Stoic: Towards Disciplined Capabilities

Fengyun Liu
EFPL

Sandro Stucki
Chalmers University of Technology

Nada Admin
Havard University

Paolo G. Giarrusso
Delft University of Technology

Martin Odersky
EPFL

Abstract

Capabilities are widely used in the design of software systems to ensure security. A system of capabilities can become a mess in the presence of objects and functions: objects may leak capabilities and functions may capture capabilities. They make reasoning and enforcing invariants in capability-based systems challenging if not intractable.

How to reason about capability-based systems formally? What abstractions that programming languages should provide to facilitate the construction of capability-based systems? Can we formulate some fundamental capability disciplines as typing rules?

In this paper we propose that *stoicity* is a useful property in designing, reasoning and organizing capabilities in systems both at the macro-level and micro-level. Stoicity means that a component of a system does not interact with its environment in any way except through its interfaces.

As an incarnation of this idea, we introduce *stoic functions* in a functional language. In contrast to normal functions, stoic functions cannot capture capabilities nor non-stoic functions from the environment. We formalize stoic functions in a language with mutable references as capabilities. In that setting, we show that stoic functions enjoy *non-interference of memory effects*. The concept of stoic functions also shows its advantage in *effect polymorphism* and *effect masking* when used to control side effects of programs.

2012 ACM Subject Classification Software and its engineering → Functional languages; Software and its engineering → Procedures, functions and subroutines

Keywords and phrases stoic function, effect polymorphism, effect masking, capability

Acknowledgements We thank the anonymous reviewers of ICFP 2017, ICFP 2018, ECOOP 2019 for their constructive comments. We gratefully acknowledge funding by the Swiss National Science Foundation under Grant 200021_166154 (Effects as Implicit Capabilities).

1 Introduction

Capabilities are widely used constructs in the human world, examples include keys, passwords, bank notes, credit cards, etc. Like their real-world counterparts, capabilities are used in computing to control access to resources in the area of operating system security [14, 29, 38, 41, 11] and memory management [7, 12].

However, *contemporary programming languages are not friendly for constructing capability-based systems*. In functional programming languages, functions are typically allowed to close over arbitrary values from the environment. This includes references to resources such as file handles and mutable state, as well as other closures referring to yet more of the environment. Closures thus pose a challenge if one is to track and control the flow of capabilities through a program. Meanwhile, in object-oriented programming, a class may mutate global state, access file systems, store and reuse capabilities that are supposed to be used once, or leak capabilities to untrusted third-party components, etc.

All these irregularities make it difficult to reason and enforce invariants of capability-based systems. In a closed-source system, programmers can resort to good programming practices to ensure capability invariants of a system. However, in the case of open platforms with components from potentially untrusted third-parties, e.g. JavaScript code from different sources on the same web page, or a tenant-based cloud service that enable customers to run customized code on a shared system or a plugin system for web browser, there is a huge security concern that current programming languages fall short to address.

How to reason capability-based systems formally? What abstractions that programming languages should provide to facilitate the construction of capability-based systems? Can we formulate some fundamental capability disciplines as typing rules?

In this paper we propose that *stoicity* is a useful property in designing, reasoning and organizing capabilities in systems both at the macro-level and micro-level. Stoicity means that a component of a system does not use any capabilities directly or indirectly from its environment in any way except those provided explicitly through its interfaces. If we think interfaces as front door of interaction, and other means of interaction with environment as backdoor, then stoicity means there should be no backdoor for capabilities.

As an incarnation of this idea, we introduce a simple abstraction for tracking and controlling the flow of capabilities in a functional language: *stoic functions*. In contrast to non-stoic functions, which can freely capture capabilities from the environment, stoic functions are more disciplined: *they may only use capabilities or non-stoic functions provided to them explicitly as function arguments; they never capture capabilities or non-stoic functions from the environment*. All stoic functions are supposed to observe this capability discipline.

We formalize stoic functions in a language with mutable references as capabilities. We prove that stoic functions enjoy *non-interference of memory effects* in a step-indexed model. This makes stoic functions a natural foundation to provide controlled isolation between components. The concept of stoic functions also shows its advantage when capabilities are used to control side effects of programs. The system supports *effect polymorphism* with succinct syntax. Also, *effect masking* for local mutations works automatically without any special syntax or typing rule.

The contributions of this paper are the following:

1. We identify *stoicity* as a useful property in reasoning and designing capability-based systems and propose *stoic functions* as its incarnation for controlling and reasoning about capabilities in functional languages (Section 2).
2. We formalize stoic functions in λ^{cap}, an extension of STLC with stoic functions and mutations. We prove that stoic functions enjoy *non-interference* of memory effects based on step-indexed models (Section 3).
3. We demonstrate that capability-aware programming languages support a common form of *effect polymorphism* with succinct syntax. Also, *effect masking* for local mutations works automatically without any special syntax or typing rule (Section 4).

2 Capabilities, Security and Effects

Capabilities are unforgeable values that can be used to activate some sensitive operations in a capability-based system. To defend against abuse, sensitive operations are usually protected by capabilities. In security, what is of interest are in fact the consequences of the sensitive operations, or potential effects that are enabled by capabilities. To control security with capabilities is in essence the same as to control effects with capabilities. Security and effects are two sides of the same coin.

Capabilities are always defined with respect to the operations enabled by them. It does not make sense to talk about *what is a capability* without referring to the corresponding operations, just like it does not make sense to talk about keys without mentioning the locks that they match. A key is defined by the lock(s) that it can open, a key that cannot open any locks is not a key. The same holds true for capabilities.

In a functional language, usually capabilities and the operations enabled by the capabilities are separate. For example, a file handle and the functions for file manipulations such as read/write are loosely connected. We can also think of a function value as a capability if the value is a reference to an effectful operation. In object-oriented languages, capabilities and operations enabled by the capabilities are usually coupled in the same class. For example, a reference of the type File is the capability, the operations are methods of the corresponding object.

While what operations count as sensitive or effectful may differ from system to system, unrestricted access to file systems or network are almost always regarded as effectful as they pose a huge threat to security. Meanwhile, local mutations are usually regarded as harmless, thus they may be masked [17, 22], in contrast to global or environmental mutations. In a specific system, programmers may define database connections or a HTTP connection as a capability.

The idea of controlling effects with capabilities is not new [34, 31, 23]: *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 effects:

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

However, as mentioned in the introduction, contemporary programming languages lack mechanisms to prevent abuses of capabilities: closures may capture capabilities and classes may leak capabilities. Both pose a challenge in tracking the usage and flow of capabilities. The main contribution of our work is to identify *stoicity* as a fundamental discipline for programming with capabilities.

To reiterate, stoicity means that a component of a system does not use any capabilities directly or indirectly from its environment in any way except those provided explicitly through its interfaces. If we think of interfaces as front door of interaction, and other means of interaction with environment as backdoor, then stoicity means there should be no backdoor for capabilities.

As an incarnation of this idea, we introduce a simple abstraction for tracking and controlling the flow of capabilities in a functional language: *stoic functions*. In contrast to non-stoic functions, which can freely capture capabilities from the environment, stoic

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

functions are more disciplined: *they may only use capabilities or non-stoic functions provided to them explicitly as function arguments; they never capture capabilities or non-stoic functions from the environment.* All stoic functions are supposed to observe this capability discipline.

