CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Infoscience - École polytechnique fédérale de Lausanne

A Study of Capability-Based Effect Systems

Fengyun Liu

A thesis submitted for the degree of Master of Computer Science
at École Polytechnique Fédérale de Lausanne

Supervisors

Martin Odersky
Professor

Nada Amin
PhD Student

Sandro Stucki
PhD Student

School of Computer and Communication Sciences

Lausanne, January 2016

Acknowledgments

I thank Professor Martin Odersky for granting me the opportunity to work on the master project to explore capability-based effect systems.

I thank Sandro Stucki and Nada Amin for their mentoring. Without their help, I could not have made such a big progress from a novice in type systems and Coq in such a short time.

Finally, I'd like to thank my wife Yinai for her long-term support and love. A special thanks goes to my little daughter Tianyao – playing with her is one of the greatest joy in my spare time.

Abstract

The problem of *effect polymorphism* is a major obstacle to wide adoption of effect systems in the programming community. The absence of effect systems reduces compiler optimization opportunities and disables effect constraints on APIs in parallel and distributed computations.

This study shows that *capability-based* effect systems, equipped with *stoic functions* and *free functions*, can easily solve the problem of *effect polymorphism* without incurring notational burden on programmers. With this advantage, capability-based effect systems stand a better chance to be adopted by the programming community.

The central idea of *capability-based* effect system is that a capability is required in order to produce side effects. If capabilities are passed as function parameters, by tracking capabilities in the type system we can track effects in the program.

To ensure that capabilities are passed through function parameters, instead of being captured from the environment, we need to impose a *variable-capturing discipline*, stipulating that capability variables cannot be captured. Functions observing the discipline are called *stoic functions*, while functions not observing the discipline are called *free functions*.

Contents

Acknowledgments **1**

Abstract **2**

1 Introduction **6**

1.1 Effect Polymorphism . . . 6
1.2 Capability-Based Effect Systems . . . 7
1.3 Contributions . . . 8
1.4 Structure of Report . . . 9

2 System STLC-Pure **10**

2.1 Definitions . . . 10
2.1.1 Variable-Capturing Discipline . . . 10
2.1.2 Stoic Functions and Free Functions . . . 12
2.1.3 Where do Effects Come From . . . 13
2.1.4 Where do Capabilities Come From . . . 13
2.1.5 Why Treat Variables as Values . . . 13
2.1.6 What If a Function Has More than One Side Effect . . . 13
2.2 Soundness . . . 14
2.3 Effect Safety . . . 15
2.3.1 Informal Formulation . . . 15
2.3.2 Inhabited Types and Environments . . . 16
2.3.3 Formalization . . . 17
2.3.4 Proof . . . 19
2.3.5 An Intuitive Proof . . . 19

3 System STLC-Impure **22**

3.1 Definitions . . . 22
3.2 Soundness . . . 26
3.3 Effect Safety . . . 26
3.3.1 Formulation . . . 26
3.3.2 Proof . . . 29
3.4 Effect Polymorphism . . . 31
3.4.1 Axiomatic Polymorphism . . . 31
3.4.2 Currying Polymorphism . . . 32

3.4.3 Stoic Polymorphism 33
3.4.4 Discussion 33

4 System F-Pure **36**
4.1 Definitions 36
4.2 Soundness 38
4.3 Effect Safety 38
4.3.1 Inhabited Types and Environments 39
4.3.2 Formulation 40
4.3.3 Proof 40

5 System F-Impure **43**
5.1 Definitions 43
5.2 Soundness 45
5.3 Effect Safety 45
5.3.1 Formulation 45
5.3.2 Proof 47

6 Conclusion **49**
6.1 Related Work 49
6.2 Future Work 50

Bibliography **51**

List of Figures

2.1 System STLC-Pure 11
2.2 System STLC-Pure Capsafe Environment 18
3.1 System STLC-Impure First Formulation 23
3.2 Subtyping: Top−Pure and Top−Impure 24
3.3 System STLC-Impure 25
3.4 System STLC-Impure Capsafe Environment 28
3.5 System STLC-Impure Axioms 30
4.1 System F-Pure 37
4.2 System F-Pure Inhabited Environment 39
4.3 System F-Pure Capsafe Environment 41
5.1 System F-Impure 44
5.2 System F-Impure Capsafe Environment 46
5.3 System F-Impure Axioms 48

1 Introduction

The main motivation of this study is to turn Scala into an effect-disciplined programming language. Currently, Scala doesn't have the ability to track effects in the type system.

The absence of an effect system disables programmers from imposing effect constraints on APIs in parallel and distributed computations. For example, in parallel computing, the functions passed to the function pmap should have no side effects, in order to enable parallel processing on lists; however, it is currently impossible to do so in Scala:

```
def pmap(xs: List[Int], f: Int => Int): List[Int]
```

An effect system can differentiate pure code from impure code in a program, thus enabling optimization of pure code. For example, independent pure expressions can be executed in parallel; pure code that is dead can be safely eliminated; common pure expressions can be reduced to one computation; pure operations on data structures can be fused[CLS07]. The absence of an effect system makes these optimizations impossible.

As pointed out by Lukas et al.[ROH12], a main difficulty in introducing a practical effect system is how to handle the problem of *effect polymorphism*.

1.1 Effect Polymorphism

Effect polymorphism can be illustrated by the function map:

```
def map[A,B](f: A => B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
}
```

In an effect system, the effect of map depends on the passed in function f. If f has IO effects, then map also has IO effects. If f is pure, then map is pure as well. The way to express effect polymorphism in classical type-and-effect systems is to introduce a new type variable E to denote the generic effect:

```
def map[A, B, E](f: A => B @E)(l: List[A]): List[B] @E
```

Java, the only industrial language integrated with an effect system for checked exceptions, exemplifies the classical approach to effect polymorphism:

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

Due to the verbosity of syntax, effect polymorphism is rarely used in Java, which reduces the effectiveness of the effect system.

The problem of effect polymorphism is how to express effect-polymorphic functions with minimal syntactical overhead. *Capability-based* effect systems provide an elegant solution to this problem.

1.2 Capability-Based Effect Systems

The central idea of *capability-based* effect system is that a capability is required in order to produce side effects. If capabilities are passed as function parameters, by tracking capabilities in the type system we can track effects in the program.

To ensure that capabilities are passed through function parameters, instead of being captured from the environment, we need to impose a *variable-capturing discipline*, stipulating that capability variables cannot be captured. Functions observe the discipline are called *stoic functions*, while functions don't observe the discipline are called *free functions*. We use $\rightarrow$ to denote the type of stoic functions and $\Rightarrow$ to denote the type of free functions. The following example shows that incorrect capturing of capability variables in a stoic function will generate a typing error:

```scala
def map(xs: List[Int], f: Int => Int): List[Int]
def pmap(xs: List[Int], f: Int -> Int): List[Int]
def print(x: Any, c: IO): ()

def bar(xs: List[Int])(c: IO) =
    map(xs, { x => print(x, c); x })

def foo(xs: List[Int], c: IO) =
    pmap(xs, { x => print(x, c); x }) // Error, stoic function required
```

In the code above, the function map can take either a stoic function or a free function, while the function pmap can only take a stoic function. The type signature $\mathsf{Int} \rightarrow \mathsf{Int}$ in pmap ensures that functions passed to pmap must be pure. That's why there is a typing error in the function foo, as the passed anonymous function is free instead of stoic.

With the combination of free functions and stoic functions, capability-based effect systems can solve the problem of *effect polymorphism* easily, while incurring no syntactical overhead. For example, the effect-polymorphic function map can be implemented as follows:

```
def map[A,B](f: A => B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
}
def squareImpure(c: IO) = map { x => println(x)(c); x*x }
def squarePure(l: List[Int]) = map { x => x*x } l
```

In the code above, the function map is exactly the same as it can be written in Scala now, which would be typed as $(\mathsf{A} \Rightarrow \mathsf{B}) \rightarrow \mathsf{List[A]} \Rightarrow \mathsf{List[B]}$. The function squareImpure is stoic, which would be typed as $\mathsf{IO} \rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$. The function squarePure is also stoic, which would be typed as $\mathsf{List[Int]} \rightarrow \mathsf{List[Int]}$. By just checking the type signature, we can conclude that squareImpure can only have IO effects and squarePure is pure. No annotation is required, effect polymorphism just works naturally. We'll see details of effect polymorphism in the chapter STLC-Impure (Section 3.4).

1.3 Contributions

The main contributions of this study are as follows:

- We formulate and prove soundness and effect safety of four capability-based effect systems, which can serve as the foundation for implementing capability-based effect system in functional programming languages. The formalization is done in Coq based on the locally-nameless representation[Cha11] and hosted on Github[1].
- We propose an approach to solve the problem of *effect polymorphism* with no syntactical overhead, thanks to free functions and stoic functions (Section 3.4).

[1] https://github.com/liufengyun/stoic

1.4 Structure of Report

In following chapters, we'll introduce four systems of increasing complexity, namely STLC-Pure, STLC-Impure, F-Pure and F-Impure. The latter three are a gradual enrichment of STLC-Pure with free functions, subtyping and universal types.

- STLC-Pure is a variant of *simply typed lambda calculus* with only stoic functions.
- STLC-Impure is an extension of STLC-Pure with free functions and subtyping.
- F-Pure is an extension of STLC-Pure with universal types.
- F-Impure is an extension of F-Pure with free functions (without subtyping).

We discuss *effect polymorphism* in the chapter STLC-Impure (Section 3.4).

2 System STLC-Pure

This chapter describes a variant of the *simply typed lambda calculus* with the extension of capabilities. We call this system *STLC-Pure*, because in this system all functions must observe a variable-capturing discipline.

The system STLC-Pure, though conceptually simple, can quite well demonstrate the main features of capability-based effect systems. We'll first introduce the formalization, then discuss soundness and effect safety. Concepts introduced here will be a foundation for more complex systems in later chapters.

2.1 Definitions

Formally, STLC-Pure is obtained by introducing a capability type and imposing a variable-capturing discipline on lambda abstractions. Figure 2.1 presents the full definition of STLC-Pure.

The syntax is almost the same as standard STLC, except the addition of the capability type E and the taking of variables as values. The evaluation rules are exactly the same, with standard call-by-value small-step semantics. The typing rule T-ABS is slightly changed by performing an operation pure on the environment. The peculiarities in the formalization are explained below.

2.1.1 Variable-Capturing Discipline

The most important change to the standard STLC lies in the following typing rule:

$$\frac{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{S} \vdash \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{S}.\mathsf{t}_2 : \mathsf{S} \to \mathsf{T}} \quad \text{(T-ABS)}$$

This typing rule imposes a *variable-capturing discipline* on lambda abstractions. This discipline stipulates that only variables whose type is not a capability type can be captured in a lambda abstraction.

The discipline is implemented with the helper function pure, which removes all variable bindings of the capability type E from the typing environment. It's easy to verify that the function pure satisfies following properties:

Syntax

$$
\begin{array}{lll}
\mathsf{t} ::= & & \text{terms:} \\
 & \mathsf{x} & \text{variable} \\
 & \lambda \mathsf{x}{:}\mathsf{T}.\mathsf{t} & \text{abstraction} \\
 & \mathsf{t}\,\mathsf{t} & \text{application} \\
\\
\mathsf{v} ::= & & \text{values:} \\
 & \lambda \mathsf{x}{:}\mathsf{T}.\mathsf{t} & \text{abstraction value} \\
 & \mathrm{x} & \text{variable value} \\
\\
\mathsf{T} ::= & & \text{types:} \\
 & \mathsf{B} & \text{basic type} \\
 & \mathsf{E} & \text{capability type} \\
 & \mathsf{T} \to \mathsf{T} & \text{type of functions}
\end{array}
$$

Evaluation $\boxed{\mathsf{t} \longrightarrow \mathsf{t}'}$

$$\frac{\mathsf{t}_1 \longrightarrow \mathsf{t}_1'}{\mathsf{t}_1\,\mathsf{t}_2 \longrightarrow \mathsf{t}_1'\,\mathsf{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathsf{t}_2 \longrightarrow \mathsf{t}_2'}{\mathsf{v}_1\,\mathsf{t}_2 \longrightarrow \mathsf{v}_1\,\mathsf{t}_2'} \quad \text{(E-APP2)}$$

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

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

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

$$\frac{\mathsf{pure}(\Gamma),\, \mathsf{x} : \mathsf{S} \vdash \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{S}.\mathsf{t}_2 : \mathsf{S} \to \mathsf{T}} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{S} \to \mathsf{T} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{S}}{\Gamma \vdash \mathsf{t}_1\,\mathsf{t}_2 : \mathsf{T}} \quad \text{(T-APP)}$$

Pure Environment

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

Figure 2.1: System STLC-Pure

LEMMA (Pure-Distributivity). pure (Γ, Δ) = pure Γ, pure Δ

LEMMA (Pure-Idempotency). pure (pure Γ) = pure Γ

Initially, we've tried a variable-capturing discipline where no free variables can be captured, i.e. $\mathsf{pure}(\Gamma) = \varnothing$. While this definition is certainly effect safe, the system is not very expressive, as we lose the ability to create closures in the system, which is usually considered to be an essential feature of functional programming.

