all typing rules hold semantically. For completeness, we first reproduce the basic definitions from Ahmed [2004] below.[3] As a convention, we write $\langle k, \Psi, t \rangle$ as a short-hand for $\langle k, \lfloor \Psi \rfloor_k, t \rangle$ to simplify the presentation.

3.2.1 Basic Definitions.

Definition 3.1 (Safe). A state (S, t) is safe for k steps if for any reduction $(S, t) \longrightarrow^j (S', t')$ of $j < k$ steps, either t' is a value or another step is possible.

$$\mathsf{safen}(k, S, t) \triangleq \forall j, S', t'.(j < k \wedge (S, t) \longrightarrow^j (S', t')) \Longrightarrow (\mathsf{val}(t') \vee \exists S'', t''.(S', t') \longrightarrow (S'', t''))$$

A state (S, t) is called safe if it is safe for any step count.

$$\mathsf{safe}(S, t) \triangleq \forall k.\mathsf{safen}(k, S, t)$$

Definition 3.2 (Approx). The k-approximation of a semantic type is the subset of its elements whose index is less than k. This concept is extended point-wise to store typings:

$$\begin{array}{rcl} \lfloor \tau \rfloor_k & \triangleq & \{ \langle j, \Psi, v \rangle \mid j < k \wedge \langle j, \Psi, v \rangle \in \tau \} \\ \lfloor \Psi \rfloor_k & \triangleq & \{ (l \mapsto \lfloor \tau \rfloor_k) \mid \Psi(l) = \tau \} \end{array}$$

---

[3] With minor adaptations.

Definition 3.3 (State Extension). A valid state extension is defined as follows:

$$(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{j}, \Psi') \triangleq \mathsf{j} \leq \mathsf{k} \wedge \forall \mathsf{l} \in \mathsf{dom}(\Psi).\lfloor \Psi' \rfloor_{\mathsf{j}}(\mathsf{l}) = \lfloor \Psi \rfloor_{\mathsf{j}}(\mathsf{l})$$

Definition 3.4 (Extensibility). A set τ of tuples of the form $\langle \mathsf{k}, \Psi, \mathsf{v} \rangle$, where v is a value, k is a nonnegative integer, and Ψ is a store typing, is *extensible* if τ is closed under state extension; that is,

$$\mathsf{extensible}(\tau) \triangleq \forall \mathsf{k}, \mathsf{j}, \Psi, \Psi', \mathsf{v}.\langle \mathsf{k}, \Psi, \mathsf{v} \rangle \in \tau \wedge (\mathsf{k}, \Psi) \sqsubseteq (\mathsf{j}, \Psi') \Longrightarrow \langle \mathsf{j}, \Psi', \mathsf{v} \rangle \in \tau$$

In the type definitions that follow, when we universally quantify over a store typing Ψ, we implicitly require that $\forall \mathsf{l} \in \mathsf{dom}(\Psi).\mathsf{extensible}(\Psi(\mathsf{l}))$. When we shrink the approximation index of store typings, this invariant is preserved due to the following facts:

- All store typings and types in store typings are step-indexed (explicitly or implicitly), i.e. of the form $\lfloor \Psi \rfloor_{\mathsf{k}}$ and $\lfloor \tau \rfloor_{\mathsf{k}}$.
- If $\mathsf{extensible}(\lfloor \tau \rfloor_{\mathsf{k}})$ and $\mathsf{j} < \mathsf{k}$, then $\mathsf{extensible}(\lfloor \tau \rfloor_{\mathsf{j}})$ (Lemma Extensibility Weakening).

Definition 3.5 (Well-typed Store). A store S is well-typed to approximation k with respect to a store typing Ψ iff $\mathsf{dom}(\Psi) \subseteq \mathsf{dom}(\mathsf{S})$ and the contents of each location $\mathsf{l} \in \mathsf{dom}(\Psi)$ has type $\Psi(\mathsf{l})$ to approximation k:

$$\mathsf{S} :_{\mathsf{k}} \Psi \triangleq \mathsf{dom}(\Psi) \subseteq \mathsf{dom}(\mathsf{S}) \wedge \forall \mathsf{j} < \mathsf{k}.\forall \mathsf{l} \in \mathsf{dom}(\Psi).\langle \mathsf{j}, \lfloor \Psi \rfloor_{\mathsf{j}}, \mathsf{S}(\mathsf{l}) \rangle \in \lfloor \Psi \rfloor_{\mathsf{k}}(\mathsf{l})$$

Definition 3.6 (Semantic Typing Judgement). For any type environment Γ and value environment σ, we write $\sigma :_{\mathsf{k},\Psi} \Gamma$ if for all variables $\mathsf{x} \in \mathsf{dom}(\Gamma)$ we have $\langle \mathsf{k}, \Psi, \sigma(\mathsf{x}) \rangle \in [\![\Gamma(\mathsf{x})]\!]$; that is

$$\sigma :_{\mathsf{k},\Psi} \Gamma \triangleq \forall \mathsf{x} \in \mathsf{dom}(\Gamma).\langle \mathsf{k}, \Psi, \sigma(\mathsf{x}) \rangle \in [\![\Gamma(\mathsf{x})]\!]$$

The semantic typing judgement is then defined as:

$$\Gamma \models \mathsf{t} : \mathsf{T} \triangleq \mathsf{FV}(\mathsf{t}) \subseteq \mathsf{dom}(\Gamma) \wedge (\forall \mathsf{k}, \sigma, \Psi.\sigma :_{\mathsf{k},\Psi} \Gamma \Longrightarrow \langle \mathsf{k}, \Psi, \sigma(\mathsf{t}) \rangle \in [\![\mathsf{T}]\!]^*)$$
$$\models \mathsf{t} : \mathsf{T} \triangleq \emptyset \models \mathsf{t} : \mathsf{T}$$

3.2.2 *Interpretation of Types.*

The interpretation of syntactic types are given in Figure 2. $[\![\mathsf{T}]\!]$ defines what it means for a value to belong to a type, and $[\![\mathsf{T}]\!]^*$ defines what it means for a term to belong to a type.

Any natural can safely take any number of steps with any capabilities — as it does not consume any resources, thus there is no requirement on Ψ. The interpretation for Unit is similar.

Function values in $\mathsf{T}_1 \Rightarrow \mathsf{T}_2$ must map argument values in T_1 to result expressions in T_2. More precisely, $\langle \mathsf{k}, \Psi, \lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t} \rangle \in [\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]$ requires that function body t satisfies $[\![\mathsf{T}_2]\!]$ for at least $\mathsf{j} < \mathsf{k}$ steps when applied to an argument that satisfies $[\![\mathsf{T}_1]\!]$ for j steps. Moreover, a value of the free function type $\mathsf{T}_1 \Rightarrow \mathsf{T}_2$ may capture references from the environment, and can assume the references in Ψ are available; so we only constraint the behavior of the body t for stores satisfying store typings Ψ' that extend Ψ. As Ψ' could be Ψ and j could be $\mathsf{k} - 1$, the store typing Ψ must at least contain the necessary capabilities for the function body t to take $\mathsf{k} - 1$ steps.