2.1 Stoic and Free Functions

We call non-stoic functions *free functions*, which can freely capture capabilities or non-stoic functions from its environment. In contrast, stoic functions are more disciplined about its potential effects – it only uses capabilities or non-stoic functions provided to them explicitly. Stoic functions do not have backdoor for capabilities. 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)    // 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`. 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 `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 (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 $(\mathsf{Int} \Rightarrow \mathsf{Int}) \Rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$, 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 [22], 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 [32], 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)
```

The inner lambda is typed as free, because it captures the first parameter $\mathsf{r} : \mathsf{Ref[Int]}$.

2.5 Flexible Adoption

The capability discipline is like an armor that protects programmers from tricky bugs caused by abuse of capabilities. 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 disregard the capability discipline completely. In a larger project, the choice of which components should be capability-disciplined may evolve over time. Moreover, a component as a whole may behave as capability-disciplined, but internally it may want to loosen such discipline.

With both stoic and free functions, capability-aware languages support *flexible adoption* of capability discipline. If programmers decide to disregard capabilities, they can just use free functions throughout in the program. During software development, if programmers want to make more components capability-disciplined, it suffices to change some free functions to stoic functions and making their effects explicit. We believe enabling programmers to flexibly mix capability-disciplined code with non-disciplined code without losing safety is another key factor for the practicality of a capability-aware language.

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 [3].

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 ($\mathsf{T} \rightarrow \mathsf{T}$).

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 makes the semantics of stoic functions explicit (with respect to stores), thus is preferred in this work.

The most important change in typing rules is the introduction of the typing rule T-Stoic, which assigns types 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 4.3.

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

3.2 Semantic Typing

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

To prove soundness of the system, we follow the step-indexed approach as demonstrated in [3]. 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 $[\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]$ is satisfied by $\langle \Psi, \mathsf{v} \rangle$ if the value v, when run in a store matching store typing Ψ, runs *safely* (without getting stuck) and maps argument values in $[\![\mathsf{T_1}]\!]$ to values in $[\![\mathsf{T_2}]\!]$.

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

Syntax

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

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

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

$$\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']}\ \text{(E-Context)}$$

$$(\lambda x{:}T.\, t_1)\ v_2 \longrightarrow [x \mapsto v_2]t_1\ \text{(E-Beta)}$$

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

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

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

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

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

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\, t_2 : T_1 \Rightarrow T_2}\ \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \Rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ \text{(T-App)}$$

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

$$\frac{\Gamma \vdash t_1 : \mathsf{Ref}\ T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 := t_2 : \mathsf{Unit}}\ \text{(T-Assign)}$$

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

$$\frac{pure(\Gamma) \vdash v : T_1 \Rightarrow T_2}{\Gamma \vdash v : T_1 \rightarrow T_2}\ \text{(T-Stoic)}$$

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

Pure Environment

$$pure(\varnothing) = \varnothing$$
$$pure(\Gamma, x{:}T_{im}) = pure(\Gamma)$$
$$pure(\Gamma, x{:}T_{pu}) = pure(\Gamma), x{:}T_{pu}$$

Figure 1 Syntax and Syntactic Typing for λ^{cap}

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 $\langle \mathsf{k}, \Psi, \mathsf{v} \rangle$. Roughly speaking, $\langle \mathsf{k}, \Psi, \mathsf{v} \rangle \in [\![\mathsf{T}]\!]$ 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 $[\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]$, then v must be a function value, and the result of applying this function value to an input in $[\![\mathsf{T_1}]\!]$ must satisfy $[\![\mathsf{T_2}]\!]$ (up to a certain number of steps).

A key insight on the connection between step-indexed models and capabilities, alluded in the footnote of [3, P. 55], is that the store typing Ψ in the tuple $\langle \mathsf{k}, \Psi, \mathsf{t} \rangle$ 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 [3] below.[2] As a convention, we write $\langle \mathsf{k}, \Psi, \mathsf{t} \rangle$ as a short-hand for $\langle \mathsf{k}, \lfloor \Psi \rfloor_{\mathsf{k}}, \mathsf{t} \rangle$ to simplify the presentation.

3.2.1 Basic Definitions

▶ **Definition 1** (Safe). *A state* (S, t) *is safe for* k *steps if for any reduction* $(\mathsf{S}, \mathsf{t}) \longrightarrow^{\mathsf{j}} (\mathsf{S}', \mathsf{t}')$ *of* $\mathsf{j} < \mathsf{k}$ *steps, either* t' *is a value or another step is possible.*

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

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

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

▶ **Definition 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:*

$$\begin{array}{lcl} \lfloor \tau \rfloor_{\mathsf{k}} & \triangleq & \{ \langle \mathsf{j}, \Psi, \mathsf{v} \rangle \mid \mathsf{j} < \mathsf{k} \land \langle \mathsf{j}, \Psi, \mathsf{v} \rangle \in \tau \} \\ \lfloor \Psi \rfloor_{\mathsf{k}} & \triangleq & \{ (\mathsf{l} \mapsto \lfloor \tau \rfloor_{\mathsf{k}}) \mid \Psi(\mathsf{l}) = \tau \} \end{array}$$

▶ **Definition 3** (State Extension). *A valid state extension is defined as follows:*

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

▶ **Definition 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,*

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

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:

[2] With minor adaptations. For example, *extensible* is called *type* in [3].

$$
\begin{array}{lll}
[\![\mathsf{Nat}]\!] & \triangleq & \{\ \langle k, \Psi, n\rangle\ \} \\[1ex]
[\![\mathsf{Unit}]\!] & \triangleq & \{\ \langle k, \Psi, \mathsf{unit}\rangle\ \} \\[1ex]
[\![T_1 \Rightarrow T_2]\!] & \triangleq & \{\ \langle k, \Psi, \lambda x{:}T_1.t\rangle \mid \forall v, \Psi', j < k. \\
 & & \quad ((k, \Psi) \sqsubseteq (j, \Psi') \wedge \langle j, \Psi', v\rangle \in [\![T_1]\!]) \Longrightarrow \langle j, \Psi', t[v/x]\rangle \in [\![T_2]\!]^*\ \} \\[1ex]
[\![T_1 \rightarrow T_2]\!] & \triangleq & \{\ \langle k, \Psi, \lambda x{:}T_1.t\rangle \mid \forall v, \Psi', j < k. \\
 & & \quad \langle j, \Psi', v\rangle \in [\![T_1]\!] \Longrightarrow \langle j, \Psi', t[v/x]\rangle \in [\![T_2]\!]^*\ \} \\[1ex]
[\![\mathsf{Ref}\ T]\!] & \triangleq & \{\ \langle k, \Psi, l\rangle \mid \lfloor \Psi \rfloor_k(l) = \lfloor [\![T]\!] \rfloor_k\ \} \\[1ex]
[\![T]\!]^* & \triangleq & \{\ \langle k, \Psi, t\rangle \mid \mathsf{val}(t) \wedge \langle k, \Psi, t\rangle \in [\![T]\!]\ \bigvee \\
 & & \quad \neg \mathsf{val}(t) \wedge \forall j, S, S', t'. \\
 & & \quad (j < k \wedge S :_k \Psi \wedge (S, t) \longrightarrow^j (S', t') \wedge \mathsf{irred}(S', t')) \\
 & & \quad \Longrightarrow \exists \Psi'.(k, \Psi) \sqsubseteq (k-j, \Psi') \wedge S' :_{k-j} \Psi' \wedge \\
 & & \qquad \langle k-j, \Psi', t'\rangle \in [\![T]\!] \qquad \}
\end{array}
$$

Figure 2 Semantic Typing for λ^{cap}

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

▶ **Definition 5** (Well-typed Store). *A store* S *is well-typed to approximation* k *with respect to a store typing* Ψ *iff* $\mathsf{dom}(\Psi) \subseteq \mathsf{dom}(S)$ *and the contents of each location* $l \in \mathsf{dom}(\Psi)$ *has type* $\Psi(l)$ *to approximation* k*:*

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

3.2.2 Interpretation of Types

The interpretation of syntactic types are given in Figure 2. $[\![T]\!]$ defines what it means for a value to belong to a type, and $[\![T]\!]^*$ 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 $T_1 \Rightarrow T_2$ must map argument values in T_1 to result expressions in T_2. More precisely, $\langle k, \Psi, \lambda x{:}T_1.t\rangle \in [\![T_1 \Rightarrow T_2]\!]$ requires that function body t satisfies $[\![T_2]\!]$ for at least $j < k$ steps when applied to an argument that satisfies $[\![T_1]\!]$ for j steps. Moreover, a value of the free function type $T_1 \Rightarrow T_2$ may capture references from the environment, and can assume the references in Ψ are available; so we only constrain the behavior of the body t for stores satisfying store typings Ψ' that extend Ψ. As Ψ' could be Ψ and j could be $k-1$, the store typing Ψ must at least contain the necessary capabilities for the function body t to take $k-1$ steps.

The interpretation for $\mathsf{T}_1 \to \mathsf{T}_2$ 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 $\mathsf{S}(\mathsf{l})$ 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 made in [3] over [2]. 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 $\mathsf{l} \in \mathsf{dom}(\Psi)$, $\mathsf{S}(\mathsf{l})$ can safely take j steps with $\lfloor \Psi \rfloor_{\mathsf{j}}$ and S. [3] shows that one can remove S from the quadruple and simplify the definition of $[\![\mathsf{Ref}\ \mathsf{T}]\!]$ with an additional definition of *well-typed store* and impose that condition in expression typings, i.e. well-formed store will remain 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,[3] then $\langle \mathsf{k}, \Psi, \mathsf{t}\rangle \in [\![\mathsf{T}]\!]^*$ is equivalent to $\langle \mathsf{k}, \Psi, \mathsf{t}\rangle \in [\![\mathsf{T}]\!]$. Otherwise, given a well-typed store S with respect to Ψ, if (S, t) reduces to an irreducible state $(\mathsf{S}', \mathsf{t}')$ in j steps for any $\mathsf{j} < \mathsf{k}$, 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 $\mathsf{k} - \mathsf{j}$ steps with the capabilities Ψ'.

▶ **Definition 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:

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

$$\models \mathsf{t} : \mathsf{T} \triangleq \emptyset \models \mathsf{t} : \mathsf{T}$$

3.2.3 Soundness

▶ **Theorem 7** (Soundness). *If $\models \mathsf{t} : \mathsf{T}$, 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 $\langle \mathsf{k}', \Psi, \mathsf{t}\rangle \in [\![\mathsf{T}]\!]^*$. In particular, it holds for $\Psi = \emptyset$ and $\mathsf{k}' = \mathsf{k}$. It is obvious that $\mathsf{S} :_{\mathsf{k}} \Psi$.