In addition to the capability type E, we also tried to exclude variables of the type $\mathsf{T} \to \mathsf{E}$ (for any T) from the pure environment. As an over-approximation, this version is certainly effect-safe. However, it rejects more types than necessary, thus reduces expressiveness of the system.

We introduce two useful definitions with the help of the function pure.

DEFINITION (Pure-Environment). An environment Γ is pure if $\mathsf{pure}\ \Gamma = \Gamma$.

DEFINITION (Pure-Type). A type is pure if the type can exist in a pure environment.

From the definition, it's obvious that the type B is pure, while E is impure. However, a pure function type doesn't imply the corresponding function is pure. For example, $\mathsf{E} \to \mathsf{B}$ is a pure type, but functions that inhabit the type may have side effects, thus is impure. The meta-theory of the system ensures that if the capability type E doesn't appear in the type signature of a function, then the function must be pure. For example, all functions that can be typed as $\mathsf{B} \to \mathsf{B}$ are guaranteed to be pure.

2.1.2 Stoic Functions and Free Functions

The variable-capturing discipline makes the functions in STLC-Pure different from functions in standard STLC. In STLC, functions can capture any variables in scope, while in STLC-Pure functions can only capture variables whose type is not the capability type E. To differentiate them (which is important as in later systems both exist), we call the more effect-disciplined functions *stoic functions* (or stoics) and the other *free functions*. We use $\to$ to denote the type of stoic functions and $\Rightarrow$ to denote the type of free functions.

Stoic functions are essential in capability-based effect systems. If functions are allowed to capture capability variables in scope, it will be impossible to tell whether a function has side effect or not (and what kind of effect) by just checking its type. Stoic functions are effect-disciplined in the sense that the only way for stoic functions to have side effects is to pass a capability as parameter, thus it can be captured by the type system.

Stoic functions are not necessarily pure functions. Stoic functions can have side effects, and if they do have side effects they are honest about that in their type signature. For example, the following function hello is a stoic function with IO effects[2].

```
def hello(c:IO) = println("hello, world!", c)
```

In the following code snippet, the function f must be pure, as it doesn't take any capability as parameter. The type system guarantees that the function indeed cannot produce any side effects.

```
def twice(f: Int -> Int)(x: Int) = f (f x)
```

[2]For the sake of readability, we'll use a syntax similar to Scala in this report. In particular, we'll use $\to$ for the type of stoic functions, and $\Rightarrow$ for free functions.

2.1.3 Where do Effects Come From

There is no formalization of effects in the current effect system. We assume the existence of primitive functions like println and readln, which take capabilities to produce side effects.

```
def println: String -> IO -> ()
def readln: IO -> String
```

2.1.4 Where do Capabilities Come From

It is impossible to create capabilities in the current system. Where do capabilities come from? There are two possible answers: (1) all capabilities are from the run-time and passed to the program through the main method; (2) there are no capabilities; they can be erased before evaluation, without changing the meaning of programs.

2.1.5 Why Treat Variables as Values

As discussed above, there is no way to create a value of capabilities explicitly. Thus, a function taking a parameter of the capability type E can never be executed in the *call-by-value* semantics, unless variables are values. The same is true for the base type B.

Treating variables as values ensures that substitution of a term with a value of the base type B or the capability type E can actually happen in the system, thus makes the preservation proof more convincing.

2.1.6 What If a Function Has More than One Side Effect

There is no support for a function with more than one kind of effects in the current system. For example, in the following code snippet, c1 cannot be used in the function body, as it's removed by pure in the typing of the function body.

```
def error(e:Error)(c1:IO)(c2:Throw) = {
  println("error happen!", c1) // Error, can't capture c1
  throw c2 e
}
```

It's straight-forward to extend the system with pairs or tuples to overcome this limitation. However, this is not an issue for later systems with *free functions*, thus we don't pursue the extension of pairs and tuples here.

2.2 Soundness

We follow the standard formulation of soundness in TAPL [Pie02], which consists of *progress* and *preservation*, defined as follows:

THEOREM (Progress). If $\varnothing \vdash \mathsf{t} : \mathsf{T}$, then either t is a value or there is some t' with $\mathsf{t} \longrightarrow \mathsf{t}'$.

THEOREM (Preservation). If $\Gamma \vdash \mathsf{t} : \mathsf{T}$, and $\mathsf{t} \longrightarrow \mathsf{t}'$, then $\Gamma \vdash \mathsf{t}' : \mathsf{T}$.

The proof of progress is the same as the proof in standard STLC. However, there is a significant difference in the proof of preservation. The classic proof of preservation for STLC (as shown in TAPL) depends on a substitution lemma, which is formulated as follows:

LEMMA (Subsitution-Classic). If $\Gamma,\ \mathsf{x} : \mathsf{S} \vdash \mathsf{t} : \mathsf{T}$, and $\Gamma \vdash \mathsf{s} : \mathsf{S}$, then $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t} : \mathsf{T}$.

However, this substitution lemma doesn't hold in the current system. For a counter-example, let's assume that $\Gamma = \{\mathsf{f} : \mathsf{E} \to \mathsf{B},\ \mathsf{c} : \mathsf{E}\}$, then it's obviously that following two typing relations hold:

$\mathsf{f} : \mathsf{E} \to \mathsf{B},\ \mathsf{c} : \mathsf{E},\ \mathsf{x} : \mathsf{B} \vdash \lambda \mathsf{z}{:}\mathsf{B}.\mathsf{x}\ :\ \mathsf{B} \to \mathsf{B}$
$\mathsf{f} : \mathsf{E} \to \mathsf{B},\ \mathsf{c} : \mathsf{E} \vdash \mathsf{f}\ \mathsf{c}\ :\ \mathsf{B}$

However, the following typing relation doesn't hold if we replace x with $\mathsf{f}\ \mathsf{c}$.

$\mathsf{f} : \mathsf{E} \to \mathsf{B},\ \mathsf{c} : \mathsf{E} \vdash \lambda \mathsf{z}{:}\mathsf{B}.\mathsf{f}\ \mathsf{c}\ :\ \mathsf{B} \to \mathsf{B}$.

In fact, the substituted term $\lambda \mathsf{z}{:}\mathsf{B}.\mathsf{f}\ \mathsf{c}$ cannot be typed, as according to the typing rule T-ABS, it cannot capture the capability variable c in the environment. To overcome this problem, we stipulate that the term s must be a value. Remember that in the current system, both lambda abstractions and variables are values, thus substitution of variables of the capability type E and the base type B can happen. The new formulation is as follows:

LEMMA (Subsitution-New). If $\Gamma,\ \mathsf{x} : \mathsf{S} \vdash \mathsf{t} : \mathsf{T}$, s is a value and $\Gamma \vdash \mathsf{s} : \mathsf{S}$, then $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t} : \mathsf{T}$.

The restriction that s must be a value implies that in a function call, arguments much first be evaluated to a value before the function is applied. Therefore, capabilities can only work with strict evaluation.

Interestingly, this strict evaluation requirement contrasts capability-based effect systems with monad-based effect systems. In Haskell, if strict evaluation is adopted, it will be impossible to track effects in the type system, as demonstrated by the following code snippet:

```
inc n = (\x -> n + 1) (putStrLn (show n))
```

The function inc has the type $(\mathsf{Num\ a}, \mathsf{Show\ a}) \Rightarrow \mathsf{a} \to \mathsf{a}$. By just checking its type, we would think it has no side effects because no IO monads appear in the type signature. However, if Haskell adopts strict evaluation, the function call $\mathsf{putStrLn}\ (\mathsf{show\ n})$ will be executed, thus breaking the monad-based effect system.

2.3 Effect Safety

Does the system really work? This question prompts us to formulate and prove effect safety of the system. We start by formulating effect safety informally, then put forward a formal formulation, and finally prove effect safety of the system.

2.3.1 Informal Formulation

A straight-forward violation of effect safety is for functions that are taken as pure to have side effects inside the function body. Thus, a tentative formulation would be as follows:

DEFINITION (Effect-Safety-Informally-1). A function typed in a pure environment cannot have side effects inside.

However, this formulation is obviously problematic, as we know stoic functions can have side effects if it takes a capability parameter. Thus, we need to restrict the functions to those not taking capability parameters:

DEFINITION (Effect-Safety-Informally-2). A function, not taking any capability parameter and typed in a pure environment, cannot have side effects inside.

This formulation looks more satisfactory, but it's a little cumbersome. If we inspect the typing rule T-ABS closely, we can find that if S is not a capability type, $\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{S}$ is equal to $\mathsf{pure}(\Gamma,\ \mathsf{x} : \mathsf{S})$.

$$\frac{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{S} \vdash \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x{:}S}.\ \mathsf{t}_2 : \mathsf{S} \to \mathsf{T}} \qquad \text{(T-ABS)}$$

Thus, instead of saying the function $\lambda x{:}S.\, t_2$ cannot have side effects inside, we say the term t_2 cannot have side effects in a pure environment. As we know, capabilities are required to produce side effects. Thus, the term t_2 cannot have side effects if we cannot construct a term of the capability type E in a pure environment. This observation leads us to the following statement of effect safety:

DEFINITION (Effect-Safety-Informally-3). It's impossible to construct a term of the capability type E in a pure environment.

However, this formulation cannot be proved. For a counter-example, let's assume $\Gamma = \{f : B \to E,\ x : B\}$. It's obvious that Γ is pure, but we can construct the term $f\ x$ of the capability type E.

The cause of the problem is that in a pure environment, there might exist uninhabited types like $B \to E$. Existence of uninhabited types in a pure environment doesn't pose a problem to the system; a function taking a parameter of an uninhabited type can never be actually called, thus is always effect-safe. So we only need to consider environments with only variables of inhabited types.

To convince readers that the current system is effect-safe, we need to exclude and only exclude uninhabited types from the pure environment and then prove that it is impossible to construct a term of the capability type E in this restricted environment. We arrive at the following formulation:

DEFINITION (Effect-Safety-Informally-4). It's impossible to construct a term of the capability type E in a pure environment with only variables of inhabited types.

2.3.2 Inhabited Types and Environments

We need to define the concept *inhabited types* precisely. What types are inhabited? Obviously, if $\varnothing \vdash t : T$, then T is inhabited. However, given a typing relation $\Gamma \vdash t : T$, we cannot immediately conclude that T is inhabited. We need to ensure that Γ only contains inhabited types. Otherwise, any type is inhabited if Γ contains a variable of the corresponding type.

An intuition is that, given $\Gamma \vdash t : T$, $x : S \in \Gamma$ and S is inhabited, we can remove $x : S$ from Γ and substitute x in the term t with a witness of the type S to obtain a new term t'. The substitution lemma tells us that the new term t' still has the type T. Continue this line of thought, we'll find out that all inhabited types can be inhabited in the empty environment. Thus, a tentative definition is as follows:

DEFINITION (Inhabited-Type-First-Try). A type T is inhabited if there exists a term t with $\varnothing \vdash \mathsf{t} : \mathsf{T}$.

However, this definition is not satisfactory in our case, as in STLC-Pure there doesn't exist values for the base type B and the capability type E, except variables, and we do want both types to be inhabited. A natural approach is to extend the empty environment with one variable of the base type and one of the capability type:

DEFINITION (Inhabited-Type-Second-Try). A type T is inhabited if there exists a term t with $\mathsf{x} : \mathsf{B},\ \mathsf{y} : \mathsf{E} \vdash \mathsf{t} : \mathsf{T}$.