The interpretation for $\mathsf{T}_1 \rightarrow \mathsf{T}_2$ is similar. The key difference is that (j, Ψ') does not need to extend (k, Ψ): there is no constraint on Ψ'. As Ψ' could be empty, this ensures that a stoic function can only use capabilities provided via its arguments.

For references, the definition requires that the capabilities provided should map the location l to the right type. Note the condition does not directly say anything about the capabilities that the value at l may need for safe execution; however, if a store S is well-typed with respect to Ψ, then the value at S(l) and the store S itself will have to satisfy $\Psi(\mathsf{l})$ and hence T (up to a suitable approximation). This reflects an improvement of the step-indexed proof technique in [Ahmed 2004] over [Ahmed et al. 2003]. In the latter, the logical relation is defined on the quadruple $\langle \mathsf{k}, \Psi, \mathsf{S}, \mathsf{v} \rangle$, as we need to check that for all $\mathsf{l} \in \mathsf{dom}(\Psi)$, S(l) can safely take j steps with $\lfloor \Psi \rfloor_{\mathsf{j}}$ and S. Ahmed

$$
\begin{aligned}
[\![\mathrm{Nat}]\!] &\triangleq \{ \langle k, \Psi, n \rangle \} \\
[\![\mathrm{Unit}]\!] &\triangleq \{ \langle k, \Psi, \mathrm{unit} \rangle \} \\
[\![T_1 \Rightarrow T_2]\!] &\triangleq \{ \langle k, \Psi, \lambda x{:}T_1.t \rangle \mid \forall v, \Psi', j < k. \\
&\qquad ((k, \Psi) \sqsubseteq (j, \Psi') \wedge \langle j, \Psi', v \rangle \in [\![T_1]\!]) \Longrightarrow \langle j, \Psi', t[v/x] \rangle \in [\![T_2]\!]^* \} \\
[\![T_1 \rightarrow T_2]\!] &\triangleq \{ \langle k, \Psi, \lambda x{:}T_1.t \rangle \mid \forall v, \Psi', j < k. \\
&\qquad \langle j, \Psi', v \rangle \in [\![T_1]\!] \Longrightarrow \langle j, \Psi', t[v/x] \rangle \in [\![T_2]\!]^* \} \\
[\![\mathrm{Ref}\ T]\!] &\triangleq \{ \langle k, \Psi, l \rangle \mid \lfloor \Psi \rfloor_k(l) = \lfloor [\![T]\!] \rfloor_k \} \\
[\![T]\!]^* &\triangleq \{ \langle k, \Psi, t \rangle \mid \mathrm{val}(t) \wedge \langle k, \Psi, t \rangle \in [\![T]\!] \ \bigvee \\
&\quad \neg\mathrm{val}(t) \wedge \forall j, S, S', t'. \\
&\quad (j < k \wedge S :_k \Psi \wedge (S, t) \longrightarrow^j (S', t') \wedge \mathrm{irred}(S', t')) \\
&\quad \Longrightarrow \exists \Psi'.(k, \Psi) \sqsubseteq (k - j, \Psi') \wedge S' :_{k-j} \Psi' \wedge \\
&\qquad \langle k - j, \Psi', t' \rangle \in [\![T]\!] \qquad \}
\end{aligned}
$$

Fig. 2. Semantic Typing for λ^{cap}

[2004] shows that one can remove S from the quadruple and simplify the definition of $[\![\mathrm{Ref}\ T]\!]$ with an additional definition of *well-typed store* and impose that condition in expression typings, i.e. well-formed store will keep well-formed during evaluation.

The expression typing specifies the condition for a term t to safely take k steps with the capabilities Ψ. If t is a value,[4] then $\langle k, \Psi, t \rangle \in [\![T]\!]^*$ is equivalent to $\langle k, \Psi, t \rangle \in [\![T]\!]$. Otherwise, given a well-typed store S with respect to Ψ, if (S, t) reduces to an irreducible state (S', t') in j steps for any $j < k$, then S' should be well-typed in an extended store typing Ψ', and t' should be a value of type T that can safely take $k - j$ steps with the capabilities Ψ'.

3.2.3 *Soundness.*

THEOREM 3.7 (SOUNDNESS). *If* $\models t : T$, *and* S *is a store, then* (S, t) *is safe.*

PROOF. We need to show that for any k, (S, t) is safe for k steps.

From the definition of semantic typing judgments, we know that for any k', Ψ, we have $\langle k', \Psi, t \rangle \in [\![T]\!]^*$. In particular, it holds for $\Psi = \emptyset$ and $k' = k$. It is obvious that $S :_k \Psi$.

From the definition of $[\![T]\!]^*$, the case that t is a value is trivial, otherwise either (S, t) can safely take k steps without reducing to a value, which concludes the proof; or $(S, t) \longrightarrow^j (S', t')$ for $j < k$ steps, and there exists some Ψ' such that $(k - j, \Psi', t') \in [\![T]\!]$. From the definition of $[\![T]\!]$, we know t' must be a value, thus (S, t) is safe for k steps. □

Definition 3.8 (Non-interference). A term t of type T is non-interferent with the store typing Ψ_1, written as $t : T \# \Psi_1$, if and only if for any k, there exists Ψ_2 with $\mathrm{dom}(\Psi_1) \cap \mathrm{dom}(\Psi_2) = \emptyset$ such that $\langle k, \Psi_2, t \rangle \in [\![T]\!]^*$.

---

[4] We need to make the case explicit in the definition, as it is needed in the proof of T-STOIC: we want to ensure that if $\langle k, \Psi, v \rangle \in [\![T]\!]^*$ then $\langle k, \Psi, v \rangle \in [\![T]\!]$ with the same Ψ, not just with some extension of Ψ.

The definition depends on the following observation: if $\langle k, \Psi, t\rangle \in [\![T]\!]^*$, then the store typing Ψ are the resources (or capabilities) that are sufficient for the safe evaluation of t for k steps. If T is a function type, the definition of $[\![T_1 \rightarrow T_2]\!]$ and $[\![T_1 \Rightarrow T_2]\!]$ also ensures that execution of the function body is safe. As k can be any number in the definition, it is impossible for t to read, write or refer to any memory locations in Ψ_1.

In the following example, both `get` and `inc` interfere with their environments, as the memory location `m` is captured and used (read/write):

```
val m = ref 0
val get = () => !m              // Unit => Int
val inc = () => m := !m + 1     // Unit => Unit
```

Moreover, the following function will be taken as interferent as well according to the definition:

```
val f = {                       // Int => Ref Int
  val m = ref 0
  (x: Int) => m                 // capture, but no read/write
}
```

In our system, the function f will be typed as $\mathsf{Int} \Rightarrow \mathsf{Ref\ Int}$. Rejecting the function as stoic is important, otherwise it will be unsafe to use stoic functions in multiple threads. In the example above, if we use the function f in two different threads, it may lead to data races on the shared location m. A stoic function should only use explicitly provided memory locations or create new memory locations, but not secretly capture memory locations.