From the definition of $[\![\mathsf{T}]\!]^*$, 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 $(\mathsf{S}, \mathsf{t}) \longrightarrow^{\mathsf{j}} (\mathsf{S}', \mathsf{t}')$ for $\mathsf{j} < \mathsf{k}$ steps, and there exists some Ψ' such that $(\mathsf{k} - \mathsf{j}, \Psi', \mathsf{t}') \in [\![\mathsf{T}]\!]$. From the definition of $[\![\mathsf{T}]\!]$, we know t' must be a value, thus (S, t) is safe for k steps. ◀

▶ **Definition 8** (Non-interference). *A term t of type T is non-interferent with the store typing Ψ_1, written as $\mathsf{t} : \mathsf{T} \# \Psi_1$, if and only if for any k, there exists Ψ_2 with $\mathsf{dom}(\Psi_1) \cap \mathsf{dom}(\Psi_2) = \emptyset$ such that $\langle \mathsf{k}, \Psi_2, \mathsf{t}\rangle \in [\![\mathsf{T}]\!]^*$.*

---

[3] 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 $\langle \mathsf{k}, \Psi, \mathsf{v}\rangle \in [\![\mathsf{T}]\!]^*$ then $\langle \mathsf{k}, \Psi, \mathsf{v}\rangle \in [\![\mathsf{T}]\!]$ with the same Ψ, not just with some extension of Ψ.

The definition depends on the following observation: if $\langle \mathsf{k}, \Psi, \mathsf{t} \rangle \in [\![\mathsf{T}]\!]^*$, 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 definitions of $[\![\mathsf{T}_1 \to \mathsf{T}_2]\!]$ and $[\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]$ also ensure 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 location 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 the calculus, the function f will be typed as $\mathsf{Int} \Rightarrow \mathsf{Ref\ Int}$. Rejecting the function as stoic is important, otherwise it could be used as a secret channel for leaking sensitive information. A stoic function should only use explicitly provided memory locations or create new memory locations, but not secretly capture memory locations.

To simplify the presentation, we also use the following definitions:

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

▶ **Theorem 9** (Non-interference). *If* $\Gamma \models \lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t} : \mathsf{T}_1 \to \mathsf{T}_2$, $\forall \Psi, \sigma, \mathsf{v}, \Psi_1$, *if* $\sigma :_\Psi \Gamma$ *and* $\mathsf{v} :_{\Psi_1} \mathsf{T}_1$, *we have* $\sigma(\mathsf{t})[\mathsf{v}/\mathsf{x}] : \mathsf{T}_2 \mathbin{\#} \Psi - \Psi_1$.

Proof. By the definition of non-interference, we need to prove that for any step-index k, there exists Ψ' such that $\mathsf{dom}(\Psi - \Psi_1) \cap \mathsf{dom}(\Psi') = \emptyset$ and $\langle \mathsf{k}, \Psi', \sigma(\mathsf{t})[\mathsf{v}/\mathsf{x}] \rangle \in [\![\mathsf{T}_2]\!]^*$.

We choose $\Psi' = \Psi_1$, it is obvious that $\mathsf{dom}(\Psi - \Psi_1) \cap \mathsf{dom}(\Psi_1) = \emptyset$, by the definition of store typing subtraction. Without loss of generality, let's choose some $\mathsf{m} > \mathsf{k}$, from the definition of semantic judgments and the fact that $\sigma(\lambda \mathsf{x}{:}\ \mathsf{T}_1.\mathsf{t})$ is a value, we have the following:

$$\langle \mathsf{m}, \Psi, \sigma(\lambda \mathsf{x}{:}\ \mathsf{T}_1.\mathsf{t}) \rangle \in [\![\mathsf{T}_1 \to \mathsf{T}_2]\!]$$

Now by the definition of $[\![\mathsf{T}_1 \to \mathsf{T}_2]\!]$ and $\langle \mathsf{m}, \mathsf{v}, \Psi_1 \rangle \in [\![\mathsf{T}_1]\!]$, we have $\forall \mathsf{j} < \mathsf{m}, \langle \mathsf{j}, \Psi_1, \sigma(\mathsf{t})[\mathsf{v}/\mathsf{x}] \rangle \in [\![\mathsf{T}_2]\!]^*$. In particular, it holds for k, as we know $\mathsf{k} < \mathsf{m}$. ◀

This theorem says that if a 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 other words, 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 [3].

▶ **Lemma 10** (State Extension Reflexive). $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k}, \Psi)$.

▶ **Lemma 11** (State Extension Transitive). *If* $(\mathsf{k}_1, \Psi_1) \sqsubseteq (\mathsf{k}_2, \Psi_2)$ *and* $(\mathsf{k}_2, \Psi_2) \sqsubseteq (\mathsf{k}_3, \Psi_3)$, *then* $(\mathsf{k}_1, \Psi_1) \sqsubseteq (\mathsf{k}_3, \Psi_3)$.

▶ **Lemma 12** (Type Set Extensible). *For any syntactic type* T, $\mathsf{extensible}(\llbracket \mathsf{T} \rrbracket)$.

▶ **Lemma 13** (Extensibility Weakening). *If* $\mathsf{extensible}(\lfloor \tau \rfloor_\mathsf{k})$ *and* $\mathsf{j} \le \mathsf{k}$ *then* $\mathsf{extensible}(\lfloor \tau \rfloor_\mathsf{j})$.

▶ **Lemma 14** (Index Cut). *If* $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{j}, \Psi')$, $\mathsf{i} < \mathsf{k}$, *and* $\mathsf{i} < \mathsf{j}$, *then* $(\mathsf{i}, \lfloor \Psi \rfloor_\mathsf{i}) \sqsubseteq (\mathsf{i}, \lfloor \Psi' \rfloor_\mathsf{i})$.

▶ **Lemma 15** (Index Weakening). *If* $\mathsf{j} < \mathsf{k}$, *then* $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{j}, \Psi)$.

▶ **Lemma 16** (Determinism of Evaluation). *If* $(\mathsf{S}, \mathsf{t}) \longrightarrow^\mathsf{i} (\mathsf{S}_1, \mathsf{t}_1) \wedge \mathsf{irred}(\mathsf{S}_1, \mathsf{t}_1)$ *and* $(\mathsf{S}, \mathsf{t}) \longrightarrow^\mathsf{j} (\mathsf{S}_2, \mathsf{t}_2) \wedge \mathsf{irred}(\mathsf{S}_2, \mathsf{t}_2)$, *then* $\mathsf{S}_1 = \mathsf{S}_2$, $\mathsf{t}_1 = \mathsf{t}_2$ *and* $\mathsf{i} = \mathsf{j}$.

▶ **Lemma 17** (Store Index Weakening). *If* $\mathsf{S} :_\mathsf{k} \Psi$ *and* $\mathsf{j} < \mathsf{k}$, *then* $\mathsf{S} :_\mathsf{j} \Psi$.

3.4 Proof of Typing Rules

To relate our syntactic type judgement $\Gamma \vdash \mathsf{t} : \mathsf{T}$ with semantic typing, we must prove that our syntactic typing rules are *sound* relative to semantic typing, as stated in the following theorem.

▶ **Theorem 18** (Soundness of Syntactic Typing). *If* $\Gamma \vdash \mathsf{t} : \mathsf{T}$ *then* $\Gamma \models \mathsf{t} : \mathsf{T}$.

This theorem is proven by induction on derivations of $\Gamma \vdash \mathsf{t} : \mathsf{T}$. 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 and are thus omitted. Only the proofs for T-Stoic and T-Degen are new; other proofs are similar to those in Section 3.5 of [3]. Thus we only show the proofs for T-Stoic and T-Degen here, and keep other proofs in the appendix.

▶ **Lemma 19** (Pure Type). *If* $\langle \mathsf{k}, \Psi, \mathsf{v} \rangle \in \llbracket \mathsf{T}_{\mathsf{pu}} \rrbracket$, *then* $\langle \mathsf{k}, \emptyset, \mathsf{v} \rangle \in \llbracket \mathsf{T}_{\mathsf{pu}} \rrbracket$.

Proof. There are three cases: Unit, Nat, $\mathsf{T}_1 \to \mathsf{T}_2$. In each case, the definition of $\llbracket \mathsf{T} \rrbracket$ does not depend on Ψ, thus we can always choose $\Psi = \emptyset$. ◀

▶ **Theorem 20** (Stoic). *The following typing rule holds:*

$$\frac{\mathsf{pure}(\Gamma) \models \mathsf{v} : \mathsf{T}_1 \Rightarrow \mathsf{T}_2}{\Gamma \models \mathsf{v} : \mathsf{T}_1 \to \mathsf{T}_2} \qquad \text{(T-Stoic)}$$

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

(G1) $\mathsf{FV}(\mathsf{v}) \subseteq \mathsf{dom}(\Gamma)$
(G2) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{v}) \rangle \in \llbracket \mathsf{T}_1 \to \mathsf{T}_2 \rrbracket^*$

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

By the definition of $[\![\mathsf{T_1} \to \mathsf{T_2}]\!]^*$ and the fact that $\sigma(\mathsf{v})$ is a value, to prove (G2), we need to show:

(G2') $\langle \mathsf{k}, \Psi, \sigma(\mathsf{v}) \rangle \in [\![\mathsf{T_1} \to \mathsf{T_2}]\!]$

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

(A1) $\mathsf{FV}(\mathsf{v}) \subseteq \mathsf{dom}(\mathsf{pure}(\Gamma))$
(A2) $\langle \mathsf{k}, \emptyset, \sigma(\mathsf{v}) \rangle \in [\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]^*$

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

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

From the definition of $[\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]$, we know there exists $\mathtt{t}$ such that $\sigma(\mathsf{v}) = \lambda \mathsf{x}{:}\mathsf{T_1}.\mathsf{t}$. Suppose $\mathsf{j} < \mathsf{k}$ and $\langle \mathsf{j}, \Psi', \mathsf{v_1} \rangle \in [\![\mathsf{T_1}]\!]$, by the definition of $[\![\mathsf{T_1} \to \mathsf{T_2}]\!]$, to prove G2' we need to show:

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

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

▶ **Theorem 21** (Degeneration). *The following typing rule holds:*

$$\frac{\Gamma \models \mathsf{t} : \mathsf{T_1} \to \mathsf{T_2}}{\Gamma \models \mathsf{t} : \mathsf{T_1} \Rightarrow \mathsf{T_2}} \quad \text{(T-Degen)}$$

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

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

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

▶ **Lemma 22** (Degeneration Value). *If* $\langle \mathsf{k}, \Psi_1, \mathsf{v} \rangle \in [\![\mathsf{T_1} \to \mathsf{T_2}]\!]$*, then* $\langle \mathsf{k}, \Psi_2, \mathsf{v} \rangle \in [\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]$.

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

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

This is immediately from the definition of $[\![\mathsf{T_1} \to \mathsf{T_2}]\!]$. ◀

▶ **Lemma 23** (Degeneration Closed). *If* $\langle \mathsf{k}, \Psi, \mathsf{t} \rangle \in [\![\mathsf{T_1} \to \mathsf{T_2}]\!]^*$*, then* $\langle \mathsf{k}, \Psi, \mathsf{t} \rangle \in [\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]^*$.

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 $\mathsf{j} < \mathsf{k}, \mathsf{S}, \mathsf{S}', \mathsf{t}', \mathsf{S} :_{\mathsf{k}} \Psi$, if $(\mathsf{S}, \mathsf{t}) \longrightarrow^{\mathsf{j}} (\mathsf{S}', \mathsf{t}')$ and $\mathsf{irred}(\mathsf{S}', \mathsf{t}')$, then there exists Ψ' such that the following holds:

(G1) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k} - \mathsf{j}, \Psi')$
(G2) $\mathsf{S}' :_{\mathsf{k}-\mathsf{j}} \Psi'$
(G3) $\langle \mathsf{k} - \mathsf{j}, \Psi', \mathsf{t}' \rangle \in [\![\mathsf{T_1} \Rightarrow \mathsf{T_2}]\!]$

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

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

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

(A3) $\langle \mathsf{k}-\mathsf{j}, \Psi_1, \mathsf{t}' \rangle \in [\![\mathsf{T}_1 \to \mathsf{T}_2]\!]$

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

3.5 Parametric Polymorphism

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

$$\frac{\mathsf{pure}(\Gamma),\ \mathsf{X} \vdash \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{X}.\, \mathsf{t}_2 : \forall \mathsf{X}.\mathsf{T}} \ (\text{T-TAbs}) \qquad \frac{\Gamma \vdash \mathsf{t}_1 : \forall \mathsf{X}.\mathsf{T}}{\Gamma \vdash \mathsf{t}_1\ [\mathsf{T}_2] : [\mathsf{X} \mapsto \mathsf{T}_2]\mathsf{T}} \ (\text{T-TApp})$$

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

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

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

The meta-theory for polymorphic types developed in [3] can be reused to prove the two typing rules. We omit the details here.

4 Properties

In this section, we discuss properties of capability systems that support stoicity, including *effect polymorphism*, *precision and granularity* and *effect masking*.

4.1 Effect Polymorphism

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