This definition indeed gives us all inhabited types in STLC-Pure. Types like E, B, $\mathsf{E} \rightarrow \mathsf{B}$, $\mathsf{E} \rightarrow \mathsf{E}$, $(\mathsf{B} \rightarrow \mathsf{E}) \rightarrow \mathsf{E}$, etc., are all inhabited, while types like $\mathsf{B} \rightarrow \mathsf{E}$ and $\mathsf{E} \rightarrow \mathsf{B} \rightarrow \mathsf{E}$ are uninhabited. The reader might want to construct a witness of the type $\mathsf{B} \rightarrow \mathsf{E}$ as follows:

$$\mathsf{x} : \mathsf{B},\ \mathsf{y} : \mathsf{E} \vdash \lambda \mathsf{x}{:}\mathsf{B}.\, \mathsf{y} : \mathsf{B} \rightarrow \mathsf{E}$$

The typing relation doesn't hold, because the typing rule T-ABS would remove the binding $\mathsf{y} : \mathsf{E}$ from the environment in the typing of the function body y.

The second formulation looks good, however, we can do better with the following definition:

DEFINITION (Inhabited-Type-Final). A type T is inhabited if there exists a value v with the typing $\mathsf{x} : \mathsf{B},\ \mathsf{y} : \mathsf{E} \vdash \mathsf{v} : \mathsf{T}$.

What if the term t in the second definition is an application? In that case, t must be able to take a step until it becomes a value due to *progress* and *normalization* of the system[3]. And the *preservation* theorem tells us the type remains unchanged during evaluation. This final definition makes proofs related to inhabited types simpler. It's useful to give a definition of inhabited environments as well:

DEFINITION (Inhabited-Environment). An environment Γ is inhabited if it only contains variables of inhabited types.

2.3.3 Formalization

With the formal definition of inhabited environment, we can formalize effect safety as follows:

[3] We didn't prove normalization of STLC-Pure, but the proof should be similar to the proof in standard STLC. We only proved progress in the empty environment, and the proof can be adapted to prove progress under $\{\mathsf{x} : \mathsf{B}, \mathsf{y} : \mathsf{E}\}$.

DEFINITION (Effect-Safety-Inhabited). If Γ is a pure and inhabited environment, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

However, this formulation doesn't give rise to a direct proof. In fact, this statement is too strong. Some uninhabited types, such as $\mathsf{E} \to \mathsf{B} \to \mathsf{E}$, don't enable us to create a term of the type E, thus it's safe to keep them in the environment. This implies it's possible to impose a looser restriction on Γ, as long as all types that can appear in a pure and inhabited environment can also appear in Γ.

When we examine the problem more closely, we found that through the lens of the *Curry-Howard isomorphism*, effect safety actually says that it is impossible to prove the capability type E from a group of "good" premises. Thus, we can classify all types (propositions) into two groups: in one group E cannot be proved and in the other group E can be proved. This leads us to a formulation of *capsafe environment*[4] given in Figure 2.2. The coined word *capsafe* is an abbreviation of capability-safe, and *caprod* an abbreviation of capability-producing.

Capsafe Type

$$\mathsf{B} \quad \text{capsafe} \qquad \text{(CS-BASE)}$$

$$\frac{\mathsf{S} \quad \text{caprod}}{\mathsf{S} \to \mathsf{T} \quad \text{capsafe}} \ \text{(CS-FUN1)}$$

$$\frac{\mathsf{T} \quad \text{capsafe}}{\mathsf{S} \to \mathsf{T} \quad \text{capsafe}} \ \text{(CS-FUN2)}$$

Caprod Type

$$\mathsf{E} \quad \text{caprod} \qquad \text{(CP-EFF)}$$

$$\frac{\mathsf{S} \text{ capsafe} \qquad \mathsf{T} \text{ caprod}}{\mathsf{S} \to \mathsf{T} \quad \text{caprod}} \ \text{(CP-FUN)}$$

Capsafe Environment

$$\varnothing \quad \text{capsafe} \qquad \text{(CE-EMPTY)}$$

$$\frac{\Gamma \text{ capsafe} \qquad \mathsf{T} \text{ capsafe}}{\Gamma, \mathsf{x} : \mathsf{T} \quad \text{capsafe}} \ \text{(CE-VAR)}$$

Figure 2.2: System STLC-Pure Capsafe Environment

In the definition, types like $\mathsf{B} \to \mathsf{B}$, $\mathsf{E} \to \mathsf{E}$, $\mathsf{E} \to \mathsf{B}$ and $(\mathsf{B} \to \mathsf{E}) \to \mathsf{B}$ are considered as *capsafe*, while types like $\mathsf{B} \to \mathsf{E}$, $(\mathsf{E} \to \mathsf{B}) \to \mathsf{E}$ are considered as *caprod*. Only *capsafe* types can appear in a *capsafe* environment. To inspect the formalization in detail, we can ask several questions.

Are capsafe types inhabited? Not necessarily. The type $\mathsf{E} \to \mathsf{B} \to \mathsf{E}$ is capsafe but uninhabited.

[4]Sandro Stucki initially suggested the idea of using *caprod* for the definition of *pure* environments. I developed it to be a formulation of *capsafe* environments and used it in the proof of effect safety.

Allowing this type in the capsafe environment doesn't enable us to construct a term of the capability type E.

Are inhabited types capsafe? Yes, except the capability type E. As the capability type E cannot appear in the pure environment, this is not a problem.

Are caprod types uninhabited? Yes, except the capability type E. As E is also excluded in the pure environment, it's justified to remove it from the capsafe environment.

Why this formulation of capsafe environment is acceptable? In short, it is because the statement *Effect-Safety-Inhabited* is logically implied by the more general statement *Effect-Safety*:

DEFINITION (Effect-Safety). If Γ is capsafe, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

The logical implication holds because a pure and inhabited environment is also a capsafe environment. This claim has been formally proved:

LEMMA (Inhabited-Capsafe). If the type T is inhabited, then either T is capsafe or $\mathsf{T} = \mathsf{E}$.

THEOREM (Inhabited-Pure-Capsafe-Env). If Γ is pure and inhabited, then Γ is capsafe.

2.3.4 Proof

The proof of effect safety depends on following lemmas, most of them are straight-forward to prove. Effect safety follows immediately from the lemma *Capsafe-Env-Capsafe*.

LEMMA (Capsafe-Not-Caprod). If type T is capsafe, then T is not caprod.

LEMMA (Capsafe-Or-Caprod). For any T, T is either capsafe or caprod.

LEMMA (Capsafe-Env-Capsafe). If Γ is capsafe and $\Gamma \vdash \mathsf{t} : \mathsf{T}$, then T is capsafe.

THEOREM (Effect-Safety). If Γ is capsafe, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

2.3.5 An Intuitive Proof

There exists an intuitive proof of effect safety without resorting to capsafe environments. The main insight is that the statement *Effect-Safety-Inhabited* is logically implied by the statement *Effect-Safety-Intuitive*:

DEFINITION (Effect-Safety-Inhabited). If Γ is a pure and inhabited environment, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

DEFINITION (Effect-Safety-Intuitive). There doesn't exist value v with $\mathsf{x : B \vdash v : E}$.

The statement *Effect-Safety-Intuitive* trivially holds, because the value v can either be a variable or a function, in neither case can it be typed as the capability type E.

Why does the logical implication hold? In short, if *Effect-Safety-Inhabited* doesn't hold, then *Effect-Safety-Intuitive* doesn't hold either. Thus, the latter logically implies the former. In the following, we present an informal proof, the core idea is that if Γ is pure and inhabited, and $\Gamma \vdash \mathsf{t : E}$, then we can "collapse" Γ to $\{\mathsf{b : B}\}$ through substitution of the witnesses of the pure and inhabited types in Γ.

For a pure and inhabited environment $\Gamma = \{\mathsf{x : T,\ y : S,\ \ldots,\ z : U}\}$, if there exists t with $\Gamma \vdash \mathsf{t : E}$, then the typing relation still holds by extending the environment with $\mathsf{b : B}$:

$$\mathsf{b : B,\ x : T,\ y : S,\ \ldots,\ z : U \vdash t : E}$$

The type U is pure and inhabited, as Γ is pure and inhabited. According to the definition of *inhabited type*, there exists a value u with $\mathsf{b : B,\ e : E \vdash u : U}$. As u is a value, it can be either a variable or a function. If u is a variable, it can only be b, as U is a pure type, thus it cannot be the capability type E. If u is a function, we have $\mathsf{b : B \vdash u : U}$, as in the typing of stoic functions the rule T-ABS will remove the binding $\mathsf{e : E}$ from the environment. In both cases, we have $\mathsf{b : B \vdash u : U}$. Now using the substitution lemma, we have:

$$\mathsf{b : B,\ x : T,\ y : S,\ \cdots \vdash [z \mapsto u]t : E}$$

Continue this process, we can reduce the typing environment to be $\{\mathsf{b : B}\}$ and the term to $\mathsf{t'}$:

$$\mathsf{b : B \vdash t' : E}$$

Now combining *progress* and *normalization* of STLC-Pure[5], $\mathsf{t'}$ can take finite evaluation steps to become a value v:

$$\mathsf{b : B \vdash v : E}$$

To summarize, we have given an intuitive proof of the following statement:

$$\neg\ \text{Effect-Safety-Inhabited} \rightarrow \neg\ \text{Effect-Safety-Intuitive}$$

By the logical law *contraposition*, we have:

$$\text{Effect-Safety-Intuitive} \rightarrow \text{Effect-Safety-Inhabited}$$

[5] We only proved that progress holds in an empty environment, it's easy to prove it also holds under $\{\mathsf{b : B}\}$. We didn't prove normalization of STLC-Pure, but the proof should be similar to the proof in standard STLC.

As *Effect-Safety-Intuitive* trivially holds, *Effect-Safety-Inhabited* follows by *modus ponens*.

Though conceptually simpler, the mechanized proof necessitates the proof of the normalization theorem, which is more involved than the proof based on capsafe environments. On the other hand, the approach based on capsafe environments works even if normalization doesn't hold. Therefore, we don't take the intuitive approach in the formal development.

3 System STLC-Impure

This chapter describes an extension of the system STLC-Pure with free functions. As stoic functions can be used as free functions, it's natural to integrate subtyping in the system.

We'll first introduce the formalization, then discuss soundness and effect safety. In the discussion, we'll focus on its difference from the system STLC-Pure.

3.1 Definitions

Initially, we arrived at a formulation of the system shown in Figure 3.1. It's a straight-forward extension of STLC-Pure with subtyping and free functions.

The definition is all good, except that preservation breaks. The problem is caused by typing an impure term as Top. To see a concrete example, let's assume $\Gamma = \{\mathsf{c} : \mathsf{E}\}$. It's obvious that the following typing relation holds:

$$\mathsf{c} : \mathsf{E} \vdash (\lambda \mathsf{x}{:}\mathsf{Top}.\, \lambda \mathsf{y}{:}\mathsf{B}.\, \mathsf{x})\ \mathsf{c}\ :\ \mathsf{B} \rightarrow \mathsf{Top}$$

However, after one evaluation step[6], we get the term $\lambda \mathsf{y}{:}\mathsf{B}.\, \mathsf{c}$, which can at best be typed as $\mathsf{B} \Rightarrow \mathsf{Top}$. Thus preservation breaks. This problem leads us to two different formulations.

The first one is to introduce two different Top types, $\mathsf{Top-Pure}$ and $\mathsf{Top-Impure}$, the former is pure and the latter is impure[7]. The capability type E and free function type $\mathsf{S} \Rightarrow \mathsf{T}$ are not subtypes of $\mathsf{Top-Pure}$. The subtyping hierarchy is shown in Figure 3.2. This formulation works well, and we've proved soundness and effect safety for the formulation. However, we lose the simplicity of the type system. And it's counter-intuitive to forbid variables of the type $\mathsf{Top-Impure}$ in pure environments, as we cannot create side effects with a variable of the type $\mathsf{Top-Impure}$.