To simplify presentation, we also use the following definitions:

(1) $\sigma :_\Psi \Gamma \triangleq \forall k.\sigma :_{k,\Psi} \Gamma$
(2) $v :_\Psi T \triangleq \forall k.\langle k, \Psi, v\rangle \in [\![T]\!]$
(3) $\Psi_1 - \Psi_2 \triangleq \{\, (l \mapsto \tau) \mid \Psi_1(l) = \tau \wedge l \notin dom(\Psi_2) \,\}$

Theorem 3.9 (Non-interference). *If* $\Gamma \models \lambda x{:}T_1.t : T_1 \rightarrow T_2$, $\forall \Psi, \sigma, v, \Psi_1$, *if* $\sigma :_\Psi \Gamma$ *and* $v :_{\Psi_1} T_1$, *we have* $\sigma(t)[v/x] : T_2 \mathbin{\#} \Psi - \Psi_1$.

Proof. By the definition of non-interference, we need to prove that for any step-index k, there exists Ψ' such that $dom(\Psi - \Psi_1) \cap dom(\Psi') = \emptyset$ and $\langle k, \Psi', \sigma(t)[v/x]\rangle \in [\![T_2]\!]^*$.

We choose $\Psi' = \Psi_1$, it is obvious that $dom(\Psi - \Psi_1) \cap dom(\Psi_1) = \emptyset$, by the definition of store typing subtraction. Without loss of generality, let's choose some $m > k$, from the definition of semantic judgments and the fact that $\sigma(\lambda x{:}\, T_1.t)$ is a value, we have the following:

$$\langle m, \Psi, \sigma(\lambda x{:}\, T_1.t)\rangle \in [\![T_1 \rightarrow T_2]\!]$$

Now by the definition of $[\![T_1 \rightarrow T_2]\!]$ and $\langle m, v, \Psi_1\rangle \in [\![T_1]\!]$, we have $\forall j < m$, $\langle j, \Psi_1, \sigma(t)[v/x]\rangle \in [\![T_2]\!]^*$. In particular, it holds for k, as we know $k < m$. □

This theorem says that if an function is typed as stoic under an environment, calling the resulting function will not read/write any memory locations from the outer environment, except those explicitly provided as an argument.

In the other direction, if the argument type and return type of a stoic function are both pure types (e.g. Nat or Unit), it is impossible for the environment to read/write locally created memory locations after execution of the stoic function. In such cases, stoic functions create completely segregated regions of memory.

In another word, the only doors that enable interference of local memory of a stoic function and its environmental memory is via function argument and return value. By controlling the front- and back-door, it is possible to predict what effects are possible during and after a stoic function call.

This is not true for free functions, as *capturing* provides a privileged channel for them to interact with the environment.

3.3 Basic Lemmas

We list the basic lemmas that will be used in the proofs. These lemmas are easy to prove, and readers can find most of the proofs in section 3.4 of Ahmed [2004].

LEMMA 3.10 (STATE EXTENSION REFLEXIVE). $(k, \Psi) \sqsubseteq (k, \Psi)$.

LEMMA 3.11 (STATE EXTENSION TRANSITIVE). *If* $(k_1, \Psi_1) \sqsubseteq (k_2, \Psi_2)$ *and* $(k_2, \Psi_2) \sqsubseteq (k_3, \Psi_3)$*, then* $(k_1, \Psi_1) \sqsubseteq (k_3, \Psi_3)$.

LEMMA 3.12 (TYPE SET EXTENSIBLE). *For any syntactic type* T, $\mathrm{extensible}(\llbracket T \rrbracket)$.

LEMMA 3.13 (EXTENSIBILITY WEAKENING). *If* $\mathrm{extensible}(\lfloor \tau \rfloor_k)$ *and* $j \leq k$ *then* $\mathrm{extensible}(\lfloor \tau \rfloor_j)$.

LEMMA 3.14 (INDEX CUT). *If* $(k, \Psi) \in (j, \Psi')$, $i < k$*, and* $i < j$*, then* $(i, \lfloor \Psi \rfloor_i) \sqsubseteq (i, \lfloor \Psi' \rfloor_i)$.

LEMMA 3.15 (INDEX WEAKENING). *If* $j < k$*, then* $(k, \Psi) \sqsubseteq (j, \Psi)$.

LEMMA 3.16 (DETERMINISM OF EVALUATION). *If* $(S, t) \longrightarrow^i (S_1, t_1) \wedge \mathrm{irred}(S_1, t_1)$ *and* $(S, t) \longrightarrow^j (S_2, t_2) \wedge \mathrm{irred}(S_2, t_2)$*, then* $S_1 = S_2$, $t_1 = t_2$ *and* $i = j$.

LEMMA 3.17 (STORE INDEX WEAKENING). *If* $S :_k \Psi$ *and* $j < k$*, then* $S :_j \Psi$.

3.4 Proof of Typing Rules

To relate our syntactic type judgement Γ with semantic typing, we must prove that our syntactic typing rules are *sound* relative to semantic typing, as stated in the following theorem.

THEOREM 3.18 (SOUNDNESS OF SYNTACTIC TYPING). *If* $\Gamma \vdash t : T$ *then* $\Gamma \models t : T$.

This theorem is proven by induction on derivations of $\Gamma \vdash t : T$. Each case can be shown as a separate typing lemma, and we show a selection of such lemmas in the rest of this section.

The typing rules T-NAT, T-UNIT and T-VAR are trivial to prove sound, thus are omitted. Only the proofs for T-STOIC and T-DEGEN are new, other proofs are similar to those in Section 3.5 of Ahmed [2004]. Thus we only show the proofs for T-STOIC and T-DEGEN here, and keep other proofs in the appendix.

LEMMA 3.19 (PURE TYPE). *If* $\langle k, \Psi, v \rangle \in \llbracket T_{pu} \rrbracket$*, then* $\langle k, \emptyset, v \rangle \in \llbracket T_{pu} \rrbracket$.

PROOF. There are three cases: Unit, Nat, $T_1 \rightarrow T_2$. In each case, the definition of $\llbracket T \rrbracket$ does not depend on Ψ, thus we can always choose $\Psi = \emptyset$. □

THEOREM 3.20 (STOIC). *The following typing rule holds:*

$$\frac{\mathrm{pure}(\Gamma) \models v : T_1 \Rightarrow T_2}{\Gamma \models v : T_1 \rightarrow T_2} \qquad \text{(T-STOIC)}$$

PROOF. We need to show that for all k, σ, Ψ, if $\sigma :_{k,\Psi} \Gamma$, then:

(G1) $FV(v) \subseteq \mathrm{dom}(\Gamma)$

(G2) $\langle k, \Psi, \sigma(v) \rangle \in \llbracket T_1 \rightarrow T_2 \rrbracket^*$

Without loss of generality, we choose k, σ, Ψ such that $\sigma :_{k,\Psi} \Gamma$. From the definition of pure, we have for all $x \in \mathrm{pure}(\Gamma)$, $(\mathrm{pure}(\Gamma))(x) = \Gamma(x)$. Now from the definition of environment typing, we have $\sigma :_{k,\Psi} \mathrm{pure}(\Gamma)$.