```
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 observable effects itself; all effects it produces during the call are produced by the function `f`.

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. [21] 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 [18], 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 of them are effect-polymorphic. Introducing an additional effect variable makes the syntax less palatable and renders the type signature more complex.

Effect polymorphism is inherent in capability-based effect systems that support both stoic and free functions. This is because a stoic function like `map` can only indirectly use the capabilities carried 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, which means that no η-expansion is required in such cases. Moreover, when we fully apply a function, effect polymorphism is achieved 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. 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 difference is not surprising as η-expansion also makes a big difference in traditional type-and-effect systems [22]. 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 the 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 mutations, it is possible to prove the following two theorems:

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

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 $(U \Rightarrow V) \to T_1 \Rightarrow T_2$, 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 mutations, the theorems T-Poly and T-Pure do not hold, as in the outer stoic function, it may create local references that are captured in $T_1 \Rightarrow T_2$.

4.2 Precision and Granularity

There are subtle differences among the following types:

1. $(\mathsf{Int} \Rightarrow \mathsf{Int}) \to \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$

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. $(\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{List[Int]} \rightarrow \mathsf{List[Int]}$
7. $(\mathsf{Int} \rightarrow \mathsf{Int}) \Rightarrow \mathsf{List[Int]} \rightarrow \mathsf{List[Int]}$
8. $(\mathsf{Int} \rightarrow \mathsf{Int}) \Rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$

Function 1-4 may accept both stoic and free function as parameters, while the others only accept stoic functions. Functions 3-4, 7-8 are non-stoic, so they may capture capabilities from the outer environment, while the others may not. The inner functions of 2-3, 6-7 are pure, while the 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.

However, for the function type $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$, 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, capability-based effect systems are less precise than traditional type-and-effect systems. In type-and-effect systems [22], the effects of all functions are precise, there are no functions with unknown effects like free functions.

However, on the other hand, capability-based effect systems are more precise and can be arbitrarily fine-grained. In a monad-based system or type-and-effect system, if there is a function of the type $(\mathsf{Int} \rightarrow \mathsf{IO\ Int}) \rightarrow \mathsf{IO\ Int}$, we are not sure if the function performs other IO effects in addition to effects allowed by the provided function. In contrast, if a stoic function has the type $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$, we can be sure that the function at most produces effects allowed by the provided function. Monad-based systems and type-and-effect systems have to resort to effect parametricity in order to achieve the same expressiveness, which is a little heavy-weight thus not friendly for programmers.

The caller of a stoic function $(\mathsf{Int} \Rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$ can fine tune the effects of the call at any granularity level without complicating the type signature thanks to free function types. This provides a general approach to derive fine-grained capabilities from coarse-grained capabilities in a light-weight way, which is important for practicality of capability-aware languages.

4.3 Mutations and Effect Masking

As the calculus demonstrates, if we take references as capabilities, then we can control mutations. 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 mutations, 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:

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

If we admit such a rule in the type system, we will be able to type the following term `f` with the stoic type $\mathsf{Int} \rightarrow \mathsf{Int}$. Now a stoic function captures references from the environment.

It means two different calls to `f` can interfere. For security, this breaks the trust we give to stoic functions, as now f could be used as a secret channel to leak sensitive information. For compiler optimizations, dead code elimination or parallelization based on stoic function types becomes impossible, as the type $\mathsf{Int} \to \mathsf{Int}$ is not necessarily a pure function any more.

```
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 [22].

But how to support effect masking in the effect system? In [22], 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 mutations [18]. This approach corresponds to runST in Haskell [17], 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 the 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 capability-based effect systems: 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. In capability-based effect systems, the code looks like the following:

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

Norman Hardy, the designer of capability-based operating system KeyKos, pointed us to another usage of stoic functions to create a *secret*:

```
val mkSecret = () => {                   // () -> (()=> Unit, ()=> Int)
  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 of `count` as 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`.

5 Open Challenges: Scoped Capabilities

Even with stoic functions, there is still an open challenge for programming with capabilities: that is how to support *scoped capabilities* with light-weight syntax.

Checked exceptions is one example where scoped capabilities are useful. A naive approach to support checking exceptions based on capabilities is to introduce an exception type `Exn` and two primitive functions as follows: [4]

```
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 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(
```

[4] 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 $\forall T$ to simplify presentation.

```
    (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!

Scoped capabilities are also commonly used to make sure that the usage of some resource is confined in scope, as the following example shows:

```
def withFile[U](n: String)(@local fn: (@local File) => U): U = {
  val f = new File(n)
  try fn(f) finally f.close()
}