The second possibility is to keep the elegance of the type system and change the evaluation rules. All terms of the type Top are equivalent, because we can do nothing with a term of the type Top. This observation inspires us to introduce a value top and replace the standard E-AppAbs rule with two evaluation rules as follows:

[6] Note that variables are values, thus we can take a step here. We can also construct a counter-example by wrapping c in a free function like $\lambda \mathsf{x}{:}\mathsf{B}.\, \mathsf{c}$.

[7] This idea is suggested by Sandro Stucki.

Syntax

t	$::=$		terms:
		x	variable
		$\lambda x{:}T.\, t$	abstraction
		$t\, t$	application
v	$::=$		values:
		$\lambda x{:}T.\, t$	abstraction value
		x	variable value
T	$::=$		types:
		Top	top type
		B	basic type
		E	capability type
		$T \rightarrow T$	type of stoic funs
		$T \Rightarrow T$	type of free funs

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\, t_2 \longrightarrow t_1'\, t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\, t_2 \longrightarrow v_1\, t_2'} \quad \text{(E-APP2)}$$

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

Pure Environment

$$\begin{aligned} \mathsf{pure}(\varnothing) &= \varnothing \\ \mathsf{pure}(\Gamma,\, x : E) &= \mathsf{pure}(\Gamma) \\ \mathsf{pure}(\Gamma,\, x : S \Rightarrow T) &= \mathsf{pure}(\Gamma) \\ \mathsf{pure}(\Gamma,\, x : T) &= \mathsf{pure}(\Gamma),\, x : T \end{aligned}$$

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

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

$$\frac{\mathsf{pure}(\Gamma),\, x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.\, t_2 : S \rightarrow T} \quad \text{(T-ABS1)}$$

$$\frac{\Gamma,\, x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.\, t_2 : S \Rightarrow T} \quad \text{(T-ABS2)}$$

$$\frac{\Gamma \vdash t_1 : S \Rightarrow T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\, t_2 : T} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

Subtyping $\boxed{S <: T}$

$$T <: \mathsf{Top} \quad \text{(S-TOP)}$$

$$T <: T \quad \text{(S-REFL)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(S-TRANS)}$$

$$S \rightarrow T <: S \Rightarrow T \quad \text{(S-DEGEN)}$$

$$\frac{S1 <: S2 \qquad T2 <: T1}{S2 \rightarrow T2 <: S1 \rightarrow T1} \quad \text{(S-FUN1)}$$

$$\frac{S1 <: S2 \qquad T2 <: T1}{S2 \Rightarrow T2 <: S1 \Rightarrow T1} \quad \text{(S-FUN2)}$$

Figure 3.1: System STLC-Impure First Formulation

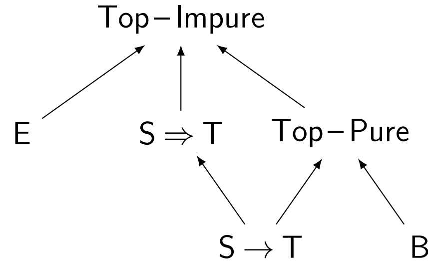

Figure 3.2: Subtyping: $\mathsf{Top-Pure}$ and $\mathsf{Top-Impure}$

$$\frac{\mathsf{T} \neq \mathsf{Top}}{(\lambda \mathsf{x}{:}\mathsf{T}.\mathsf{t}_1)\mathsf{v}_2 \longrightarrow [\mathsf{x} \mapsto \mathsf{v}_2]\mathsf{t}_1} \text{ (E-APPABS1)} \qquad \frac{\mathsf{T} = \mathsf{Top}}{(\lambda \mathsf{x}{:}\mathsf{T}.\mathsf{t}_1)\mathsf{v}_2 \longrightarrow [\mathsf{x} \mapsto \mathsf{top}]\mathsf{t}_1} \text{ (E-APPABS2)}$$

The two rules have the effect that if a function takes a parameter of the type Top, when called it will drop the passed parameter and replace it with the value top. There is no need to consider the case that the parameter type T is like $\mathsf{B} \rightarrow \mathsf{Top}$ or $(\mathsf{Top} \rightarrow \mathsf{B}) \Rightarrow \mathsf{B}$, as function types cannot mask impure terms as pure like Top does. We follow this approach in the formulation and the full definition is presented in Figure 3.3.

Note that in the current system, we need to change the definition of the function pure to exclude free function types from the pure environment. This restriction is important, because there's no way to know what side effects there might be inside free functions. If stoic functions have access to free functions, we'll loose the ability to track the effects of stoic functions in the type system.

However, mixed-arrow types are allowed in pure environments, as long as the first arrow is $\rightarrow$. For example, the type $\mathsf{E} \rightarrow \mathsf{B} \Rightarrow \mathsf{B}$ is allowed in the pure environment. As will be shown in the effect safety section, we have formally proved that having such mixed-arrow types in the pure environment doesn't enable pure stoic functions to produce side effects.

In principle, we can keep some free functions that can never be called in the pure environment, such as $(\mathsf{B} \rightarrow \mathsf{E}) \Rightarrow \mathsf{B}$, as it's impossible to get an actual instance of the inhabited type $\mathsf{B} \rightarrow \mathsf{E}$ in order to call this function. However, it's not useful to have functions in the environment if they can never be called. Thus, to simplify the system without sacrificing usability, we removed all free function types from the pure environment.

Syntax

t	::=		terms:
		top	top value
		x	variable
		$\lambda x{:}T.\,t$	abstraction
		$t\,t$	application
v	::=		values:
		$\lambda x{:}T.\,t$	abstraction value
		x	variable value
		top	top value
T	::=		types:
		Top	top type
		B	basic type
		E	capability type
		$T \to T$	type of stoic funs
		$T \Rightarrow T$	type of free funs

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\,t_2 \longrightarrow t_1'\,t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\,t_2 \longrightarrow v_1\,t_2'} \quad \text{(E-APP2)}$$

$$\frac{T \neq \mathsf{Top}}{(\lambda x{:}T.\,t_1)\,v_2 \longrightarrow [x \mapsto v_2]t_1} \quad \text{(E-APPABS1)}$$

$$\frac{T = \mathsf{Top}}{(\lambda x{:}T.\,t_1)\,v_2 \longrightarrow [x \mapsto \mathsf{top}]t_1} \quad \text{(E-APPABS2)}$$

Pure Environment

$$\begin{aligned}
\mathsf{pure}(\varnothing) &= \varnothing \\
\mathsf{pure}(\Gamma, x : E) &= \mathsf{pure}(\Gamma) \\
\mathsf{pure}(\Gamma, x : S \Rightarrow T) &= \mathsf{pure}(\Gamma) \\
\mathsf{pure}(\Gamma, x : T) &= \mathsf{pure}(\Gamma), x : T
\end{aligned}$$

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

$$\Gamma \vdash \mathsf{top} : \mathsf{Top} \quad \text{(T-TOP)}$$

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

$$\frac{\mathsf{pure}(\Gamma), x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.\,t_2 : S \to T} \quad \text{(T-ABS1)}$$

$$\frac{\Gamma, x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.\,t_2 : S \Rightarrow T} \quad \text{(T-ABS2)}$$

$$\frac{\Gamma \vdash t_1 : S \Rightarrow T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\,t_2 : T} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

Subtyping $\boxed{S <: T}$

$$T <: \mathsf{Top} \quad \text{(S-TOP)}$$

$$T <: T \quad \text{(S-REFL)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(S-TRANS)}$$

$$S \to T <: S \Rightarrow T \quad \text{(S-DEGEN)}$$

$$\frac{S1 <: S2 \qquad T2 <: T1}{S2 \to T2 <: S1 \to T1} \quad \text{(S-FUN1)}$$

$$\frac{S1 <: S2 \qquad T2 <: T1}{S2 \Rightarrow T2 <: S1 \Rightarrow T1} \quad \text{(S-FUN2)}$$

Figure 3.3: System STLC-Impure

3.2 Soundness

We proved both progress and preservation of the system.

THEOREM (Progress). If $\varnothing \vdash \mathsf{t} : \mathsf{T}$, then either t is a value or there is some t' with $\mathsf{t} \longrightarrow \mathsf{t}'$.

THEOREM (Preservation). If $\Gamma \vdash \mathsf{t} : \mathsf{T}$, and $\mathsf{t} \longrightarrow \mathsf{t}'$, then $\Gamma \vdash \mathsf{t}' : \mathsf{T}$.

As you can imagine, now we need two different substitution lemmas in the proof of preservation, corresponding to the two reduction rules.

LEMMA (Subsitution-Not-Top). If $\Gamma,\ \mathsf{x} : \mathsf{S} \vdash \mathsf{t} : \mathsf{T}$, $\mathsf{S} \neq \mathsf{Top}$, s is a value and $\Gamma \vdash \mathsf{s} : \mathsf{S}$, then $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t} : \mathsf{T}$.

LEMMA (Subsitution-Top). If $\Gamma,\ \mathsf{x} : \mathsf{Top} \vdash \mathsf{t} : \mathsf{T}$, then $\Gamma \vdash [\mathsf{x} \mapsto \mathsf{top}]\mathsf{t} : \mathsf{T}$.

We restrict s to be a value in the lemma *Substitution-Not-Top* for the same reason as in the system STLC-Pure.

3.3 Effect Safety

We follow the same approach as in the system STLC-Pure in the formulation of effect safety. The formulation is an extension of the definition of *capsafe* and *caprod* in STLC-Pure with free function types.

3.3.1 Formulation

The definitions of *inhabited type* and *inhabited environment* are the same as in the system STLC-Pure.

With the presence of free functions, the previous formulation of effect safety is not enough. We not only need to ensure that it's impossible to construct a term of the capability type E in pure and inhabited environments, but also need to ensure only stoic functions can be called in pure and inhabited environments. The two conditions together guarantee that there cannot be actual side effects inside a pure stoic function. Thus, we need two statements about effect safety.

DEFINITION (Effect-Safety-Inhabited-1). If Γ is a pure and inhabited environment, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

DEFINITION (Effect-Safety-Inhabited-2). If Γ is a pure and inhabited environment, and $\Gamma \vdash t_1\ t_2 : \mathsf{T}$, then there exists U, V such that $\Gamma \vdash t_1 : \mathsf{U} \to \mathsf{V}$.

A tempting formulation of the second effect safety statement would be that *in a pure and inhabited environment it's impossible to construct a term of the free function type*. However, this formulation has no hope to be proved, as $\mathsf{S} \to \mathsf{T}$ is a subtype of $\mathsf{S} \Rightarrow \mathsf{T}$, any term that can be typed as the former can also be typed as the latter.

As in the system STLC-Pure, the proof of these two statements depends on two more general statements about *capsafe environments*. If we can arrive at such a definition of *capsafe environment* that a pure and inhabited environment is also capsafe, then it suffices to prove following two statements about capsafe environments:

DEFINITION (Effect-Safety-1). If Γ is capsafe, there doesn't exist t with $\Gamma \vdash t : \mathsf{E}$.

DEFINITION (Effect-Safety-2). If Γ is capsafe and $\Gamma \vdash t_1\ t_2 : \mathsf{T}$, then there exists U, V such that $\Gamma \vdash t_1 : \mathsf{U} \to \mathsf{V}$.

Now we need to extend the definition of *capsafe* and *caprod* for free function types. Our first attempt is to add the following rule:

$$\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{caprod} \tag{CP-FUN2}$$

With this rule, the type $(\mathsf{B} \Rightarrow \mathsf{B}) \to \mathsf{E}$ would be considered *capsafe*, according to the rule CS-FUN1. However, with a variable f of this type and another variable g of the *capsafe* type $\mathsf{B} \to \mathsf{B}$ in the capsafe environment, the constructed term $f g$ has the capability type E; the first statement of effect safety breaks. This formulation also breaks the connection between pure inhabited environments and capsafe environments. A pure and inhabited environment is no longer capsafe. For example, the type $\mathsf{B} \to \mathsf{B} \Rightarrow \mathsf{B}$ is pure and inhabited, but it's not *capsafe*.