By the definition of $\llbracket T_1 \rightarrow T_2 \rrbracket^*$ and the fact that $\sigma(v)$ is a value, to prove (G2), we need to show:

(G2') $\langle k, \Psi, \sigma(v)\rangle \in [\![T_1 \rightarrow T_2]\!]$

From the definition of pure, we know $\forall x \in dom(pure(\Gamma)), (pure(\Gamma))(x) = T_{pu}$ for some T_{pu}. Now use the definition of pure again and the Lemma Pure Type, we have $\sigma :_{k,\emptyset} pure(\Gamma)$. Now from the premise and definition of semantic judgments, we have:

(A1) $FV(v) \subseteq dom(pure(\Gamma))$

(A2) $\langle k, \emptyset, \sigma(v)\rangle \in [\![T_1 \Rightarrow T_2]\!]^*$

Now from A2, the definition of $[\![T_1 \Rightarrow T_2]\!]^*$ and the fact that $\sigma(v)$ is a value, we have:

(B) $\langle k, \emptyset, \sigma(v)\rangle \in [\![T_1 \Rightarrow T_2]\!]$

From the definition of $[\![T_1 \Rightarrow T_2]\!]$, we know there exists t such that $\sigma(v) = \lambda x{:}T_1.t$. Suppose $j < k$ and $\langle j, \Psi', v_1\rangle \in [\![T_1]\!]$, by the definition of $[\![T_1 \rightarrow T_2]\!]$, to prove G' we need to show:

(G2") $\langle j, \Psi', t[v_1/x]\rangle \in [\![T_2]\!]^*$

From (B), the definition of $[\![T_1 \Rightarrow T_2]\!]$, $j < k$, $(k, \emptyset) \sqsubseteq (j, \Psi')$ and $\langle j, \Psi', v_1\rangle \in [\![T_1]\!]$, we have exactly G2". And G1 holds trivially from A1. □

Theorem 3.21 (Degeneration). *The following typing rule holds:*

$$\frac{\Gamma \models t : T_1 \rightarrow T_2}{\Gamma \models t : T_1 \Rightarrow T_2} \qquad \text{(T-Degen)}$$

Proof. By the definition of semantic judgments, for any k, σ, Ψ, suppose $\sigma :_{k,\Psi} \Gamma$, then we need to show:

$$\langle k, \Psi, \sigma(t)\rangle \in [\![T_1 \Rightarrow T_2]\!]^*$$

From the premise and definition of semantic judgments, we have $\langle k, \Psi, \sigma(t)\rangle \in [\![T_1 \rightarrow T_2]\!]^*$. The conclusion follows immediately from the lemma Degeneration Closed. □

Lemma 3.22 (Degeneration Value). *If* $\langle k, \Psi_1, v\rangle \in [\![T_1 \rightarrow T_2]\!]$, *then* $\langle k, \Psi_2, v\rangle \in [\![T_1 \Rightarrow T_2]\!]$.

Proof. By the definition of $[\![T_1 \rightarrow T_2]\!]$, we know it must be the case that $v = \lambda x{:}T_1.t$. By the definition of $[\![T_1 \Rightarrow T_2]\!]$, suppose $j < k$, $(k, \Psi_2) \sqsubseteq (j, \Psi')$ and $\langle j, \Psi', v_1\rangle \in [\![T_1]\!]$, we need to prove:

(G) $\langle j, \Psi', t[v_1/x]\rangle \in [\![T_2]\!]$

This is immediately from the definition of $[\![T_1 \rightarrow T_2]\!]$. □

Lemma 3.23 (Degeneration Closed). *If* $\langle k, \Psi, t\rangle \in [\![T_1 \rightarrow T_2]\!]^*$, *then* $\langle k, \Psi, t\rangle \in [\![T_1 \Rightarrow T_2]\!]^*$.

Proof. If t is a value, the result is immediately from the definition of expression typing and the lemma Degeneration Value.

If t is not a value, we need to show that for any $j < k, S, S', t', S :_k \Psi$, if $(S, t) \longrightarrow^j (S', t')$ and $irred(S', t')$, then there exists Ψ' such that the following holds:

(G1) $(k, \Psi) \sqsubseteq (k - j, \Psi')$

(G2) $S' :_{k-j} \Psi'$

(G3) $\langle k - j, \Psi', t'\rangle \in [\![T_1 \Rightarrow T_2]\!]$

Without loss of generality, suppose $S :_k \Psi$ and $(S, t) \longrightarrow^j (S', t') \wedge irred(S', t')$ for $j < k$ steps. From the premises and the definition of $[\![T_1 \rightarrow T_2]\!]^*$, there exists Ψ_1 such that:

(A1) $(k, \Psi) \sqsubseteq (k - j, \Psi_1)$

(A2) $S' :_{k-j} \Psi_1$

(A3) $\langle k - j, \Psi_1, t'\rangle \in [\![T_1 \rightarrow T_2]\!]$

Now choose $\Psi' = \Psi_1$, G1 holds from A1, G2 from A2, G3 from A3 and the lemma Degeneration Value. □

4 PROPERTIES AND EXTENSIONS

In this section, we give an in-depth analysis of *effect polymorphism* and *effect masking* from the perspective of the calculus. We also discuss the extension of the system to support checked exceptions and parametric polymorphism.

4.1 Effect Polymorphism

Recall that the following function `map` has the type signature $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{List}[\mathsf{Int}] \Rightarrow \mathsf{List}[\mathsf{Int}]$:

```
  val map =                           // (Int => Int) -> List[Int] => List[Int]
    (f: Int => Int) => (xs: List[Int]) =>
      xs match {
        case Nil => Nil
        case x :: xs => f(x) :: map(f)(xs)
      }
```

The function `map` is *stoic*, as it does not capture any capabilities or access any non-stoic functions in the outer environment. The inner function is non-stoic, because it captures the non-stoic function `f`. All capabilities in usage during a call of `map` must come from the passed in function `f`. In the language of effects, it means `map` does not produce any effects itself, all effects it produces during the call are produced by the function `f`.

Effect polymorphism is inherently built in capability-based effect systems that support both stoic and free functions. This is because a stoic function like `map` can only indirectly uses whatever capabilities carried along by `f`. The following code snippet shows the usage of `map` with stoic and non-stoic function parameters:

```
val main = (io: IO) => {
  val xs: List[Int] = ...
  map { x => println(x)(io); x * x } xs
  map { x -> x * x } l
}
```

A small caveat is that, when we curry the function `map` with a stoic function, by the typing rule T-App, it can only get the type $\mathsf{Int} \Rightarrow \mathsf{Int}$ instead of $\mathsf{Int} \rightarrow \mathsf{Int}$:

```
val squarePure1 = map { x -> x * x }      // List[Int] => List[Int]
```

A small trick to get back the stoic function is to resort to η-expansion:

```
val squarePure2 =                         // List[Int] -> List[Int]
  (xs: List[Int]) => map { x -> x * x } xs
```

This implies when an expected type is a stoic function type, sometimes we need to do η-expansion. However, usually only higher-order functions expect function values, and higher-order functions like `map` are usually effect-polymorphic, they accept free functions as parameters, no η-expansion is required in such cases. Moreover, when we fully apply a function, effect polymorphism happens implicitly without η-expansion. Thus, we expect the need for η-expansion will be rare in practice.