withFile("out.txt") { file => file.print("Hello, World!") }
```

Osvald et al [34] introduced second-class citizenship to handle the problem: the `@local` annotation in the code above means that a parameter is 2nd-class, i.e. it should not leak to the heap when the call returns. The system stipulates that only 1st-class values may be returned from a function. It is still unclear whether this rule restricts practical programming patterns.

Scoped capabilities are also related to *effect masking* in the literature [13, 22, 17].

6 Related Work

6.1 Capabilities

There has been a long history in using capabilities in computer systems for security. For example, KeyKOS [14] is the first operating system to implement confinement based on capabilities. [29] uses capabilities in the design of distributed operating systems. The recent verified secure kernel seL4 [16, 11] is also designed around capabilities.

The work by Miller et al. clears three common misconceptions about capability systems [27]: the *equivalence myth* that access control list systems and capability systems are formally equivalent; the *confinement myth* that capability systems cannot enforce confinement; and the *irrevocability myth* that capability-based access cannot be revoked.

The *object-capability model* is a security model based on objects [9, 26]. Several programming languages are implemented based on the model, such as E, Joule and Pony [25, 1, 6]. And there are some verification efforts for object-capabilities, like [30, 10, 39]. Our work complements this line of research by controlling capabilities in the type systems. We believe the type system improves expressiveness of capability models. For example, the guarantee of types makes it possible to establish trust on objects from untrusted sources and delegate capabilities to those objects without worrying about leaking of the capabilities.

6.2 Syntactic Control of Interference

The lineage of work on syntactic control of interference is closely related to stoic functions [36, 37, 33]. In its original formulation [36], two phrases [5] do not interfere if they do not share free variables, as variables are the only channels for interference. They introduce the dichotomy of *passive phrases* and *active phrases*. Passive phrases are impossible to cause interference, examples are side-effect-free expressions and procedures that do not assign to global variables. Their notion of passive prodcedures is slightly different from stoic functions in that stoic functions cannot read environmental mutable references either.

However, [36] is unsound under beta-reduction. The unsoundness is closely related to the insight in our earlier work on syntactic typing of stoic functions where we find stoic functions work better with call-by-value semantics than call-by-name semantics. With a naive type system for call-by-name semantics, it is impossible to prove the substitution lemma, which is essential in the soundness proof in the style of structural operational semantics.

The unsoundness is addressed in [37], which presents a type system with passive function types and active function types, which is akin to stoic function types and free function types. The system is formulated based on call-by-name semantics. In order to prove the substitution lemma, they have to impose more syntactic restrictions in the typing rule for application. For example, in the application $\mathsf{e}_1\ \mathsf{e}_2$, if e_1 is a passive function type $\mathsf{S} \to \mathsf{T}$, then (1) S should be passive if T is passive; (2) e_1 and e_2 should not share free variables. Our formalization of stoic functions based on stoic functions do not have such restrictions. Moreover, our treatment of stoic functions is more semantic in the sense that it shows how stoic functions behave with respect to the store.

[33] proposes a simpler substructural type system to solve the soundness problem. The typing judegement in their systems have the form $\Pi \mid \Gamma \vdash \mathsf{t} : \mathsf{T}$, where Π is the passive environment, and Γ is the active environment. Contraction is only allowed in the passive environment. They have a promotion rule that is similar to the rule T-Stoic, and a dereliction rule that is similar to T-Degen. Their system is based on categorical semantics.

A system based on modal logic and comonads is proposed [5], similar to the system $\lambda_{\mathsf{e}}^{\to\Box}$ introduced in [8]. The system establishes the link between capabilities and comonads, as well as capabilities and modal logic, which is not covered by our work. The design of our system, in contrast, is the simplest and most natural extension of the step-indexed semantics for simply-typed lambda calculus with mutations [3].

6.3 Effect Systems

[13, 22] 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.

[28] introduced monads for giving semantics to computational effects. [42] 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 [35, 4, 15] provides a different approach to give semantics to (user-defined) effects. Algebraic effects may also be equipped with a type system for checking effects [19]. Our work focuses on checking effects instead of giving semantics to effects, thus it is closer to [42].

[5] Phrases is a general term for expressions, procedures or statements in [36].

6.3.1 Effect Polymorphism

In Haskell, almost every general purpose higher-order function needs both a monadic version and a non-monadic version. As reported by [21, Section 1.6], Haskell has fractured into monadic and non-monadic sub-languages. Solutions based on parametric polymorphism, such as Koka [18], complicate the syntax and type signature of higher-order functions (though the user is supported by type inference). In Frank [20], effect polymorphism is achieved without introducing effect variables by using *ambient ability*. This confirms again the advantage of capabilities in achieving effect polymorphism.

6.3.2 Effect Masking

In [22], special syntax and typing rules for private regions are introduced to support masking of local effects. In Haskell, effect masking is supported by the ST monads and runST [17], the safety is guaranteed by parametricity of the rank-2 polymorphic type:

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

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 mutations [18]. In our system, effect masking is supported automatically without any special syntax or typing rule.

6.3.3 Memory Management

[40] introduced LIFO-style region-based memory management. [7] proposed a capability calculus in a continuation-passing style language, which allows arbitrary allocation and deallocation order. Static capabilities ensure that freed regions will not be used. They use bounded polymorphism to control aliasing of capabilities.

[12] proposed linear regions to support dynamic regions. Linearity gives arise to a kind of ownership. In constrast, stoic functions seems to express the dual notion *non-ownership*. That is, a function is stoic if it does not cpature any capabilities or resources. However, our system cannot be used for memory management because it does not support freeing unused memory.

6.3.4 Capability-Based Effect Systems

[34] introduced second-class citizenship. Second-class citizens observe a stack discipline; they cannot be leaked into the heap after the function call finishes. They implement an effect system for Scala based on the idea *effects as capabilities* and *capabilities as 2nd-class citizens*. 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 mutations, as heap references may not be returned from functions.

[23] 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 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.

6.3.5 Type-Based Capture Control

[24] 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.

7 Conclusion

We show that *stoicity* is a good discipline in capability-based systems, and we propose the notion of a *stoic function*, which is an incarnation of stoicity in functional languages, as a useful language abstraction to facilitate the construction of capability-based systems. We formalize stoic functions in STLC with mutation, taking heap references as capabilities. We prove that stoic functions in that setting enjoy non-interference of memory effects, which could be used to implement light-weight in-process memory isolation.

We show that capability-aware programming languages support a common form of effect-polymorphism without introducing effect variables. 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 of local mutations is supported automatically without any special syntax nor typing rule. The capability discipline can be applied flexibly in a system.

References

1. Inc. Agorics. Joule: Distributed application foundations, December 1995. URL: `http://www.erights.org/history/joule/index.html`.
2. Amal Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references, 2003.
3. Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.
4. Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.
5. Vikraman Choudhury and Neel Krishnaswami. Recovering purity with comonads and capabilities. *ArXiv*, abs/1907.07283, 2019.
6. Sylvan Clebsch, Sebastian Blessing, Sophia Drossopoulou, and Andrew McNeil. The pony language, 2015. URL: `https://github.com/ponylang/ponyc`.
7. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275. ACM, 1999.
8. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001. URL: `https://doi.org/10.1145/382780.382785`, `doi:10.1145/382780.382785`.

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

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

11 Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the sel4 microkernel. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 99–114. Springer, 2008.

12 Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *European Symposium on Programming*, pages 7–21. Springer, 2006.

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

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

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

16 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

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

18 Daan Leijen. The koka book, 2017. URL: `https://koka-lang.github.io/koka/doc/kokaspec.html`.

19 Daan Leijen. 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*, pages 486–499, 2017. URL: `http://dl.acm.org/citation.cfm?id=3009872`.

20 Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *POPL*, 2017.

21 Ben Lippmeier. *Type inference and optimisation for an impure world*. PhD thesis, Australian National University, 2010.

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

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

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

25 Mark S. Miller. The e language, 1997. URL: `http://www.erights.org/elang/index.html`.

26 Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

27 Mark S. Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Johns Hopkins University, 2003.

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

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

30 Toby C. Murray. *Analysing the security properties of object-capability patterns.* PhD thesis, University of Oxford, UK, 2010.

31 Martin Odersky. Scala — where it came from, where it is going, 2015. URL: `http://www.slideshare.net/Odersky/scala-days-san-francisco-45917092`.

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

33 Peter W. O'Hearn, John Power, Makoto Takeyama, and Robert D. Tennent. Syntactic control of interference revisited. *Theor. Comput. Sci.*, 228:211–252, 1999.

34 Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 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*, pages 234–251. ACM, 2016.

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

36 John C. Reynolds. Syntactic control of interference. In *POPL*, 1978.

37 John C Reynolds. Syntactic control of interference part 2. In *International Colloquium on Automata, Languages, and Programming*, pages 704–722. Springer, 1989.

38 Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

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

40 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

41 E. Dean Tribble, Mark S. Miller, Norman Hardy, Christopher T. Hibbert, and Eric C. Hill. Capability security for transparent distributed object systems, July 14 1998. US Patent 5,781,633.

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

8 Appendix: Proofs

▶ **Theorem 24** (Abstraction). *The following typing rule holds:*

$$\frac{\Gamma, \mathsf{x}{:}\mathsf{T}_1 \models \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \models \lambda \mathsf{x}{:}\mathsf{T}_1.\, \mathsf{t}_2 : \mathsf{T}_1 \Rightarrow \mathsf{T}_2} \qquad \text{(T-ABS)}$$

Proof. We need to show that for all k, σ, Ψ, if $\sigma :_{\mathsf{k},\Psi} \Gamma$, then $\langle \mathsf{k}, \Psi, \sigma(\lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2)\rangle \in [\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]^*$. From the definition of $[\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]^*$, we only need to prove:

(G0) $\langle \mathsf{k}, \Psi, \sigma(\lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2)\rangle \in [\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]$

Without loss of generality, suppose $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{j}, \Psi')$ for some $\mathsf{j} < \mathsf{k}$, and $\langle \mathsf{j}, \Psi', \mathsf{v}\rangle \in [\![\mathsf{T}_1]\!]$. Then by the definition of $[\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]$ we need to prove:

(G1) $\langle \mathsf{j}, \Psi', \sigma(\mathsf{t}_2[\mathsf{v}/\mathsf{x}])\rangle \in [\![\mathsf{T}_2]\!]^*$

From the definition of environment typing, extensibility of type sets and $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{j}, \Psi')$, we have:

(A) $\sigma[\mathsf{x} \mapsto \mathsf{v}] :_{\mathsf{j},\Psi'} \Gamma, \mathsf{x}{:}\mathsf{T}_1$

From the definition of $\Gamma, \mathsf{x}{:}\mathsf{T}_1 \models \mathsf{t}_2 : \mathsf{T}_2$ and A, we have:

(B) $\langle \mathsf{j}, \Psi', \sigma[\mathsf{x} \mapsto \mathsf{v}](\mathsf{t}_2)\rangle \in [\![\mathsf{T}_2]\!]^*$

But $\sigma[\mathsf{x} \mapsto \mathsf{v}](\mathsf{t}_2) = \sigma(\mathsf{t}_2[\mathsf{v}/\mathsf{x}])$, which completes the proof.

◀

▶ **Theorem 25** (Application). *The following typing rule holds:*

$$\frac{\Gamma \models \mathsf{t}_1 : \mathsf{T}_1 \Rightarrow \mathsf{T}_2 \qquad \Gamma \models \mathsf{t}_2 : \mathsf{T}_1}{\Gamma \models \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}_2} \qquad \text{(T-APP)}$$

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

(G1) $\mathsf{FV}(\mathsf{t}_1\ \mathsf{t}_2) \subseteq \mathsf{dom}(\Gamma)$

(G2) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}_1\ \mathsf{t}_2)\rangle \in [\![\mathsf{T}_2]\!]^*$

From the premises we know the following:

(A1) $\mathsf{FV}(\mathsf{t}_1) \subseteq \mathsf{dom}(\Gamma)$

(A2) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}_1)\rangle \in [\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]^*$

(A3) $\mathsf{FV}(\mathsf{t}_2) \subseteq \mathsf{dom}(\Gamma)$

(A4) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}_2)\rangle \in [\![\mathsf{T}_1]\!]^*$

G1 follows from A1 and A3 trivially. Let's focus on G2. By the definition of $[\![\mathsf{T}_2]\!]^*$, without loss of generality, we choose $\mathsf{j} < \mathsf{k}$ and $\mathsf{S} :_{\mathsf{k}} \Psi$, suppose $(\mathsf{S}, \sigma(\mathsf{t}_1\ \mathsf{t}_2)) \longrightarrow^{\mathsf{j}} (\mathsf{S}', \mathsf{t}')$ and $\mathsf{irred}(\mathsf{t}')$, we need to show that there exists Ψ':

(G2a) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k} - \mathsf{j}, \Psi')$

(G2b) $\mathsf{S}' :_{\mathsf{k}-\mathsf{j}} \Psi'$

(G2c) $\langle \mathsf{k} - \mathsf{j}, \Psi', \mathsf{t}'\rangle \in [\![\mathsf{T}_2]\!]$

Now let's consider $(\mathsf{S}, \sigma(\mathsf{t}_1))$. From A2, it's safe for k steps. It must be the case that $(\mathsf{S}, \sigma(\mathsf{t}_1)) \longrightarrow^{\mathsf{i}_1} (\mathsf{S}_1, \mathsf{t}_1')$ for some $\mathsf{i}_1 < \mathsf{j}$. Otherwise, $(\mathsf{S}, \sigma(\mathsf{t}_1\ \mathsf{t}_2))$ cannot stop in j steps. From A2 we know there exists Ψ_1 such that:[6]

(B1) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k} - \mathsf{i}_1, \Psi_1)$

(B2) $\mathsf{S}_1 :_{\mathsf{k}-\mathsf{i}_1} \Psi_1$

(B3) $\langle \mathsf{k} - \mathsf{i}_1, \Psi_1, \mathsf{t}_1'\rangle \in [\![\mathsf{T}_1 \Rightarrow \mathsf{T}_2]\!]$

[6] Note this covers the case $\sigma(\mathsf{t}_1)$ is a value, as in that case we can choose $\Psi' = \Psi$, $\mathsf{S}' = \mathsf{S}$ and $\mathsf{i}_1 = 0$, then the result trivially holds. We use the same trick in all proofs to simplify presentation.

From B3 and the definition of type sets, we know that:

(C1) $t_1' = \lambda x : T_1.t_3$

From B1 and the extensibility of typing environment, we have:

(D1) $\sigma :_{k-i_1,\Psi_1} \Gamma$

From D1 and $\Gamma \models t_2 : T_1$, we have:

(E1) $\langle k - i_1, \Psi_1, \sigma(t_2)\rangle \in [\![T_1]\!]^*$

Now consider E1, B2 and $(S_1, \sigma(t_2))$. It must be the case that $(S_1, \sigma(t_2)) \longrightarrow^{i_2} (S_2, t_2')$ for some $i_2 < j - i_1$. Otherwise, $(S, \sigma(t_1\ t_2))$ cannot stop in j steps. From E1 we know there exists Ψ_2 such that:

(F1) $(k - i_1, \Psi_1) \sqsubseteq (k - i_1 - i_2, \Psi_2)$

(F2) $S_2 :_{k-i_1-i_2} \Psi_2$

(F3) $\langle k - i_1 - i_2, \Psi_2, t_2'\rangle \in [\![T_1]\!]$

We know from F1 and index weakening and definition of state extension that:

(K1) $(k - i_1, \Psi_1) \sqsubseteq (k - i_1 - i_2 - 1, \Psi_2)$

(K2) $\langle k - i_1 - i_2 - 1, \Psi_2, t_2'\rangle \in [\![T_1]\!]$ [by K1, F3]

Now from the definition of $[\![T_1 \Rightarrow T_2]\!]$, B3, C1, K1 and K2 we get:

(L1) $\langle k - i_1 - i_2 - 1, \Psi_2, t_3[t_2'/x]\rangle \in [\![T_2]\!]^*$

From F2 and memory index weakening, we get:

(M1) $S_2 : k - i_1 - i_2 - 1\Psi_2$