On the other hand, we cannot do the opposite to take $\mathsf{S} \Rightarrow \mathsf{T}$ as *capsafe*, as it would allow calling free functions in capsafe environments; the second statement of effect safety breaks. In the meanwhile, the pure and inhabited type $(\mathsf{B} \Rightarrow \mathsf{E}) \to \mathsf{E}$ is not *capsafe*, thus a pure and inhabited environment is no longer a capsafe environment.

This dilemma prompts us to reexamine the meaning of *capsafe* and *caprod*. When these facilities were first introduced in STLC-Pure, they were formulated in terms of the provability of the capability type E. From the perspective of the provability of E, $\mathsf{S} \to \mathsf{T}$ and $\mathsf{S} \Rightarrow \mathsf{T}$ don't make much difference. Thus, free function types should be formulated the same way as stoic function

types. We tried this approach, and it worked[8]. The full formulation is presented in Figure 3.4.

Capsafe Type

$$\mathsf{B} \quad \mathsf{capsafe} \qquad \text{(CS-BASE)}$$

$$\frac{\mathsf{S} \quad \mathsf{caprod}}{\mathsf{S} \to \mathsf{T} \quad \mathsf{capsafe}} \qquad \text{(CS-FUN1)}$$

$$\frac{\mathsf{T} \quad \mathsf{capsafe}}{\mathsf{S} \to \mathsf{T} \quad \mathsf{capsafe}} \qquad \text{(CS-FUN2)}$$

$$\frac{\mathsf{S} \quad \mathsf{caprod}}{\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{capsafe}} \qquad \text{(CS-FUN3)}$$

$$\frac{\mathsf{T} \quad \mathsf{capsafe}}{\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{capsafe}} \qquad \text{(CS-FUN4)}$$

Caprod Type

$$\mathsf{E} \quad \mathsf{caprod} \qquad \text{(CP-EFF)}$$

$$\frac{\mathsf{S}\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{caprod}}{\mathsf{S} \to \mathsf{T} \quad \mathsf{caprod}} \qquad \text{(CP-FUN1)}$$

$$\frac{\mathsf{S}\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{caprod}}{\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{caprod}} \qquad \text{(CP-FUN2)}$$

Capsafe Environment

$$\varnothing \quad \mathsf{capsafe} \qquad \text{(CE-EMPTY)}$$

$$\frac{\Gamma\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{capsafe}}{\Gamma, \mathsf{x} : \mathsf{T} \quad \mathsf{capsafe}} \qquad \text{(CE-VAR)}$$

Figure 3.4: System STLC-Impure Capsafe Environment

Now a capsafe environment is no longer pure. For example, the type $\mathsf{B} \Rightarrow \mathsf{B}$ is *capsafe*, but it's not pure. As a consequence, we need to slightly update the definition of the second statement of effect safety:

DEFINITION (Effect-Safety-2'). If Γ is pure and capsafe, and $\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}$, then there exists U, V such that $\Gamma \vdash \mathsf{t}_1 : \mathsf{U} \to \mathsf{V}$.

Why this formulation of capsafe environment is acceptable? In short, it's because the statement *Effect-Safety-1* and *Effect-Safety-2'* logically imply the statement *Effect-Safety-Inhabited-1* and *Effect-Safety-Inhabited-2* respectively.

The logical implications hold because a pure and inhabited environment is also a capsafe (and pure) environment. This claim has been formally proved:

LEMMA (Inhabited-Capsafe). If the type T is inhabited, then either T is capsafe or $\mathsf{T} = \mathsf{E}$ or T is a free function type.

[8] Sandro Stucki suggested to take this approach.

THEOREM (Inhabited-Pure-Capsafe-Env). If Γ is pure and inhabited, then Γ is capsafe.

COROLLARY (Inhabited-Pure-Capsafe-Env'). If Γ is pure and inhabited, then Γ is pure and capsafe.

Note that the last corollary follows immediately from the second theorem, as we already know from the premise that Γ is pure.

3.3.2 Proof

The proof of the first statement of effect safety is almost the same as in STLC-Pure. The only change worth mention is that, in the presence of subtyping, an additional lemma *Capsafe-Sub* is required. The first effect safety statement follows immediately from the lemma *Capsafe-Env-Capsafe*.

LEMMA (Capsafe-Not-Caprod). If type T is capsafe, then T is not caprod.

LEMMA (Capsafe-Or-Caprod). For any T, T is either capsafe or caprod.

LEMMA (Capsafe-Sub). If S is capsafe and $\mathsf{S} <: \mathsf{T}$, then T is capsafe.

LEMMA (Capsafe-Env-Capsafe). If Γ is capsafe and $\Gamma \vdash \mathsf{t} : \mathsf{T}$, then T is capsafe.

THEOREM (Effect-Safety-1). If Γ is capsafe, then there doesn't exist term t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

However, it's impossible to prove the second statement of effect safety, which states that if we can construct an application $\mathsf{t}_1\ \mathsf{t}_2$ in a pure and capsafe environment, then t_1 can be typed as $\mathsf{S} \to \mathsf{T}$ for some S and T. To see an example why it cannot be proved, let's assume $\Gamma = \{\mathsf{f} : \mathsf{B} \to \mathsf{B} \Rightarrow \mathsf{B},\ \mathsf{x} : \mathsf{B}\}$. It's obvious Γ is capsafe and pure. However, $\mathsf{f}\ \mathsf{x}$ in the term $(\mathsf{f}\ \mathsf{x})\ \mathsf{x}$ has the type $\mathsf{B} \Rightarrow \mathsf{B}$. It's impossible to prove that $\mathsf{f}\ \mathsf{x}$ has the type $\mathsf{B} \to \mathsf{B}$. If f is not a variable, but a fully defined function, then we can prove that f also has the type $\mathsf{B} \to \mathsf{B} \to \mathsf{B}$. This is because the inner function of the type $\mathsf{B} \Rightarrow \mathsf{B}$ cannot capture any capabilities or free functions. Otherwise, the outer function cannot be typed as stoic.

This observation leads us to assume four axioms listed in Figure 3.5. These axioms can only be proved if Γ is empty[9]. Otherwise, if t is a variable, we can do nothing.

The justification for the axiom AX-BASE is as follows. Suppose $\mathsf{t} = \lambda \mathsf{x}{:}\mathsf{B}.\, \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1$ and $\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \Rightarrow \mathsf{T}$. The typing rule for t should be the typing rule for stoic functions:

[9]Except the axiom AX-POLY, which cannot be proved even if Γ is empty.

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \to \mathsf{T}} \text{ (AX-BASE)} \qquad \frac{\Gamma \vdash \mathsf{t} : (\mathsf{U} \to \mathsf{V}) \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t} : (\mathsf{U} \to \mathsf{V}) \to \mathsf{S} \to \mathsf{T}} \text{ (AX-STOIC)}$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{Top} \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t} : \mathsf{Top} \to \mathsf{S} \to \mathsf{T}} \text{ (AX-TOP)} \qquad \frac{\Gamma \vdash \mathsf{t}_2 : \mathsf{U} \to \mathsf{V} \quad \Gamma \vdash \mathsf{t}_1 : (\mathsf{U} \Rightarrow \mathsf{V}) \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t}_1\, \mathsf{t}_2 : \mathsf{S} \to \mathsf{T}} \text{ (AX-POLY)}$$

Figure 3.5: System STLC-Impure Axioms

$$\frac{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B} \vdash \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1 : \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{B}.\, \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1 : \mathsf{B} \to \mathsf{S} \Rightarrow \mathsf{T}} \tag{STEP-1}$$

Then what's the rule used in the typing of $\lambda y{:}S.\, t_1$? If it's first typed as $S \to T$ and then subsumed as $S \Rightarrow T$, we are done. Otherwise, $\lambda y{:}S.\, t_1$ is typed using the rule of free functions:

$$\frac{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B},\ \mathsf{y} : \mathsf{S} \vdash \mathsf{t}_1 : \mathsf{T}}{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B} \vdash \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1 : \mathsf{S} \Rightarrow \mathsf{T}} \tag{STEP-2}$$

According to the definition of *pure*, we know that following two equations hold:

$$\begin{aligned} \mathsf{pure}(\mathsf{pure}(\Gamma)) &= \mathsf{pure}(\Gamma) \\ \mathsf{pure}(\Gamma,\ \mathsf{x} : \mathsf{B}) &= \mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B} \end{aligned}$$

Then we can obtain the following equation:

$$\mathsf{pure}(\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B}) = \mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B}$$

If we substitute the equation in Step-2, we get exactly the precondition for typing stoic functions. Thus $\lambda y{:}S.\, t_2$ can be typed as stoic function:

$$\frac{\mathsf{pure}(\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B}),\ \mathsf{y} : \mathsf{S} \vdash \mathsf{t}_1 : \mathsf{T}}{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B} \vdash \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1 : \mathsf{S} \to \mathsf{T}} \tag{STEP-2'}$$

Given that $\lambda y{:}S.\, t_1$ can be typed as $S \to T$, Step-1 can be updated as follows:

$$\frac{\mathsf{pure}(\Gamma),\ \mathsf{x} : \mathsf{B} \vdash \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1 : \mathsf{S} \to \mathsf{T}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{B}.\, \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t}_1 : \mathsf{B} \to \mathsf{S} \to \mathsf{T}} \tag{STEP-1'}$$

Put all the steps together, we obtained $\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \to \mathsf{T}$ from the fact $\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \Rightarrow \mathsf{T}$. That's the justification for the axiom AX-BASE. The justifications for AX-TOP and AX-STOIC are similar.

The axiom AX-POLY can be justified as follows. Suppose the following typing relation holds, what impure variables can be captured by t?

$$\Gamma \vdash \lambda \mathsf{f}{:}\mathsf{U} \Rightarrow \mathsf{V}.\, \lambda \mathsf{y}{:}\mathsf{S}.\, \mathsf{t} : (\mathsf{U} \Rightarrow \mathsf{V}) \to \mathsf{S} \Rightarrow \mathsf{T}.$$

As the function is stoic, the only impure variable can be captured is f. Now, if we supply a stoic function as parameter to the function, it has the same effect as saying that f has the type $\mathsf{U} \to \mathsf{V}$. Thus, in this context, the function can be typed as $(\mathsf{U} \to \mathsf{V}) \to \mathsf{S} \Rightarrow \mathsf{T}$. Now according to the axiom AX-STOIC, the term can also be typed as $(\mathsf{U} \to \mathsf{V}) \to \mathsf{S} \to \mathsf{T}$. Finally by the rule T-APP, the type of the application is $\mathsf{S} \to \mathsf{T}$. That's the justification for the axiom AX-POLY.

Assuming the axioms above, it's straight-forward to prove a lemma *Capsafe-Pure-Stoic*, and the second statement of effect safety follows immediately from the lemma.

LEMMA (Capsafe-Pure-Stoic). If Γ is pure and capsafe, and $\Gamma \vdash \mathsf{t} : \mathsf{S} \Rightarrow \mathsf{T}$, then $\Gamma \vdash \mathsf{t} : \mathsf{S} \to \mathsf{T}$.

THEOREM (Effect-Safety-2'). If Γ is pure and capsafe, and $\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}$, then there exists U, V such that $\Gamma \vdash \mathsf{t}_1 : \mathsf{U} \to \mathsf{V}$.

3.4 Effect Polymorphism

There are three kinds of effect polymorphism in capability-based effect systems, namely axiomatic polymorphism, curry polymorphism and stoic polymorphism. The first one depends on the axiom AX-POLY, while the other two are inherent in capability-based effect systems.

3.4.1 Axiomatic Polymorphism

Axiomatic polymorphism depends on the axiom AX-POLY. It can be illustrated by the following example:

```
def map[A,B](f: A => B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
}
def squareImpure(c: IO) = map { x => println(x)(c); x*x }
```

```
def squarePure = map { x => x*x }
```

In the code above, the function map has the type $(\mathsf{A} \Rightarrow \mathsf{B}) \rightarrow \mathsf{List[A]} \Rightarrow \mathsf{List[B]}$. Without the axiom AX-POLY, the function squareImpure would be typed as $\mathsf{IO} \rightarrow \mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$, while the function squarePure would be typed as $\mathsf{List[Int]} \Rightarrow \mathsf{List[Int]}$. Effect polymorphism doesn't work in this case, as squarePure can actually be typed as a stoic function.