Note that η-expansion is necessary when the expected type is a stoic function type. This is due to the inability of our system to tell whether `map` has some internal effects or not. It is possible to have a different version of `map` with the same type signature:

```
  val mapE =                           // (Int => Int) -> List[Int] => List[Int]
    (f: Int => Int) => {
      val m = ref Nil
      (xs: List[Int]) => {
        m := !m ++ xs
```

```
      !m
    }
  }
```

Now in the following code, `double` and `doubleEta` will behave differently:

```
val double   =                    // List[Int] => List[Int]
  mapE { (x: Int) => x * x }
val doubleEta = (xs: List[Int]) =>      // List[Int] -> List[Int]
  mapE { (x: Int) => x * x } xs

double(List(1, 2)) == List(1, 2)
double(List(3, 4)) == List(1, 2, 3, 4)
doubleEta(List(1, 2)) == List(1, 2)
doubleEta(List(3, 4)) == List(3, 4)
```

Theoretically, this is not surprising as η-expansion also makes a big difference in traditional type-and-effect systems [Lucassen and Gifford 1988]. This can be demonstrated by the following example:

- $\mathsf{f{:}Int} \xrightarrow{e_1} \mathsf{Int} \xrightarrow{e_2} \mathsf{Int}, \mathsf{x{:}Int} \vdash \mathsf{f\ x} : \mathsf{Int} \xrightarrow{e_2} \mathsf{Int}\ !\ e_1$
- $\mathsf{f{:}Int} \xrightarrow{e_1} \mathsf{Int} \xrightarrow{e_2} \mathsf{Int}, \mathsf{x{:}Int} \vdash \lambda \mathsf{y{:}Int.f\ x\ y} : \mathsf{Int} \xrightarrow{e_1,e_2} \mathsf{Int}\ !\ \mathsf{PURE}$

As we see from above, η-expansion delays the effect e_1. In our case, it ensures that a stoic function indeed does not capture mutable references from its environment: it turns environmental references captured in a non-stoic function into local references of a stoic function.

In the absence of mutational effects, it is possible to prove the following two theorems:

$$\frac{\Gamma \models t_1 : (U \Rightarrow V) \rightarrow T_1 \Rightarrow T_2 \qquad \Gamma \models t_2 : U \rightarrow V}{\Gamma \models t_1\ t_2 : T_1 \rightarrow T_2} \text{ (T-Poly)}$$

$$\frac{\Gamma \models t_1 : T_{pu} \rightarrow T_1 \Rightarrow T_2}{\Gamma \models t_1 : T_{pu} \rightarrow T_1 \rightarrow T_2} \text{ (T-Pure)}$$

The intuition for T-Poly is that in an abstraction $t_1 = \lambda f{:}U \Rightarrow V.\, \lambda y{:}T_1.\, t$ of the type $(U \Rightarrow V) \rightarrow T_1 \Rightarrow T_2$, the nested abstraction of the type $T_1 \Rightarrow T_2$ is typed in a pure context plus f. Therefore it cannot capture any capabilities or free functions, except f. Otherwise, the enclosing abstraction t_1 could not be typed as stoic. Now we know f is instantiated with a stoic function t_2, thus we can also give the inner function a stoic type as well. The intuition for T-Pure is similar: the inner function cannot capture any capabilities nor free functions, thus we can type it as stoic as well.

In the presence of mutational effects, the theorem T-Poly and T-Pure do not hold, as in the outer stoic function, it may create local references that are captured in $T_1 \Rightarrow T_2$. This can be demonstrated by the following stoic function, which has the type $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{Int} \Rightarrow \mathsf{Unit}$:

```
(f: Int => Int) => {
  val m = ref 0
  (x: Int) => m := f(x)          // Int => Unit
}
```

There are subtle differences among the following types:

(1) $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$
(2) $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{List[Int]} \rightarrow \mathsf{List[Int]}$
(3) $(\mathsf{Int} \Rightarrow \mathsf{Int}) \Rightarrow \mathsf{List[Int]} \rightarrow \mathsf{List[Int]}$
(4) $(\mathsf{Int} \Rightarrow \mathsf{Int}) \Rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$
(5) $(\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$

(6) $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{List}[\mathsf{Int}] \to \mathsf{List}[\mathsf{Int}]$
(7) $(\mathsf{Int} \to \mathsf{Int}) \Rightarrow \mathsf{List}[\mathsf{Int}] \to \mathsf{List}[\mathsf{Int}]$
(8) $(\mathsf{Int} \to \mathsf{Int}) \Rightarrow \mathsf{List}[\mathsf{Int}] \Rightarrow \mathsf{List}[\mathsf{Int}]$

Function 1-4 may accept both stoic and free function as parameters, while others only accept stoic function. Function 3-4, 7-8 are non-stoic, so they may capture capabilities from the outer environment, while others not. The inner function of 2-3, 6-7 are pure, while others may be impure. The inner function of 4 and 8 may have arbitrary effects, the inner function of 1 may only have as many effects as the provided function plus read/write local references in its outer function, the inner function of 5 may only read/write local references in its outer function.

Note that for the function type $(\mathsf{Int} \Rightarrow \mathsf{Int}) \to \mathsf{List}[\mathsf{Int}] \Rightarrow \mathsf{List}[\mathsf{Int}]$, we are not sure if the inner function only read/write references in its outer function, whether the first parameter of the type $\mathsf{Int} \Rightarrow \mathsf{Int}$ is actually used or not in the inner function. In this sense, our system is an approximation and is less precise than traditional type-and-effect systems. In type-and-effect systems [Lucassen and Gifford 1988], the effects of all functions are precise, there are no functions with unknown effects like free functions do in our system. By trading precision (but not soundness) for usability, we hope to make effect systems more popular among programmers.

4.2 Mutational Effects and Effect Masking

As our calculus demonstrates, if we take references as capabilities, then we can control mutational effects. The property of non-interference guarantees that during the execution of a stoic function, the function can only read or write memory locations that are explicitly made possible through function parameters.

It is important that when we use the calculus to control mutational effects, we do not generalize the typing rule T-Stoic from value to arbitrary term, i.e. the following typing rule cannot be proved in the system:

$$\frac{\mathsf{pure}(\Gamma) \models \mathsf{t} : \mathsf{T}_1 \Rightarrow \mathsf{T}_2}{\Gamma \models \mathsf{t} : \mathsf{T}_1 \to \mathsf{T}_2} \qquad \text{(T-Stoic')}$$

If we admit such a rule in the type system, we'll be able to type the following term `f` with the stoic type $\mathsf{Int} \to \mathsf{Int}$. Now a stoic function captures references from the environment. It means two different calls to `f` can interfere, it becomes impossible to perform compiler optimizations, like dead code elimination or parallelization, based on stoic function types.

```
val f = {                    // Int => Unit
  val m = ref 0
  (x: Int) => m := x
}
```

A function may locally create new references and mutate them. If they are not observable from outside, those effects can be masked. This is also called *effect masking* in the literature [Lucassen and Gifford 1988].