Remember in the beginning we restrict ourself to the case $T_2 \neq U_1 \to U_2$. By the definition of $[\![T_2]\!]^*$, L1, M1 and $(S_2, t_3[t_2'/x])$, it must be the case that $(S_2, t_3[t_2'/x]) \longrightarrow^{i_3} (S_3, t_3')$ for some $i_3 = j - i_1 - i_2 - 1$. Otherwise, $(S, \sigma(t_1\ t_2))$ cannot stop at exactly the step j. So there exists Ψ_3 such that:

(N1) $(k - i_1 - i_2 - 1, \Psi_2) \sqsubseteq (k - i_1 - i_2 - i_3 - 1, \Psi_3)$

(N2) $S_3 :_{k-i_1-i_2-i_3-1} \Psi_3$

(N3) $\langle k - i_1 - i_2 - i_3 - 1, \Psi_3, t_3'\rangle \in [\![T_2]\!]$

To summarize, what we get is $(S, \sigma(t_1\ t_2)) \longrightarrow^{i_1+i_2+1+i_3} (S_3, t_3')$. By the determinism of evaluation, it must be that $S' = S_3$, $t' = t_3'$ and $j = i_1 + i_2 + 1 + i_3$.

Now we choose $\Psi' = \Psi_3$. The goals can be rewritten as follows:

(G2a) $(k, \Psi) \sqsubseteq (k - i_1 - i_2 - i_3 - 1, \Psi_3)$

(G2b) $S_3 :_{k-i_1-i_2-i_3-1} \Psi_3$

(G2c) $\langle k - i_1 - i_2 - i_3 - 1, \Psi_3, t_3'\rangle \in [\![T_2]\!]$

G2a follows from B1, F1, N1, transitivity of state extension and index weakening. G2b folllows from N2. G2c follows from N3.

◀

▶ **Theorem 26** (Reference). *The following typing rule holds:*

$$\frac{\Gamma \models t : T}{\Gamma \models \text{ref } t : \text{Ref } T} \qquad (\text{T-Ref})$$

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

(G1) $FV(\text{ref } t) \subseteq \Gamma$

(G2) $\langle k, \Psi, \sigma(\text{ref } t)\rangle \in [\![\text{Ref } T]\!]^*$

From the premise and the definition of semantic judgments we know:

(A1) $FV(t) \subseteq \Gamma$

(A2) $\langle k, \Psi, \sigma(t)\rangle \in [\![T]\!]^*$

G1 follows from A1, and G2 follows from A2 and the following lemma. ◀

▶ **Lemma 27** (Reference Closed). *If* $\langle k, \Psi, t\rangle \in [\![T]\!]^*$, *then* $\langle k, \Psi, \text{ref } t\rangle \in [\![\text{Ref } T]\!]^*$.

Proof. By the definition of expression typing, suppose that:

(Z1) $j < k$

(Z2) $S :_k \Psi$

(Z3) $(S, \text{ref } t) \longrightarrow^{j} (S', t')$

(Z4) $\text{irred}(S', t')$

We need to show that there exists Ψ' such that:

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

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

(G3) $\langle k - j, \Psi', t' \rangle \in [\![\text{Ref } T]\!]$

Now let's consider (S, t). From the premise, it's safe for k steps. It must be the case that $(S, t) \longrightarrow^{i_1} (S_1, t')$ for some $i_1 < j$. Otherwise, $(S, \text{ref } t)$ cannot stop in j steps. By the definition of expression typing, there exists Ψ_1 such that:

(B1) $(k, \Psi) \sqsubseteq (k - i_1, \Psi_1)$

(B2) $S_1 :_{k-i_1} \Psi_1$

(B3) $\langle k - i_1, \Psi_1, t' \rangle \in [\![T]\!]$

By the definition of value typing and B3, we know t' is a value. Thus there exists $l \notin \text{dom}(S_1)$:

(C1) $(S_1, \text{ref } t') \longrightarrow (S_1[l \mapsto t'], l)$

To summarize, what we get is $(S, \text{ref } t) \longrightarrow^{i_1+1} (S_1[l \mapsto t'], l)$. By the determinism of evaluation, it must be that $S' = S_1[l \mapsto t']$, $j = i_1 + 1$ and $t' = l$.

From $l \notin \text{dom}(S_1)$, B1 and B2, we have:

(D1) $l \notin \text{dom}(\Psi_1)$

(D2) $l \notin \text{dom}(\Psi)$

Let $\Psi_2 = \lfloor \Psi_1 \rfloor_{k-i_1-1} \cup (l \mapsto \lfloor [\![T]\!] \rfloor_{k-i_1-1})$. By the definition of approximation and D1, D2, we have:

(E1) $\lfloor \Psi_2 \rfloor_{k-i_1-1} = \Psi_2$

(E2) $\forall l' \in \text{dom}(\Psi_1). \lfloor \Psi_2 \rfloor_{k-i_1-1}(l') = \lfloor \Psi_1 \rfloor_{k-i_1-1}(l')$

From E2 and the definition of state extension, we have:

(F1) $(k - i_1 - 1, \Psi_1) \sqsubseteq (k - i_1 - 1, \Psi_2)$

(F2) $(k - i_1 - 1, \lfloor \Psi_1 \rfloor_{k-i_1-1}) \sqsubseteq (k - i_1 - 1, \Psi_2)$

(F3) $(k, \Psi) \sqsubseteq (k - i_1 - 1, \Psi_2)$ [By B1, F1 and index weakening]

From the definition of Ψ_2 and E1, we have the following:

(H1) $\lfloor \Psi_2 \rfloor_{k-i_1-1}(l) = \lfloor [\![T]\!] \rfloor_{k-i_1-1}$

(H2) $\langle k - i_1 - 1, \Psi_2, l \rangle \in [\![\text{Ref } T]\!]$ [By H1 and definition of $[\![\text{Ref } T]\!]$]

To show that $S_1[l \mapsto t'] :_{k-i_1-1} \Psi_2$, choose $i < k - i_1 - 1$ and $l' \in \Psi_2$, we need to show that:

(J1) $\langle i, \lfloor \Psi_2 \rfloor_i, S_1[l \mapsto t'](l') \rangle \in \lfloor \Psi_2 \rfloor_{k-i_1-1}(l')$

There two cases, depending on whether $l' = l$.

Case $l = l'$. From E1, we need to show $\langle i, \lfloor \Psi_2 \rfloor_i, t' \rangle \in [\![T]\!]_{k-i_1-1}$. From F1 and index weakening, we have $(k - i_1, \Psi_1) \sqsubseteq (i, \Psi_2)$. From $(i, \Psi_2) \sqsubseteq (i, \lfloor \Psi_2 \rfloor_i)$ and transitivity of state extension, we have $(k - i_1, \Psi_1) \sqsubseteq (i, \lfloor \Psi_2 \rfloor_i)$. Now from B3 and type set extensibility, we have $(i, \lfloor \Psi_2 \rfloor_i, t') \in [\![T]\!]$. As $i < k - i_1 - 1$, thus by definition of approximation we have $(i, \lfloor \Psi_2 \rfloor_i, t') \in \lfloor [\![T]\!] \rfloor_{k-i_1-1}$.

Case $l' \neq l$. We need to show that $\langle i, \lfloor \Psi_2 \rfloor_i, S_1(l') \rangle \in \lfloor \Psi_1 \rfloor_{k-i_1-1}(l')$. From B2 (i.e. $S_1 :_{k-i_1} \Psi_1$), we have $\langle k - i_1 - 1, \lfloor \Psi_1 \rfloor_{k-i_1-1}, S_1(l') \rangle \in \lfloor \Psi_1 \rfloor_{k-i_1}(l')$. From F2 and the fact that $\lfloor \Psi_1 \rfloor_{k-i_1}(l')$ is extensible, we have $\langle k - i_1 - 1, \Psi_2, S_1(l') \rangle \in \lfloor \Psi_1 \rfloor_{k-i_1}(l')$. Since $i < k - i_1 - 1$ and $(k - i_1 - 1, \Psi_2) \sqsubseteq (i, \lfloor \Psi_2 \rfloor_i)$, we have $\langle i, \lfloor \Psi_2 \rfloor_i, S_1(l') \rangle \in \lfloor \Psi_1 \rfloor_{k-i_1}(l)$. Since $i < k - i_1 - 1$, we have $\langle i, \lfloor \Psi_2 \rfloor_i, S_1(l') \rangle \in \lfloor \Psi_1 \rfloor_{k-i_1-1}(l')$ as needed.

To summarise, for the goals G1-G3, we know $S' = S_1[l \mapsto t']$, $j = i_1 + 1$ and $t' = l$ by determinism of evaluation. And we choose $\Psi' = \Psi_2$, now G1 holds from F3, G2 holds from

J1, G3 holds from H2. ◀

▶ **Theorem 28** (Dereference). *The following typing rule holds:*

$$\frac{\Gamma \models \mathsf{t} : \mathsf{Ref\ T}}{\Gamma \models !\mathsf{t} : \mathsf{T}} \qquad \text{(T-Deref)}$$

Proof. We need to show that for all k, Ψ, σ, if $\sigma :_{\mathsf{k},\Psi} \Gamma$, then:
(G1) $\mathsf{FV}(!\mathsf{t}) \subseteq \Gamma$
(G2) $\langle \mathsf{k}, \Psi, \sigma(!\mathsf{t}) \rangle \in [\![\mathsf{T}]\!]^*$

From the premise and the definition of semantic judgments we know:
(A1) $\mathsf{FV}(\mathsf{t}) \subseteq \Gamma$
(A2) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}) \rangle \in [\![\mathsf{Ref\ T}]\!]^*$

G1 follows from A1, and G2 follows from A2 and the following lemma. ◀

▶ **Lemma 29** (Dereference Closed). *If* $\langle \mathsf{k}, \Psi, \mathsf{t} \rangle \in [\![\mathsf{Ref\ T}]\!]^*$*, then* $\langle \mathsf{k}, \Psi, !\mathsf{t} \rangle \in [\![\mathsf{T}]\!]^*$.

Proof. By the definition of expression typing, suppose that:
(Z1) $\mathsf{j} < \mathsf{k}$
(Z2) $\mathsf{S} :_{\mathsf{k}} \Psi$
(Z3) $(\mathsf{S}, !\mathsf{t}) \longrightarrow^{\mathsf{j}} (\mathsf{S}', \mathsf{t}')$
(Z4) $\mathsf{irred}(\mathsf{S}', \mathsf{t}')$

We need to show that there exists Ψ' such that:
(G1) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k} - \mathsf{j}, \Psi')$
(G2) $\mathsf{S}' :_{\mathsf{k}-\mathsf{j}} \Psi'$
(G3) $\langle \mathsf{k} - \mathsf{j}, \Psi', \mathsf{t}' \rangle \in [\![\mathsf{T}]\!]$

Now let's consider (S, t). From the premise, it's safe for k steps. It must be the case that $(\mathsf{S}, \mathsf{t}) \longrightarrow^{\mathsf{i}_1} (\mathsf{S}_1, \mathsf{t}_1)$ for some $\mathsf{i}_1 < \mathsf{j}$. Otherwise, $(\mathsf{S}, !\mathsf{t})$ cannot stop in j steps. By the definition of expression typing, there exists Ψ_1 such that:
(B1) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k} - \mathsf{i}_1, \Psi_1)$
(B2) $\mathsf{S}_1 :_{\mathsf{k}-\mathsf{i}_1} \Psi_1$
(B3) $\langle \mathsf{k} - \mathsf{i}_1, \Psi_1, \mathsf{t}_1 \rangle \in [\![\mathsf{Ref\ T}]\!]$

By the definition of $[\![\mathsf{Ref\ T}]\!]$, we know that there exists l:
(C1) $\mathsf{t}_1 = \mathsf{l}$
(C2) $\mathsf{l} \in \Psi_1$
(C3) $\mathsf{l} \in \mathsf{dom}(\mathsf{S}_1)$ [By C2 and B2]

From C3, we know $(\mathsf{S}_1, !\mathsf{l})$ can take a step for some v:
(D1) $(\mathsf{S}_1, !\mathsf{l}) \longrightarrow (\mathsf{S}_1, \mathsf{S}_1(\mathsf{l}))$

From the definition of $[\![\mathsf{Ref\ T}]\!]$, B3 and C1, we have:
(E1) $\lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1}(\mathsf{l}) = \lfloor [\![\mathsf{T}]\!] \rfloor_{\mathsf{k}-\mathsf{i}_1}$

From B2, C2 and the definition of memory typing, we have:
(F1) $\langle \mathsf{k} - \mathsf{i}_1 - 1, \lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1-1}, \mathsf{S}_1(\mathsf{l}) \rangle \in \lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1}(\mathsf{l})$
(F2) $\langle \mathsf{k} - \mathsf{i}_1 - 1, \lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1-1}, \mathsf{S}_1(\mathsf{l}) \rangle \in \lfloor [\![\mathsf{T}]\!] \rfloor_{\mathsf{k}-\mathsf{i}_1}$ [By E1]