In the presence of the axiom AX-POLY, squarePure can be typed as $\mathsf{List[Int]} \rightarrow \mathsf{List[Int]}$, as the function passed to map has the stoic function type $\mathsf{Int} \rightarrow \mathsf{Int}$. Now both squarePure and squareImpure are typed as stoic functions, all effects are captured in the type system even in the presence of (local) free functions.

The reader might be wondering, what if we want to stipulate that the passed function f can only have IO effects? How does effect polymorphism work in such cases? The answer is as follows:

```
private def mapImpl[A,B](f: A => B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
}

def map[A,B](f: A -> B) = mapImpl(f)

def map[A,B](c: IO)(f: IO -> A => B) = mapImpl(f c)
```

In the code above, we protect the implementation function mapImpl with the keyword private. All callings of the map function has to go through the two exposed interfaces. The two interfaces impose that the passed function f must either be pure or only have IO side effects. It's straightforward to add a new interface to allow exception effects without changing the implementation or affecting existing code.

3.4.2 Currying Polymorphism

There is another kind of effect polymorphism inherent in capability-based effect systems without resorting to the axiom AX-POLY. This kind of polymorphism is related to *currying*, thus is called *currying polymorphism*. It can be demonstrated by the following example:

```
def map[A,B](f: A => B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
```

```
}
def squareImpure(c: IO) = map { x => println(x)(c); x*x }
def squarePure(l: List[Int]) = map { x => x*x } l
```

Note that the only change we made compared to the example in *axiomatic polymorphism* is to add the parameter l : List[Int] explicitly to squarePure, instead of resorting to currying. This seemingly trivial change makes a big difference in capability-based effect systems. Now squarePure can be typed as List[Int] $\to$ List[Int] without using the axiom AX-POLY.

3.4.3 Stoic Polymorphism

Stoic functions that take a free function and return a value of a pure type are inherently effect-polymorphic. This kind of effect polymorphism is called *stoic polymorphism*. Stoic polymorphism can be illustrated by the following example:

```
def twice(f: Int => Int) = f (f 0)
def pure(x: Int) = twice { n => n + x }
def impure(x: Int)(c: IO) = twice { n => println(n)(c); n + x }
```

In the code above, the function twice is typed as $(\text{Int} \Rightarrow \text{Int}) \to \text{Int}$. By just checking the type signature of twice, we know it might have side effects, if the passed function f has side effects. Without adding any annotation or resorting to any axiom, the type system automatically types the function pure as Int $\to$ Int and impure as Int $\to$ IO $\to$ Int. Effect polymorphism works naturally.

3.4.4 Discussion

As reported in section 1.6 of the thesis[Lip09a], 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. For example, in Haskell it requires two versions of the function map , but in capability-based effect systems, only one effect-polymorphic map is required, as the following code shows:

```
-- monad-based effect system
map :: (a -> b) -> List a -> List b
mapM :: Monad m => (a -> m b) -> List a -> m (List b)
-- capability-based effect system
```

```
map :: (a => b) -> List a => List b
```

In Haskell, it's possible to implement the non-monadic version based on the monadic version by using the identity monad:

```
map :: (a -> b) -> List a -> List b
map f xx = runIdentity (mapM (\x -> return (f x)) xx)
```

However, in practice programmers usually first come up with the non-monadic version, and later realize the need for a monadic version. Turning the non-monadic code into the monadic version requires almost a rewrite of the function, as demonstrated by the following code:

```
map :: (a -> b) -> List a -> List b
map f xs
  = case xs of
      Nil -> Nil
      Cons x xs -> Cons (f x) (map f xs)

mapM :: Monad m => (a -> m b) -> List a -> m (List b)
mapM f xs
  = case xs of
    Nil -> return Nil
    Cons x xs -> do x' <- f x
      xs' <- mapM f xs
      return (Cons x' xs')
```

In capability-based effect systems, if programmers first come up with the pure version, turning it to the effect-polymorphic version only requires changing $\to$ to $\Rightarrow$, as demonstrated by the following code:

```
// pure version
def map[A,B](f: A -> B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
}
// effect-polymorphic version
def map[A,B](f: A => B)(l: List[A]) = l match {
  case Nil => Nil
  case x::xs => f(x)::map(f)(xs)
}
```

Monads are not only heavier in handling effect polymorphism, but also more difficult to learn and use than capabilities. The concept *capability* has an intuitive meaning as it has in daily life, while the abstruse concept *monad* comes from category theory, which is the most abstract field of mathematics. Therefore, we expect capability-based effect systems to be welcomed by a larger audience than monad-based effect systems.

Type-and-effect systems based on type-and-effect inference[TJ92, TJ94] can greatly reduce the syntactical overhead in effect systems, but it still complicates the type signature of effect-polymorphic functions by introducing generic effect type variables. For example, the type signature inferred for the effect-polymorphic function map would look like $\forall \mathsf{E}.(\mathsf{A} \to \mathsf{B@E}) \to \mathsf{List[A]} \to \mathsf{List[B]@E}$, which is a little daunting to programmers, compared to $(\mathsf{A} \Rightarrow \mathsf{B}) \to \mathsf{List[A]} \Rightarrow \mathsf{List[B]}$ in our case.

Effect polymorphism is an advantage of capability-based effect systems over other kinds of effect systems. In capability-based effect systems, effect polymorphism can be easily achieved without introducing generic effect type variables in the type signature or depending on additional annotations.

4 System F-Pure

In this chapter, we present the system F-Pure, which is an extension of the system STLC-Pure with universal types. In this system, not only functions need to observe a variable-capturing discipline, type abstractions also need to observe the same variable-capturing discipline.

We'll first introduce the formalization, then discuss soundness and effect safety. In the discussion, we'll focus on its difference from the system STLC-Pure.

4.1 Definitions

The system F-Pure extends STLC-Pure with universal types. Figure 4.1 presents the full definition of F-Pure, with the difference from the system STLC-Pure highlighted.

The extension of syntax and evaluation rules are exactly the same as the extension of standard STLC with universal types. The essential difference lies in the two new typing rules T-TABS and T-TAPP. The typing rule T-TABS stipulates that type abstraction must observe the variable-capturing discipline.

$$\frac{\mathsf{pure}(\Gamma),\ X \vdash t_2 : T}{\Gamma \vdash \lambda X.\, t_2 : \forall X.T} \qquad \text{(T-TABS)}$$

We made this design choice in order to allow universal types to be present in pure environments. Otherwise, if type abstractions can capture capability variables, application of a type abstraction could generate a term of the capability type or have side effects. This makes it incorrect to have universal types in pure environments, thus renders universal types useless in the system.

The typing rule T-TAPP requires that the type parameter cannot be the capability type E. However, it's allowed to supply uninhabited types like $B \to E$ as parameter to type abstraction.

$$\frac{\Gamma \vdash t_1 : \forall X.T \qquad T_2 \neq E}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T} \qquad \text{(T-TAPP)}$$

Without the restriction, preservation of the system breaks [10], as can be seen from the following term t, which has the type $\forall X.X \to B \to X$:

[10] It's also possible to allow E as argument to type application, and restore preservation by treating variables with the type of a type variable as impure. We'll follow this approach in the system F-Impure.

Syntax

t	::=		terms:
		x	variable
		$\lambda x{:}T.t$	abstraction
		tt	application
		$\lambda X.t$	type abstraction
		$t[T]$	type application
v	::=		values:
		$\lambda x{:}T.t$	abstraction value
		x	variable value
		$\lambda X.t$	type abstraction value
T	::=		types:
		X	type variable
		B	basic type
		E	capability type
		$T \to T$	type of functions
		$\forall X.T$	universal type

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X.t_1)[T_2] \longrightarrow [X \mapsto T_2]t_1 \quad \text{(E-TAPPTABS)}$$

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

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

$$\frac{\text{pure}(\Gamma),\ x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.t_2 : S \to T} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : S \to T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T} \quad \text{(T-APP)}$$

$$\frac{\text{pure}(\Gamma),\ X \vdash t_2 : T}{\Gamma \vdash \lambda X.t_2 : \forall X.T} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T \qquad T_2 \neq E}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T} \quad \text{(T-TAPP)}$$

Pure Environment

$\text{pure}(\varnothing)$	=	$\varnothing$
$\text{pure}(\Gamma, x : E)$	=	$\text{pure}(\Gamma)$
$\text{pure}(\Gamma, x : T)$	=	$\text{pure}(\Gamma), x : T$
$\text{pure}(\Gamma, X)$	=	$\text{pure}(\Gamma), X$

Figure 4.1: System F-Pure

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

If we allow E as parameter to type application, the term $t\ [E]$ would have the type $E \to B \to E$. However, after one evaluation step, the term $\lambda x{:}E.\, \lambda y{:}B.\, x$ cannot be typed anymore, as the capability variable x cannot be captured in the inner-most lambda; thus preservation breaks.

The definition of the function pure is changed slightly by allowing type variables to be in the pure environment. Type variables themselves are harmless to effect safety, their presence in the pure environment ensures the well-formedness of the environment.

Note that a hidden change to the function pure is that the binding $x : X$, where X is a type variable, may appear in pure environments. This is natural, as we know in the typing rule T-TAPP that the type variable X cannot be the capability type E.

4.2 Soundness

We proved both progress and preservation of the system.

THEOREM (Progress). If $\varnothing \vdash t : T$, then either t is a value or there is some t' with $t \longrightarrow t'$.

THEOREM (Preservation). If $\Gamma \vdash t : T$, and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

The proof of progress is the same as in System F. In the proof of preservation, we need to make small changes to the standard substitution lemmas in System F.

LEMMA (Subsitution-Term). If $\Gamma,\ x : S \vdash t : T$, s is a value and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

LEMMA (Subsitution-Type). If $\Gamma,\ X \vdash t : T$ and $P \neq E$, then $\Gamma \vdash [X \mapsto P]t : [X \mapsto P]T$.

We restrict s to be a value in the lemma *Substitution-Term* for the same reason as in the system STLC-Pure. In the lemma *Substitution-Type*, we restrict that P is not the capability type E. Otherwise, the lemma cannot be proved as explained in the previous section.

4.3 Effect Safety

We follow the same approach as in the system STLC-Pure in the formulation of effect safety.

4.3.1 Inhabited Types and Environments

In the presence of universal types, we need to adapt the definition of inhabited type and inhabited environment.

The definition of inhabited environment in STLC-Pure would reject a well-formed environment with type variables like $\{\mathsf{T},\ \mathsf{S},\ \mathsf{f} : \mathsf{T} \to \mathsf{S},\ \mathsf{x} : \mathsf{T}\}$, which could be the environment for the following well-defined function:

```
def apply[T, S](f: T -> S)(x: T) = f x
```

To handle this problem, we propose the definition of *inhabited type* and *inhabited environment* as shown in Figure 4.2.

Primitive Environment

$$\mathsf{x} : \mathsf{B},\ \mathsf{y} : \mathsf{E} \quad \text{primitive} \quad \text{(P-BASE)}$$

$$\frac{\Sigma \quad \text{primitive}}{\Sigma,\ \mathsf{X} \quad \text{primitive}} \quad \text{(P-TVAR)}$$

$$\frac{\Sigma \quad \text{primitive}}{\Sigma,\ \mathsf{x} : \mathsf{X} \quad \text{primitive}} \quad \text{(P-TYPE)}$$

Inhabited Type

$$\frac{\Sigma\ \text{primitive} \qquad \Sigma \vdash \mathsf{v} : \mathsf{T}}{\mathsf{T} \quad \text{inhabited}} \quad \text{(IT)}$$

Inhabited Environment

$$\varnothing \quad \text{inhabited} \quad \text{(IE-EMPTY)}$$

$$\frac{\Gamma \quad \text{inhabited}}{\Gamma,\ \mathsf{X} \quad \text{inhabited}} \quad \text{(IE-TVAR)}$$

$$\frac{\Gamma\ \text{inhabited} \qquad \mathsf{T}\ \text{inhabited}}{\Gamma,\ \mathsf{x} : \mathsf{T} \quad \text{inhabited}} \quad \text{(IE-TYPE)}$$

Figure 4.2: System F-Pure Inhabited Environment

The definition of inhabited type depends on *primitive environments*. A primitive environment is an extension of the environment $\{\mathsf{x} : \mathsf{B}, \mathsf{y} : \mathsf{E}\}$ with any type variables and type variable bindings. This definition would take types like X, $\mathsf{X} \to \mathsf{Y}$ as inhabited, as expected.