But how to support effect masking in the effect system? In Lucassen and Gifford [1988], they invented special syntax and typing rules for private regions in order to support masking of local effects. In Koka, the compiler needs to do some proof work to show that a function is fully polymorphic on the heap type `h` in `st<h>` in order to safely mask local mutational effects [Leijn 2017]. This approach corresponds to runST in Haskell [Launchbury and Peyton Jones 1994], its safety is guaranteed by parametricity of the rank-2 polymorphic type:

$$\mathsf{runST} :: (\forall \beta . \mathsf{ST}\ \beta\ \alpha) \to \alpha$$

To write a dummy `increment` operation that uses mutation internally, we have to write following code in Haskell:

```
increment :: Int -> Int
increment x = runST $ do
    ref <- newSTRef x
    modifySTRef ref (+1)
    readSTRef ref
```

In contrast, effect masking is automatically supported in our system: a stoic function can always safely create new memory references and mutate them. As long as the function can be type checked in a pure environment, non-interference of memory effects is guaranteed. Non-observable effects are disregarded automatically by the typing rule T-Stoic. Based on our system, the code looks like the following:

```
val increment = (x: Int) => {            // Int -> Int
  val y = ref x
  y := !y + 1
  !y
}
```

Norman Hardy pointed us to another usage of stoic functions to create a *secret*:

```
val mkSecret = () => {
  val count = ref 0

  val inc = () => count := !count + 1      // Unit => Unit
  val get = () => !count                   // Unit => Int

  (inc, get)
}
```

In the code above, we can think `count` is a secret shared by `inc` and `get`. It is a secret because the only possible way to manipulate it is through `inc` and `get`. The fact that `mkSecret` is a stoic function guarantees that there is an authentic secret. Otherwise, if `count` is declared outside of `mkSecret`, it may be observed and manipulated by other means.

The example above is closely related to the property of non-interference of memory effects. The fact that `mkSecret` does not take any reference as input implies that its local memory region is going to be separated from other memory regions with `inc` and `get` as the only indirect link. The typing rule for T-Stoic guarantees that there is no way for affecting the local memory region except through `inc` and `get`.

4.3 Checked Exceptions

A naive approach to support checking exceptions based on capabilities is to introduce an exception type `Exn` and two primitive functions as follows: [5]

```
try   : (Exn => T, String => T) -> T
throw : String -> Exn -> Bot
```

The function `try` takes two free functions: one is the normal execution code with an exception capability as parameter. The second is the exception handling code with an error message as

---

[5] Strictly speaking, try should have a polymorphic type. But as try needs to be a keyword and deserves a typing rule, we omit the universal type quantifier ∀T to simplify presentation.

parameter. The function `throw` takes an error message and an exception capability, its return type is the bottom type `Bot`.

A benign usage of `try` and `throw` can be demonstrated by the following example:

```
val calc = (io: IO) => (a: Int) => {      // IO -> Int => Int
  try(
    (exn: Exn) => {                       // Exn => Int
      println("start computing...")(io)
      throw("some info")(exn)
    },
    (msg: String) => {                    // String => Int
      println("error found:" + msg)
      0
    }
  )
}
```

In the code above, the calculation throws an exception, the handler prints the error message and returns `0`. The primitive function `try` masks the exception effect with the handler, so that the function `calc` only exposes I/O effects.

It seems that if we prevent programmers from creating an exception capability *ex nihilo*, then we have the guarantee that the only possible way to mask an exception effect is by using `try` or indirectly using an exception capability provided by `try`.

However, this design is unsound. We need to ensure that the exception capability does not *escape* from the scope of `try`. The problem can be demonstrated by the following example:

```
val calc = (io: IO) => (a: Int) => {      // IO -> Int => Int
  val m = ref ((x: Int) => x)             // Ref[Int => Int]
  try(
    (exn: Exn) => {
      m := (x: Int) => throw(exn, "error")
    },
    (msg: String) => {
      println("error found:" + msg)
      unit
    }
  )
  (!m)(3)
}
```

In the code above, we capture the exception capability in a free function and store the function in the mutable cell m. Now the call (!m)(3) will throw exceptions, but the function `calc` does not have exception effects in its type signature!

If we examine the problem more closely, we will find the problem is two-fold:

(1) The exception capability can be returned from `try` as a value or captured in a free function.
(2) Unrestricted capturing of types like `Ref[Int =>Int]` makes it possible to leak the capability.

We fix the two problems respectively:

(1) We require that the return value of `try` does not capture the exception capability.
(2) We only allow capturing variables that cannot leak the capability.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathrm{capture}(\mathrm{Exn}) & = & \mathrm{true} & \mathrm{leaky}(\mathrm{Ref}\ T) & = & \mathrm{capture}(T) \\
\mathrm{capture}(T_1 \Rightarrow T_2) & = & \mathrm{true} & \mathrm{leaky}(T_1 \Rightarrow T_2) & = & \mathrm{capture}(T_1)\ \text{or}\ \mathrm{leaky}(T_2) \\
\mathrm{capture}(\mathrm{Ref}\ T) & = & \mathrm{capture}(T) & \mathrm{leaky}(T) & = & \text{false, otherwise} \\
\mathrm{capture}(T) & = & \text{false, otherwise} & & &
\end{array}
$$

Fig. 3. Definition of *capture* and *leaky*

To control the capturing of variables in the `try` code block, we use the definitions in Figure 3. The function *capture* defines if a value of the type can hold or capture an exception capability value. The function *leaky* defines whether an environment variable of such a type can potentially leak the exception capability.

Now, we can have two more restrictions on the primitive function `try`:

(1) The return type T of `try` cannot be a *capture* type (that is, capture(T) = false).
(2) Only variables of non-leaky type T can be captured inside the code block of `try` (that is, leaky(T) = false).

With the restrictions above, we conjecture the exception capability cannot leak from the `try` block. We leave its formal proof to future work.

4.4 Parametric Polymorphism

To extend the system with parametric polymorphism, we need the following two syntactic typing rules:

$$
\frac{\mathrm{pure}(\Gamma),\ X \vdash t_2 : T}{\Gamma \vdash \lambda X.\, t_2 : \forall X.T}\ (\text{T-TAbs}) \qquad \frac{\Gamma \vdash t_1 : \forall X.T}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T}\ (\text{T-TApp})
$$

Since we restrict in T-TAbs that type abstractions cannot capture any capabilities, we can treat universal types like $\forall X.T$ as pure types. However, for soundness, we need to treat type variables as impure and remove bindings of type variables like $x : X$ from pure environments. This is important to guarantee preservation of the system. This can be seen from the following term t. Without the restriction, it can be typed as $\forall X.X \to \mathrm{Nat} \to X$:

$$t = \lambda X.\, \lambda x{:}X.\, \lambda y{:}\mathrm{Nat}.\, x$$

Now the term $t\ [\mathrm{IO}]$ will have the type $\mathrm{IO} \to \mathrm{Nat} \to \mathrm{IO}$ by T-TApp. However, after one evaluation step, the term $\lambda x{:}\mathrm{IO}.\, \lambda y{:}\mathrm{Nat}.\, x$ has the type $\mathrm{IO} \to \mathrm{Nat} \Rightarrow \mathrm{IO}$, as the capability variable x is captured in the inner lambda; thus preservation breaks.