(F3) $\langle \mathsf{k} - \mathsf{i}_1 - 1, \lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1-1}, \mathsf{S}_1(\mathsf{l}) \rangle \in [\![\mathsf{T}]\!]$

To summarize, what we get is $(\mathsf{S}, !\mathsf{t}) \longrightarrow^{\mathsf{i}_1+1} (\mathsf{S}_1, \mathsf{v})$. By the determinism of evaluation, it must be that $\mathsf{S}' = \mathsf{S}_1$, $\mathsf{j} = \mathsf{i}_1 + 1$ and $\mathsf{t}' = \mathsf{S}_1(\mathsf{l})$.

For the goals G1-G4, we choose $\Psi = \Psi_1$. G1 holds from B1 and index weakening. G2 holds from B2 and memory index weakening. G3 holds from F3. ◀

▶ **Theorem 30** (Assignment). *The following typing rule holds:*

$$\frac{\Gamma \models \mathsf{t}_1 : \mathsf{Ref\ T} \qquad \Gamma \models \mathsf{t}_2 : \mathsf{T}}{\Gamma \models \mathsf{t}_1 := \mathsf{t}_2 : \mathsf{Unit}} \tag{T-Assign}$$

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

(G1) $\mathsf{FV}(\mathsf{t}_1 := \mathsf{t}_2) \subseteq \mathsf{dom}(\Gamma)$
(G2) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}_1 := \mathsf{t}_2)\rangle \in [\![\mathsf{Unit}]\!]^*$

From the premises we know the following:

(A1) $\mathsf{FV}(\mathsf{t}_1) \subseteq \mathsf{dom}(\Gamma)$
(A2) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}_1)\rangle \in [\![\mathsf{Ref\ T}]\!]^*$
(A3) $\mathsf{FV}(\mathsf{t}_2) \subseteq \mathsf{dom}(\Gamma)$
(A4) $\langle \mathsf{k}, \Psi, \sigma(\mathsf{t}_2)\rangle \in [\![\mathsf{T}]\!]^*$

G1 follows from A1 and A3 trivially. Let's consider G2. By the definition of $[\![\mathsf{Unit}]\!]^*$, without loss of generality, we choose $\mathsf{j} < \mathsf{k}$ and $\mathsf{S} :_{\mathsf{k}} \Psi$, suppose $(\mathsf{S}, \sigma(\mathsf{t}_1 := \mathsf{t}_2)) \longrightarrow^{\mathsf{j}} (\mathsf{S}', \mathsf{t}')$ and $\mathsf{irred}(\mathsf{t}')$, we need to show that there exists Ψ':

(G2a) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k}-\mathsf{j}, \Psi')$
(G2b) $\mathsf{S}' :_{\mathsf{k}-\mathsf{j}} \Psi'$
(G2c) $\langle \mathsf{k}-\mathsf{j}, \Psi', \mathsf{t}'\rangle \in [\![\mathsf{Unit}]\!]$

Now let's consider $(\mathsf{S}, \sigma(\mathsf{t}_1))$. From A2, it's safe for k steps. It must be the case that $(\mathsf{S}, \sigma(\mathsf{t}_1)) \longrightarrow^{\mathsf{i}_1} (\mathsf{S}_1, \mathsf{t}_1')$ for some $\mathsf{i}_1 < \mathsf{j}$. Otherwise, $(\mathsf{S}, \sigma(\mathsf{t}_1 := \mathsf{t}_2))$ cannot stop in j steps. From A2 we know there exists Ψ_1 such that:

(B1) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k}-\mathsf{i}_1, \Psi_1)$
(B2) $\mathsf{S}_1 :_{\mathsf{k}-\mathsf{i}_1} \Psi_1$
(B3) $\langle \mathsf{k}-\mathsf{i}_1, \Psi_1, \mathsf{t}'\rangle \in [\![\mathsf{Ref\ T}]\!]$

From B4 and the definition of $[\![\mathsf{Ref\ T}]\!]$, we know that:

(C1) $\mathsf{t}_1' = \mathsf{l}$

From B1 and the extensibility of typing environment, we have:

(D1) $\sigma_{\mathsf{k}-\mathsf{i}_1,\Psi_1}\Gamma$

From D1 and $\Gamma \models \mathsf{t}_2 : \mathsf{T}$, we have:

(E1) $\langle \mathsf{k}-\mathsf{i}_1, \Psi_1, \sigma(\mathsf{t}_2)\rangle \in [\![\mathsf{T}]\!]^*$

It must be the case that $(\mathsf{S}_1, \sigma(\mathsf{t}_2)) \longrightarrow^{\mathsf{i}_2} (\mathsf{S}_2, \mathsf{t}_2')$ for some $\mathsf{i}_2 < \mathsf{j}-\mathsf{i}_1$. Otherwise, $(\mathsf{S}, \sigma(\mathsf{t}_1\ \mathsf{t}_2))$ cannot stop in j steps. From E1 we know there exists Ψ_2 such that:

(F1) $(\mathsf{k}-\mathsf{i}_1, \Psi_1) \sqsubseteq (\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2, \Psi_2)$
(F2) $\mathsf{S}_2 :_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2} \Psi_2$
(F3) $\langle \mathsf{k}-\mathsf{i}_1-\mathsf{i}_2, \Psi_2, \mathsf{t}_2'\rangle \in [\![\mathsf{T}]\!]$

We know from F1 and index weakening and definition of state extension that:

(K1) $(\mathsf{k}-\mathsf{i}_1, \Psi_1) \sqsubseteq (\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1, \Psi_2)$
(K2) $\langle \mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1, \Psi_2, \mathsf{t}_2'\rangle \in [\![\mathsf{T}]\!]$ [by K1, F3]

Now from the definition of $[\![\mathsf{Ref\ T}]\!]$, C1 and B4 we get:

(L1) $\lfloor\Psi_1\rfloor(\mathsf{l}) = \lfloor[\![\mathsf{T}]\!]\rfloor$

From L1, F1 and F2, we know:

(L2) $\mathsf{l} \in \mathsf{S}_2$

From F4 we know t_2' is a value, thus we have the following step:

(L3) $(\mathsf{S}_2, \mathsf{l} := \mathsf{t}_2') \longrightarrow (\mathsf{S}_2[\mathsf{l} \mapsto \mathsf{t}_2'], \mathsf{unit})$

To summarize, what we get is $(\mathsf{S}, \sigma\mathsf{t}_1 := \mathsf{t}_2) \longrightarrow^{\mathsf{i}_1+1} (\mathsf{S}_2[\mathsf{l} \mapsto \mathsf{t}_2'], \mathsf{l})$. By the determinism of evaluation, it must be that $\mathsf{S}' = \mathsf{S}_2[\mathsf{l} \mapsto \mathsf{t}_2']$, $\mathsf{j} = \mathsf{i}_1 + \mathsf{i}_2 + 1$ and $\mathsf{t}' = \mathsf{unit}$.

By B1, F1 and index weakening, we have:

(M1) $(\mathsf{k}, \Psi) \sqsubseteq (\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1, \Psi_2)$

By the definition of $[\![\mathsf{Unit}]\!]$, we have:

(M2) $\langle \mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1, \Psi_2, \mathsf{unit}\rangle \in [\![\mathsf{Unit}]\!]$

To show that $\mathsf{S}_2[\mathsf{l} \mapsto \mathsf{t}'] :_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1} \Psi_2$, choose $\mathsf{i} < \mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1$ and $\mathsf{l}' \in \Psi_2$, we need to show that:

(O1) $\langle \mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{S}_2[\mathsf{l} \mapsto \mathsf{t}_2'](\mathsf{l}') \rangle \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l}')$

There two cases, depending on whether $\mathsf{l}' = \mathsf{l}$.

Case $\mathsf{l} = \mathsf{l}'$. We need to show $\langle \mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{t}_2' \rangle \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l}')$. From $\mathsf{i} < \mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1$, $(\mathsf{i}, \Psi_2) \sqsubseteq (\mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i})$ and transitivity of state extension, we have $(\mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2, \Psi_2) \sqsubseteq (\mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i})$. Now from F4 and type set extensibility, we have $(\mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{t}_2') \in [\![\mathsf{T}]\!]$. As $\mathsf{i} < \mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1$, thus by definition of approximation we have $(\mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{t}') \in \lfloor [\![\mathsf{T}]\!] \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}$. From B3, C1 and the definition of $[\![\mathsf{Ref}\ \mathsf{T}]\!]$, we have $\lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1}(\mathsf{l}) = \lfloor [\![\mathsf{T}]\!] \rfloor_{\mathsf{k}-\mathsf{i}_1}$. From L1 we know $\mathsf{l} \in \mathsf{dom}(\Psi_1)$. From K1 and the definition of state extension, we have $\lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l}) = \lfloor \Psi_1 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l}) = \lfloor [\![\mathsf{T}]\!] \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}$. Thus we have $(\mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{t}') \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l})$, which is exactly what we need.

Case $\mathsf{l}' \neq \mathsf{l}$. We need to show that $\langle \mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{S}_2(\mathsf{l}') \rangle \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l}')$. From F2 (i.e. $\mathsf{S}_2 :_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2} \Psi_2$) and the fact that $\lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2}(\mathsf{l}')$ is extensible, we have $\langle \mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1, \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}, \mathsf{S}_2(\mathsf{l}') \rangle \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2}(\mathsf{l}')$. Since $\mathsf{i} < \mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1$ and $(\mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1, \Psi_2) \sqsubseteq (\mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i})$, we have $\langle \mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{S}_2(\mathsf{l}') \rangle \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2}(\mathsf{l}')$. Since $\mathsf{i} < \mathsf{k} - \mathsf{i}_1 - \mathsf{i}_2 - 1$, we have $\langle \mathsf{i}, \lfloor \Psi_2 \rfloor_\mathsf{i}, \mathsf{S}_2(\mathsf{l}') \rangle \in \lfloor \Psi_2 \rfloor_{\mathsf{k}-\mathsf{i}_1-\mathsf{i}_2-1}(\mathsf{l}')$ as needed.

To summarise, for the goals G2a-G2c, we know $\mathsf{S}' = \mathsf{S}_1[\mathsf{l} \mapsto \mathsf{t}_2']$, $\mathsf{j} = \mathsf{i}_1 + \mathsf{i}_2 + 1$ and $\mathsf{t}' = \mathsf{unit}$ by determinism of evaluation. And we choose $\Psi' = \Psi_2$, now G2a holds from B1 and K1, G2b holds from O1, G2c holds from M2.

◀