The definition of inhabited environment not only allow bindings of inhabited types, but also allow any type variables to be present in an inhabited environment. This definition ensures that only uninhabited types, such as $\mathsf{B} \to \mathsf{E}$, $\forall \mathsf{X}.\mathsf{X}$ and $\forall \mathsf{X}.\forall \mathsf{Y}.\mathsf{X} \to \mathsf{Y}$ are rejected from pure environments.

4.3.2 Formulation

As in STLC, the standard formulation is given based on inhabited environments:

DEFINITION (Effect-Safety-Inhabited). If Γ is a pure and inhabited environment, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

The proof of the statement depends on a more general statement about *capsafe environments*. If we can arrive at such a definition of *capsafe environment* that a pure and inhabited environment is also capsafe, then it suffices to prove the following statement about capsafe environments:

DEFINITION (Effect-Safety). If Γ is capsafe, there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

What *capsafe* and *caprod* rules we need for universal types? Obviously, we need to take the uninhabited type $\forall \mathsf{X}.\mathsf{X}$ as *caprod*, as with a variable of this type, it's possible to create a term of the capability type E. For example, if x is of the type $\forall \mathsf{X}.\mathsf{X}$ and b is of the type B, then $\mathsf{x}\,[\mathsf{B} \to \mathsf{E}]\,\mathsf{b}$ has the type E. We also need to take the uninhabited type $\forall \mathsf{X}.\forall \mathsf{Y}.\mathsf{X} \to \mathsf{Y}$ as *caprod*. Otherwise, if x is of the type $\forall \mathsf{X}.\forall \mathsf{Y}.\mathsf{X} \to \mathsf{Y}$ and b is of the type B, then $\mathsf{x}\,[\mathsf{B}]\,[\mathsf{B} \to \mathsf{E}]\,\mathsf{b}\,\mathsf{b}$ has the type E. This observation leads us to the definition of *capsafe environment* presented in Figure 4.3, with differences from STLC-Pure highlighted.

Why this formulation of capsafe environment is acceptable? In short, it's because the statement *Effect-Safety* logically implies the statement *Effect-Safety-Inhabited*.

The logical implication holds because a pure and inhabited environment is also a capsafe environment. This claim has been formally proved:

LEMMA (Inhabited-Capsafe). If the type T is inhabited, then either T is capsafe or $\mathsf{T} = \mathsf{E}$.

THEOREM (Inhabited-Pure-Capsafe-Env). If Γ is pure and inhabited, then Γ is also capsafe.

4.3.3 Proof

The proof of effect safety is more involved than in STLC-Pure. We need to introduce the *degree* of types in the proof of relevant lemmas about types.

Capsafe Type

$$\mathsf{B} \quad \mathsf{capsafe} \qquad (\text{CS-BASE})$$

$$\mathsf{X} \quad \mathsf{capsafe} \qquad (\text{CS-VAR})$$

$$\frac{\mathsf{S} \quad \mathsf{caprod}}{\mathsf{S} \to \mathsf{T} \quad \mathsf{capsafe}} \ (\text{CS-FUN1})$$

$$\frac{\mathsf{T} \quad \mathsf{capsafe}}{\mathsf{S} \to \mathsf{T} \quad \mathsf{capsafe}} \ (\text{CS-FUN2})$$

$$\frac{[\mathsf{X} \mapsto \mathsf{B}]\mathsf{T}\ \mathsf{capsafe} \qquad [\mathsf{X} \mapsto \mathsf{E}]\mathsf{T}\ \mathsf{capsafe}}{\forall \mathsf{X}.\mathsf{T} \quad \mathsf{capsafe}} \ (\text{CS-ALL})$$

Caprod Type

$$\mathsf{E} \quad \mathsf{caprod} \qquad (\text{CP-EFF})$$

$$\frac{\mathsf{S}\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{caprod}}{\mathsf{S} \to \mathsf{T} \quad \mathsf{caprod}} \ (\text{CP-FUN})$$

$$\frac{[\mathsf{X} \mapsto \mathsf{B}]\mathsf{T}\ \mathsf{caprod}}{\forall \mathsf{X}.\mathsf{T} \quad \mathsf{caprod}} \ (\text{CP-ALL1})$$

$$\frac{[\mathsf{X} \mapsto \mathsf{E}]\mathsf{T}\ \mathsf{caprod}}{\forall \mathsf{X}.\mathsf{T} \quad \mathsf{caprod}} \ (\text{CP-ALL2})$$

Capsafe Environment

$$\varnothing \quad \mathsf{capsafe} \qquad (\text{CE-EMPTY})$$

$$\frac{\Gamma\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{capsafe}}{\Gamma,\ \mathsf{x} : \mathsf{T} \quad \mathsf{capsafe}} \ (\text{CE-VAR})$$

$$\frac{\Gamma \quad \mathsf{capsafe}}{\Gamma,\ \mathsf{X} \quad \mathsf{capsafe}} \ (\text{CE-TVAR})$$

Figure 4.3: System F-Pure Capsafe Environment

DEFINITION (Degree of Type). The degree of a type T is defined as follows:

$$\mathsf{degree}(\mathsf{T}) = \begin{cases} \mathsf{max}(\mathsf{degree}(\mathsf{t}_1), \mathsf{degree}(\mathsf{t}_2)) & \text{if } \mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2, \\ \mathsf{degree}(\mathsf{T}_1) + 1 & \text{if } \mathsf{T} = \forall \mathsf{X}.\mathsf{T}_1, \\ 0 & \text{others} \end{cases}$$

With the help of the definition above, it's possible to prove following lemmas based on a nested induction on the degree of types and the type T.

LEMMA (Capsafe-Not-Caprod). If type T is capsafe, then T is not caprod.

LEMMA (Capsafe-Or-Caprod). For any type T, T is either capsafe or caprod.

LEMMA (Capsafe-All-Subst). If $\forall \mathsf{X}.\mathsf{T}$ is capsafe, then for any type U, $[\mathsf{X} \mapsto \mathsf{U}]\mathsf{T}$ is capsafe.

To prove the lemma *Capsafe-Env-Capsafe*, we need a similar definition on terms, and then do a nested induction on the degree of terms and the typing relation.

DEFINITION (Degree of Term). The degree of a term t is defined as follows:

$$\mathsf{degree}(\mathsf{t}) = \begin{cases} \mathsf{degree}(\mathsf{t}_1) & \text{if } \mathsf{t} = \lambda \mathsf{x}{:}\mathsf{T}.\, \mathsf{t}_1, \\ \max(\mathsf{degree}(\mathsf{t}_1), \mathsf{degree}(\mathsf{t}_2)) & \text{if } \mathsf{t} = \mathsf{t}_1\ \mathsf{t}_2, \\ \mathsf{degree}(\mathsf{t}_1) + 1 & \text{if } \mathsf{t} = \lambda \mathsf{X}.\, \mathsf{t}_1, \\ \mathsf{degree}(\mathsf{t}_1) & \text{if } \mathsf{t} = \mathsf{t}_1\ [\mathsf{T}], \\ 0 & \text{others} \end{cases}$$

Effect safety follows immediately from the lemma *Capsafe-Env-Capsafe*.

LEMMA (Capsafe-Env-Capsafe). If Γ is capsafe and $\Gamma \vdash \mathsf{t} : \mathsf{T}$, then T is capsafe.

THEOREM (Effect-Safety). If Γ is capsafe, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

5 System F-Impure

In this chapter, we present the system F-Impure, which is an extension of the system F-Pure with free functions. It can also be seen as an extension of the system STLC-Impure with universal types, but without subtyping. Extending the system with subtyping would lead us to bounded quantification, which we are still working on. Given the importance of parametric polymorphism and the fact that subtyping is not a necessary add-on of functional programming, the system F-Impure deserves a separate presentation here.

We'll first introduce the formalization, then discuss soundness and effect safety. In the discussion, we'll focus on its difference from the system STLC-Impure and F-Pure.

5.1 Definitions

Figure 5.1 presents the full definition of F-Impure, with the difference from the system F-Impure highlighted. As can be seen from the figure, we introduced free function types and added a typing rule for free functions. We have to add a typing rule T-DEGEN to restore the subtyping relation between stoic function types and free function types, as there is no subtyping in the current system.

As in STLC-Impure, we adapted the definition of pure to exclude free function types from pure environments. If stoic functions have access to free functions, we'll loose the ability to track the effects of stoic functions in the type system.

Different from F-Pure, in the rule T-TAPP we allow both capability types and free function types as type parameter in type application. We know in F-pure, without such restriction preservation of the system breaks. To restore preservation of the system, in the definition of pure we remove $x : X$ from the pure environment. That is, we treat variables typed with type variable as impure, which cannot be captured by a stoic function.

We could also introduce free type abstractions in the system. Such an extension will not be very useful in real-world programming, as in practice polymorphic functions rarely capture free variables, not mention capability variables. Thus for the sake of simplicity, we don't pursue the extension.

Syntax

t	::=		terms:
		x	variable
		$\lambda x{:}T.\,t$	abstraction
		$t\,t$	application
		$\lambda X.\,t$	type abstraction
		$t[T]$	type application
v	::=		values:
		$\lambda x{:}T.\,t$	abstraction value
		x	variable value
		$\lambda X.\,t$	type abstraction value
T	::=		types:
		X	type variable
		B	basic type
		E	capability type
		$T \rightarrow T$	type of stoic funs
		$T \Rightarrow T$	type of free funs
		$\forall X.T$	universal type

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X.\,t_1)[T_2] \longrightarrow [X \mapsto T_2]t_1 \quad \text{(E-TAPPTABS)}$$

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

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

$$\frac{\mathsf{pure}(\Gamma),\ x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.\,t_2 : S \rightarrow T} \quad \text{(T-ABS1)}$$

$$\frac{\Gamma,\ x : S \vdash t_2 : T}{\Gamma \vdash \lambda x{:}S.\,t_2 : S \Rightarrow T} \quad \text{(T-ABS2)}$$

$$\frac{\Gamma \vdash t : S \rightarrow T}{\Gamma \vdash t : S \Rightarrow T} \quad \text{(T-DEGEN)}$$

$$\frac{\Gamma \vdash t_1 : S \Rightarrow T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T} \quad \text{(T-APP)}$$

$$\frac{\mathsf{pure}(\Gamma),\ X \vdash t_2 : T}{\Gamma \vdash \lambda X.\,t_2 : \forall X.T} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T} \quad \text{(T-TAPP)}$$

Pure Environment

$\mathsf{pure}(\varnothing)$	=	$\varnothing$
$\mathsf{pure}(\Gamma,\ x : E)$	=	$\mathsf{pure}(\Gamma)$
$\mathsf{pure}(\Gamma,\ x : X)$	=	$\mathsf{pure}(\Gamma)$
$\mathsf{pure}(\Gamma,\ x : S \Rightarrow T)$	=	$\mathsf{pure}(\Gamma)$
$\mathsf{pure}(\Gamma,\ x : T)$	=	$\mathsf{pure}(\Gamma),\ x : T$
$\mathsf{pure}(\Gamma,\ X)$	=	$\mathsf{pure}(\Gamma),\ X$

Figure 5.1: System F-Impure

5.2 Soundness

We proved both progress and preservation of the system. The subtleties in the proof are the same as stated in the system F-Pure.

THEOREM (Progress). If $\varnothing \vdash \mathsf{t} : \mathsf{T}$, then either t is a value or there is some t' with $\mathsf{t} \longrightarrow \mathsf{t}'$.

THEOREM (Preservation). If $\Gamma \vdash \mathsf{t} : \mathsf{T}$, and $\mathsf{t} \longrightarrow \mathsf{t}'$, then $\Gamma \vdash \mathsf{t}' : \mathsf{T}$.

5.3 Effect Safety

We first introduce the formulation, which is a combination of the formulation in STLC-Impure and F-Pure, then discuss the proof of effect safety.

5.3.1 Formulation

The definitions of *inhabited type* and *inhabited environment* are the same as in the system F-Pure.