In practice, we may want to introduce restrictions on type parameters. For example, given the following definition of the parallel map function `pmap`, the system cannot guarantee that all calls to `pmap` are parallelizable, as the stoic function `f` may be impure:

```
def pmap [T, U](f: T -> U)(l: List[T]): List[U] = ...
```

The problem is that the type parameter `T` could be instantiated to a capability type or free function type. For example, consider:

```
pmap { io -> print("hello")(io) } List(system.io)
pmap { f -> f() } List(() => println("hello, world")(system.io))
```

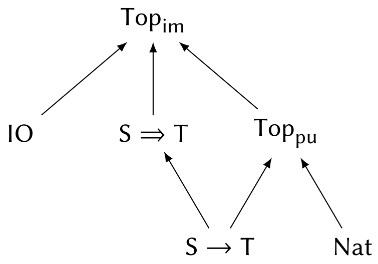

Fig. 4. Subtyping: $\mathsf{Top}_{\mathsf{pu}}$ and $\mathsf{Top}_{\mathsf{im}}$

There are two possibilities. The first is to introduce a type lattice as it is shown in Figure 4, then we can resort to $F_{<:}$. Another way is to introduce kinding on type parameters, to indicate whether the type parameter can be instantiated with an impure type or not.

5 RELATED WORK

In the body of the paper, we already discussed related work about *effect polymorphism* and *effect masking*. We briefly recap them here.

In Haskell, almost every general purpose higher-order function needs both a monadic version and a non-monadic version. As reported by Lippmeier [2010, Section 1.6], Haskell has fractured into monadic and non-monadic sub-languages. Solutions based on parametric polymorphism, such as Koka [Leijn 2017], complicate the syntax and type signature of higher-order functions (though the user is supported by type inference). Our solution to effect polymorphism is supported by the combination of stoic functions and stoic functions, enabling a succint syntax.

In Lucassen and Gifford [1988], special syntax and typing rules for private regions are introduce to support masking of local effects. In Haskell, effect masking is supported by the ST monads and runST [Launchbury and Peyton Jones 1994], the safety is guaranteed by parametricity of the rank-2 polymorphic type:

$$\mathsf{runST} :: (\forall \beta . \mathsf{ST}\ \beta\ \alpha) \to \alpha$$

However, this approach is heavy in syntax. Koka improves its usability by moving the burden of proof from programmers to the compiler: the compiler needs to do some proof work to show that a function is fully polymorphic on the heap type `h` in `st<h>` in order to safely mask local mutational effects [Leijn 2017]. In our system, effect masking is supported automatically without any special syntax or typing rule.

Note that the usability of our system is achieved by sacrificing some precision but not soundness of the effect system. For example, the function `map` will take the following type in a type-and-effect system [Lucassen and Gifford 1988]:

$$(\mathsf{Int} \xrightarrow{\sigma} \mathsf{Int}) \to \mathsf{List[Int]} \xrightarrow{\sigma} \mathsf{List[Int]}$$

From the type signature, it is obvious that the first parameter $(\mathsf{Int} \xrightarrow{\sigma} \mathsf{Int})$ is not used in the outer function, as its effect is empty. In our system, the same function `map` takes the type $(\mathsf{Int} \Rightarrow \mathsf{Int}) \to \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$. We only know that the inner function can produce as many effects as the first parameter plus possible effects on locally allocated memory cells inside the outer function. By trading precision (but not soundness) for usability, we hope to make effect systems more popular among programmers.

Meanwhile, as we have shown in 4.2, our system also supports compiler optimizations, as a stoic function is pure if its parameter and return type are pure.

5.1 Capabilities

There has been a long history in using capabilities in computer systems for security. For example, KeyKos [Hardy 1985] is the first operating system to implement confinement based on capabilities. Mullender and Tanenbaum [1986] uses capabilities in the design of distributed operating systems. The recent verified secure kernel seL4 [Elkaduwe et al. 2008; Klein et al. 2009] is also designed around capabilities.

Dennis and Van Horn [1966] propose the *object-capability model* as a conceptual framework of capability systems, and Miller [2006] refines the model. Several programming languages are implemented based on the model, such as E, Joule and Pony [Agorics 1995; Clebsch et al. 2015; Miller 1997]. And there are some verification efforts for object-capabilities, like [Devriese et al. 2016; Murray 2010; Swasey et al. 2017].

A major difference between our work and this line of research is that capabilities in our system are controlled by a static type system, while the others depend on clever design patterns.

5.2 Checking Effects

Gifford and Lucassen [1986]; Lucassen and Gifford [1988] first introduced type-and-effect systems and effect polymorphism using effect type parameterization. In the same work, they also introduced the concept *effect masking* for memory effects.

Moggi [1991] introduced monads for giving semantics to computational effects. Wadler and Thiemann [2003] showed that it is possible to transpose any type-and-effect system into a corresponding system for checking effects based on monads. The work on algebraic effects [Bauer and Pretnar 2015; Kammar et al. 2013; Plotkin and Pretnar 2009] provides a different approach to give semantics to (user-defined) effects. Algebraic effects may also be equipped with a type system for checking effects [Leijen 2017]. Our work focuses on checking effects instead of giving semantics to effects, thus it is closer to Wadler and Thiemann [2003].

Osvald et al. [2016] introduced second-class citizenship. Second-class citizens observe stack discipline, they cannot be leaked into the heap after the function call finishes. They capitalize on the idea *effects as capabilities* and *capabilities as 2nd-class citizens* to implement an effect system for Scala. The type system will ensure the usage of capabilities observes stack discipline by checking that a first-class function does not capture capabilities. However, the system restricts that the return value of a function must be first-class. This is an obstacle to use the system to control mutational effects, as heap references may not be returned from functions.

Miller et al. [2014] introduced *spores*, which enable programmers to control what types of values can or cannot be captured inside a closure. The abstraction is primarily motivated for safe concurrent and distributed computing. For example, it can ensure that the closures shared between two machines are serializable and there is no accidental capturing of non-serializable values from the environment. Spores have more refined control on the capturing behaviors of closures, while stoic functions can only be used to control the capturing of capabilities. Due to this restriction, stoic functions are conceptually simpler and syntactically more succinct. We believe the two have different usage: spores are more suitable for distributed and concurrent computing, while stoic functions fit better for capability-based systems.