As in the system STLC-Impure, in the presence of free functions, we need two statements of effect safety:

DEFINITION (Effect-Safety-Inhabited-1). If Γ is a pure and inhabited environment, then there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

DEFINITION (Effect-Safety-Inhabited-2). If Γ is a pure and inhabited environment, and $\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}$, then there exists U, V such that $\Gamma \vdash \mathsf{t}_1 : \mathsf{U} \to \mathsf{V}$.

As in the system STLC-Impure, the proof of these two statements depends on two more general statements about *capsafe environments*. Given that we've seen how universal types and free function types are extended in the formulation of *capsafe environment*, we can easily combine them to arrive at the formulation shown in Figure 5.2, with the changes from F-Pure highlighted. Notice that type variables are now treated as caprod (CP-VAR).

Why this formulation of capsafe environment is acceptable? In short, it's because the statement *Effect-Safety-Inhabited-1* and *Effect-Safety-Inhabited-2* are logically implied by the statement *Effect-Safety-1* and *Effect-Safety-2* respectively.

DEFINITION (Effect-Safety-1). If Γ is capsafe, there doesn't exist t with $\Gamma \vdash \mathsf{t} : \mathsf{E}$.

Capsafe Type

$$\mathsf{B} \quad \mathsf{capsafe} \qquad \text{(CS-BASE)}$$

$$\frac{\mathsf{S} \quad \mathsf{caprod}}{\mathsf{S} \rightarrow \mathsf{T} \quad \mathsf{capsafe}} \quad \text{(CS-FUN1)}$$

$$\frac{\mathsf{T} \quad \mathsf{capsafe}}{\mathsf{S} \rightarrow \mathsf{T} \quad \mathsf{capsafe}} \quad \text{(CS-FUN2)}$$

$$\frac{\mathsf{S} \quad \mathsf{caprod}}{\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{capsafe}} \quad \text{(CS-FUN3)}$$

$$\frac{\mathsf{T} \quad \mathsf{capsafe}}{\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{capsafe}} \quad \text{(CS-FUN4)}$$

$$\frac{[\mathsf{X} \mapsto \mathsf{B}]\mathsf{T}\ \mathsf{capsafe} \qquad [\mathsf{X} \mapsto \mathsf{E}]\mathsf{T}\ \mathsf{capsafe}}{\forall \mathsf{X}.\mathsf{T} \quad \mathsf{capsafe}} \quad \text{(CS-ALL)}$$

Caprod Type

$$\mathsf{E} \quad \mathsf{caprod} \qquad \text{(CP-EFF)}$$

$$\mathsf{X} \quad \mathsf{caprod} \qquad \text{(CP-VAR)}$$

$$\frac{\mathsf{S}\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{caprod}}{\mathsf{S} \rightarrow \mathsf{T} \quad \mathsf{caprod}} \quad \text{(CP-FUN1)}$$

$$\frac{\mathsf{S}\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{caprod}}{\mathsf{S} \Rightarrow \mathsf{T} \quad \mathsf{caprod}} \quad \text{(CP-FUN2)}$$

$$\frac{[\mathsf{X} \mapsto \mathsf{B}]\mathsf{T}\ \mathsf{caprod}}{\forall \mathsf{X}.\mathsf{T} \quad \mathsf{caprod}} \quad \text{(CP-ALL1)}$$

$$\frac{[\mathsf{X} \mapsto \mathsf{E}]\mathsf{T}\ \mathsf{caprod}}{\forall \mathsf{X}.\mathsf{T} \quad \mathsf{caprod}} \quad \text{(CP-ALL2)}$$

Capsafe Environment

$$\varnothing \quad \mathsf{capsafe} \qquad \text{(CE-EMPTY)}$$

$$\frac{\Gamma\ \mathsf{capsafe} \qquad \mathsf{T}\ \mathsf{capsafe}}{\Gamma,\ \mathsf{x} : \mathsf{T} \quad \mathsf{capsafe}} \quad \text{(CE-VAR)}$$

$$\frac{\Gamma \quad \mathsf{capsafe}}{\Gamma,\ \mathsf{X} \quad \mathsf{capsafe}} \quad \text{(CE-TVAR)}$$

Figure 5.2: System F-Impure Capsafe Environment

DEFINITION (Effect-Safety-2). If Γ is pure and capsafe, and $\Gamma \vdash t_1\ t_2 : T$, then there exists U, V such that $\Gamma \vdash t_1 : U \rightarrow V$.

The logical implications hold because a pure and inhabited environment is also a capsafe (and pure) environment. This claim has been formally proved:

LEMMA (Inhabited-Capsafe). If the type T is inhabited, then either T is capsafe or $T = E$ or T is a free function type.

THEOREM (Inhabited-Pure-Capsafe-Env). If Γ is pure and inhabited, then Γ is also capsafe.

COROLLARY (Inhabited-Pure-Capsafe-Env'). If Γ is pure and inhabited, then Γ is pure and capsafe.

Note that the last corollary follows immediately from the second theorem, as we already know from the premise that Γ is pure.

5.3.2 Proof

The proof of the first effect safety statement is almost the same as in the system F-Pure, thus we omit here.

The proof of the second statement of effect safety faces the same problem as in the system STLC-Impure. We need to assume a set of axioms, as shown in Figure 5.3, with the newly added axioms highlighted. The justification for the axiom AX-ALL is the same as the justification for the axiom AX-BASE in STLC-Impure. In short, because the outer function is stoic and the type of the first parameter is pure, the inner function cannot capture variables of capabilities or free functions, thus it's fair enough to type the inner function as stoic.

The justification for the axiom AX-TABS is similar. If a term t can be typed as $\forall X.T_1 \Rightarrow T_2$ under Γ, according to the typing rule T-ALL, the whole term can be typed under $\text{pure}(\Gamma)$. Then the inner function can be typed under $\text{pure}(\Gamma), X$, which is equal to $\text{pure}(\Gamma, X)$. Thus, it's fair enough to type the inner function as stoic.

The justification for the axiom AX-POLY is the same as given in STLC-Impure, thus we omit here.

Assuming these axioms, it's straight-forward to prove a lemma *Capsafe-Pure-Stoic*, and the second statement of effect safety follows immediately from the lemma.

LEMMA (Capsafe-Pure-Stoic). If Γ is pure and capsafe, and $\Gamma \vdash t : S \Rightarrow T$, then $\Gamma \vdash t : S \rightarrow T$.

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t} : \mathsf{B} \to \mathsf{S} \to \mathsf{T}} \text{ (Ax-Base)}$$

$$\frac{\Gamma \vdash \mathsf{t} : \forall \mathsf{X}.(\mathsf{T}_1 \Rightarrow \mathsf{T}_2)}{\Gamma \vdash \mathsf{t} : \forall \mathsf{X}.(\mathsf{T}_1 \to \mathsf{T}_2)} \text{ (Ax-TAbs)}$$

$$\frac{\Gamma \vdash \mathsf{t} : (\forall \mathsf{X}.\mathsf{T}) \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t} : (\forall \mathsf{X}.\mathsf{T}) \to \mathsf{S} \to \mathsf{T}} \text{ (Ax-All)}$$

$$\frac{\Gamma \vdash \mathsf{t} : (\mathsf{U} \to \mathsf{V}) \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t} : (\mathsf{U} \to \mathsf{V}) \to \mathsf{S} \to \mathsf{T}} \text{ (Ax-Stoic)}$$

$$\frac{\Gamma \vdash \mathsf{t}_2 : \mathsf{U} \to \mathsf{V} \qquad \Gamma \vdash \mathsf{t}_1 : (\mathsf{U} \Rightarrow \mathsf{V}) \to \mathsf{S} \Rightarrow \mathsf{T}}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{S} \to \mathsf{T}} \text{ (Ax-Poly)}$$

Figure 5.3: System F-Impure Axioms

THEOREM (Effect-Safety-2). If Γ is pure and capsafe, and $\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}$, then there exists U, V such that $\Gamma \vdash \mathsf{t}_1 : \mathsf{U} \to \mathsf{V}$.

6 Conclusion

We formalized four capability-based effect systems and proved soundness and effect safety for each system. The four systems can serve as the theoretical foundation for implementing capability-based effect systems in functional languages.

The existence of *stoic functions* is the main trait of capability-based effect systems. The interplay between *stoic functions* and *free functions* enables flexible programming patterns that trivially solve the problem of *effect polymorphism*.

Capability-based effect systems have to be paired with strict evaluation, just like monad-based effect systems have to be paired with lazy evaluation.

6.1 Related Work

Lucassen and Gifford first introduced type-and-effect systems[GL86] and effect polymorphism using effect type parameterization [LG88], which is further developed by Talpin and Jouvelot to provide type-and-effect inference [TJ92, TJ94]. Type-and-effect inference can greatly reduce verbosity in syntax, but it only works in languages with global type inference, while Scala is based on local type inference. Even in those languages with global type-and-effect inference, the type signature for effect-polymorphic functions are much more complex than in capability-based effect systems, which is an obstacle to programmers.

Moggi introduced the usage of monads for computation effects[Mog91]. Wadler popularized the usage of monads[Wad92, Wad95] and proved that it's possible to transpose any type-and-effect system into a corresponding monad system[WT03]. As reported in section 1.6 of the thesis[Lip09a], almost every general purpose higher-order function in Haskell needs both a monadic version and non-monadic version. Lippmeier proposed the usage of region variables and dependently kinded witness to encode mutability polymorphism[Lip09b].

Lukas *et al.* studied type-and-effect systems for Scala[ROH12, RAO13, Ryt14]. In lightweight polymorphic effects[ROH12], the dichotomy between *effect-polymorphic function type* and *monomorphic function type* resembles the dichotomy between *stoic function type* and *free function type*. However, in the system effect polymorphism doesn't work for nested functions. To overcome this problem, they unified the two function types in a framework called *relative effect annotation* based on dependent types. In the framework, the function `map` can be marked as effect-polymorphic without introducing a generic effect variable:

```
def map[A, B](f: A => B)(l: List[A]): List[B] @pure(f)
```

The annotation `@pure(f)` says that the effect of `map` depends on the function `f`. Nested functions are handled specially in the system to avoid duplicate annotations.

Heather *et al.* proposed a variant of functions called *spores* for Scala[MHO14]. Compared to normal functions, spores observe a variable capturing discipline. The set of types that can or cannot be captured is part of the type signature of spores, thus can be used by library authors to impose constraints on parameters of function types.

Crary *et al.* proposed a capability calculus for typed memory management[CWM99], which improves the LIFO-style (last-in, first-out) region-based memory management introduced by Tofte and Talpin[TT97] by allowing arbitrary allocation and deallocation order. The safety of deallocation of memory is guaranteed by the type system. In the capability calculus, a capability is a set of regions that are presently valid to access. It tracks uniqueness of capabilities in the type system to control aliasing of capabilities. Functions may be polymorphic over types, regions or capabilities. Capability variables cannot be captured in closures and have to be passed around in the program.

6.2 Future Work

Recursive types, objects, mutation and general recursion are predominant features of industrial languages. To provide a more practical model for industrial languages, it's useful to extend existing systems with these features.

Effect masking can be a useful feature in real-world programming, especially when dealing with exception effects and local mutations. It's worthy to explore effect masking in existing systems.

Another direction of work is to extend the proposed systems with some implicit calculus, in order to avoid explicitly passing capabilities around in the code. This would result in significant savings in boilerplate code, thus make the system more friendly to programmers.

References

[Cha11] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011. 10.1007/s10817-011-9225-2.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, volume 42, pages 315–326. ACM, 2007.

[CWM99] 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.

[GL86] 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.

[LG88] 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.

[Lip09a] Ben Lippmeier. *Type inference and optimisation for an impure world*. Australian National University, 2009.

[Lip09b] Ben Lippmeier. Witnessing purity, constancy and mutability. In *Programming Languages and Systems*, pages 95–110. Springer, 2009.

[MHO14] 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.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[RAO13] Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, page 4. ACM, 2013.

[ROH12] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP 2012–Object-Oriented Programming*, pages 258–282. Springer, 2012.

[Ryt14] Lukas Rytz. *A Practical Effect System for Scala*. EPFL, 2014.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of functional programming*, 2(03):245–271, 1992.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical structures in computer science*, 2(04):461–493, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[WT03] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003.