Marino and Millstein [2009] proposed a general effect system based on the idea *effect systems as privilege checking*. For the example of checked exceptions, a `try` block grants the privilege `canThrow` to the body of `try`, while a `throw` clause involves checking the privilege. The idea is in the same spirit as *effects as capabilities*. They impose a set of monotonicity requirements on the externally provided privilege discipline to guarantee type soundness. The proposed framework is more general than ours in that it can be instantiated to control memory effects, ensure strong atomicity for

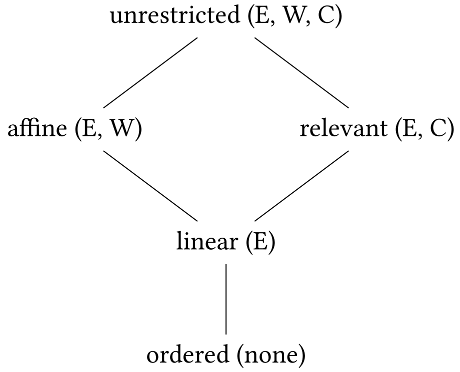

Fig. 5. Substructural type systems, exchange (E), weakening (W), contraction (C)

software transactional memory and etc. However, they do not propose a concept like *stoic* as we do. Our work is more specific, and it covers more concrete topics like effect polymorphism and effect masking.

5.3 Substructural types

The two function types in our system are reminiscent of two function types in one style of linear type systems [Mazurak et al. 2010; Morris 2016; Wadler 1990]. In such linear type systems, there exists two function types: linear function type and unrestricted function type. Unrestricted function types exhibit similar capturing behaviors as stoic functions. For example, unrestricted functions cannot capture linear functions nor variables of linear types. Besides the similarity in capturing control, our system does not have restrictions on substructural properties, thus it is not a substructural type system.

More generally, depending on whether the substructural properties hold or not in the type system, type systems can be classified as in Figure 5 [Pierce 2005].

Our work shows that in the area *unrestricted*, there is an interesting system which exhibits similar capturing behaviors as substructural type systems. This similarity is not superficial. For example, the work by Morris [2016] is an important inspiration for us in our on-going work in developing a type inference algorithm.

6 CONCLUSION

We propose *stoic function* as a useful abstraction for capability systems. We formalize the concept in STLC with mutation, taking heap references as capabilities. We prove that stoic functions in that setting enjoy non-interference of memory effects.

We show that our system supports a common form of effect-polymorphism without introducing effect variables. The capability-way of thinking is what programmers already familiar with in daily life, thus the cognitive load is low. The ability to embed non-stoic functions inside stoic functions reduces the syntactic overhead to be only at the interface. Also, combining multiple effects is easy as capabilities combine easily. Effect masking is supported automatically without any special syntax nor typing rule. The effect system can be adopted incrementally. The benefits of usability are achieved by sacrificing a little precision but not soundness of the effect systems (Section 4.1).

Future work. Norman Hardy, the designer of KeyKos, pointed to us the potential usage of stoic functions to solve the *confinement problem* [Lampson 1973]. We want to explore the application of stoic functions in safe and efficient operating systems without segregation based on virtual memory, following the formal approach to OS design as in Hunt and Larus [2007]. The work in this paper is based on the approach of foundational proof-carrying code [Appel and McAllester 2001], which makes the application in security promising.

Second, we are motivated to implement an effect system for Standard ML or OCaml. One immediate challenge is to develop a type inference algorithm. The work by Morris [2016] is an inspiration for us.

Third, we are working on an effect system for object-oriented languages that can control mutational effects, non-determinism and input/output. There are more challenges in the OO setting, including the support for read-only references, transitive object immutability, immutability polymorphism, inheritance, interfaces, inner classes and so on.

REFERENCES

Inc. Agorics. 1995. Joule: Distributed Application Foundations. (Dec. 1995). Retrieved February 8, 2017 from http://www.erights.org/history/joule/index.html

Amal Ahmed, Andrew W. Appel, and Roberto Virga. 2003. An indexed model of impredicative polymorphism and mutable references. (2003).

Amal Jamil Ahmed. 2004. *Semantics of types for mutable state.* Ph.D. Dissertation. Princeton University.

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

Sylvan Clebsch, Sebastian Blessing, Sophia Drossopoulou, and Andrew McNeil. 2015. The Pony Language. (2015). Retrieved February 8, 2017 from https://github.com/ponylang/ponyc

Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about object capabilities with logical relations and effect parametricity. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on.* IEEE, 147–162.

Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. 2008. Verified protection model of the seL4 microkernel. In *Working Conference on Verified Software: Theories, Tools, and Experiments.* Springer, 99–114.

David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* ACM, 28–38.

Norman Hardy. 1985. KeyKOS architecture. *ACM SIGOPS Operating Systems Review* 19, 4 (1985), 8–25.

Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. *Operating Systems Review* 41, 2 (2007), 37–49. https://doi.org/10.1145/1243418.1243424

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 145–158.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.* ACM, 207–220.

Butler W. Lampson. 1973. A note on the confinement problem. *Commun. ACM* 16, 10 (1973), 613–615.

John Launchbury and Simon L. Peyton Jones. 1994. Lazy functional state threads. In *ACM SIGPLAN Notices*, Vol. 29. ACM, 24–35.

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 486–499. http://dl.acm.org/citation.cfm?id=3009872

Daan Leijn. 2017. The Koka Book. (2017). Retrieved February 4, 2018 from https://koka-lang.github.io/koka/doc/kokaspec.html

Ben Lippmeier. 2010. *Type inference and optimisation for an impure world.* Ph.D. Dissertation. Australian National University.

John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 47–57.

Daniel Marino and Todd Millstein. 2009. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation.* ACM, 39–50.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in System F. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in Language Design and Implementation.* ACM, 77–88.

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: a type-based foundation for closures in the age of concurrency and distribution. In *ECOOP 2014–Object-Oriented Programming*. Springer, 308–333.

Mark S. Miller. 1997. The E Language. (1997). Retrieved February 8, 2017 from http://www.erights.org/elang/index.html

Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.

J. Garrett Morris. 2016. The best of both worlds: linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* 448–461. https://doi.org/10.1145/2951913.2951925

Sape J. Mullender and Andrew S. Tanenbaum. 1986. The design of a capability-based distributed operating system. *Comput. J.* 29, 4 (1986), 289–299. https://doi.org/10.1093/comjnl/29.4.289

Toby C. Murray. 2010. *Analysing the security properties of object-capability patterns.* Ph.D. Dissertation. University of Oxford, UK.

Martin Odersky. 2015. Scala — where it came from, where it is going. (2015). Retrieved February 8, 2017 from http://www.slideshare.net/Odersky/scala-days-san-francisco-45917092

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: foundations and applications of implicit function types. *PACMPL* 2, POPL (2018), 42:1–42:29. https://doi.org/10.1145/3158130

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 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 234–251.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Benjamin C. Pierce. 2005. *Advanced topics in types and programming languages.* MIT Press.

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

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 89.

Philip Wadler. 1990. Linear types can change the world. In *IFIP TC*, Vol. 2. 347–359.

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)* 4, 1 (2003), 1–32.