Type and Effect System for Multi-Staged Exceptions *

Hyunjun Eo[1], Ik-Soon Kim[2], and Kwangkeun Yi[1]

[1] Seoul National University, Korea
[2] École Polytechnique, France

Abstract. We present a type and effect system for a multi-staged language with exceptions. The proposed type and effect system checks if we safely synthesize complex controls with exceptions in multi-staged programming. The proposed exception constructs in multi-staged programming has no artificial restriction. Exception-raise and -handle expressions can appear in expressions of any stage, though they are executed only at stage 0. Exceptions can be raised during code composition and may escape before they are handled. Our effect type system support such features. We prove our type and effect system sound: empty effect means the input program has no uncaught exceptions during its execution.

1 Introduction

Staged computation, which explicitly divides a computation into separate stages, is a unifying framework for existing program generation systems: partial evaluation [5, 1], run-time code generation [7, 10], function inlining and macro expansion [11, 3] are all instances of staged computation. The stage levels are determined by the nesting depth of program generations: stage 0 generates a program of stage 1 that generates a program of stage 2, and so on. The key aspect of multi-staged language is to have code templates (program fragments) as first-class objects. Code templates are freely passed, composed with code of other stages, and executed. At stage 0, computation include all normal computation plus generating code and executing generated code. At stage > 0, computation is just code-composition: it just visits expression's sub-expressions and substitutes code into code when appropriate.

Example 1. As a specializer example in multi-stage programming, consider a recursive `map` function:

```
fun map f nil = nil
  | map f (x::r) = (f x) :: (map f r)
```

* Eo and Yi were supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by Institute for Information Technology Advancement, by National Security Research Institute, and by Microsoft Research Asia. Kim was supported by post-doctoral grants from École Polytechnique and École Normale Supérieure.

The `map` function applies function `f` to each element in the input list, and builds a list with the results returned by `f`. If we know which list is available, we can specialize `map` function with the input list. For example, if input list is `1::2::nil`, we specialize the `map` function to `fn f => (f 1)::(f 2):::nil`. The specialized function is more efficient than the original `map` function because it does not need to traverse the list structure. This specialization can be achieved by the following two functions in Lisp's quasi-quote syntax [11]:

```
fun map_ls nil = `nil
  | map_ls (x::r) = `((f ,x) :: ,(map_ls r))
fun smap ls = eval `(fn f => ,(map_ls ls))
```

At stage 0, the function `smap`, along with `map_ls`, traverses input list `ls` and generates a specialized function of stage 1: the stage increases by the number of surrounding backquotes (`), and decreases by the number of commas (,). Because the application `(f ,x)` in `map_ls` is at stage 1 (surrounded by one backquote), it will not be evaluated. However, the recursive call `(map_ls r)` will be evaluated because it is at stage 0 (surrounded by one backquote and one comma). ■

Exception handling allows the programmer to define, raise and handle exceptional conditions. Exceptional conditions are brought (by a raise expression) to the attention of another expression where the raised exceptions may be handled. Raised exceptions abort the usual program continuation, transfer ("long jump") the control to its handling point, and continue there with the handler expression. Hence by using exceptions programmers can divert any control structure to a point where the corresponding exception is handled. The exception facilities, however, can provide a hole for program safety. Programs can abruptly halt when an exception is raised and never handled.

In this paper we extend the Lisp-like multi-staged language λ^{sim}_{open} [6] with such exceptions and then present a sound type and effect system that statically estimate may-uncaught exceptions in the input programs.

The proposed exception facility in the multi-staged language has no artificial restriction. Lexically, exception-raise and -handle expressions can appear in expressions of any stage. Only restriction, which is natural, is on their dynamics: exceptions must be raised and handled only at stage 0 (at normal computation). Hence, the most interesting feature of our language is exceptions raised during code composition. During computation at stage > 0 (during code composition) an expression can be brought to stage 0 and evaluated there to return a code to substitute for the expression at the code composition. During this stage-0 evaluation an exception can be raised. This raised exception can be caught by a handler only at stage 0. Which handler is that? Any handler at stage 0 in the continuation of the raised exception. A handler that is installed during the stage-0 evaluation can catch it. Or, a handler that is installed at stage 0 before the code composition can catch it and continue.

Example 2. We explain this staged exception semantics by an example. The following function `f` gets a list `ls` and generates a code that multiplies free variable `a` with every element in `ls`.

```
fun g nil = ‘1
  | g x::r = ‘(,x * ,(g r))
fun f ls = ‘(a * ,(g ls))
```

When the input list `ls` is `‘2::‘0::‘3::nil`, the result code will be `‘(a * 2 * 0 * 3 * 1)`. We can prepare a more efficient code by using exceptions. We change `g` to raise `Zero` whenever an element of `ls` is `‘0`.

```
fun g1 nil = ‘1
  | g1 x::r = if x = ‘0 then raise Zero
              else ‘(,x * ,(g1 r))
```

Then, to catch the raised exception `Zero`, we can install a handler during the stage-0 evaluation inside the code composition:

```
fun f1 ls = ‘(a * ,((g1 ls) handle Zero => ‘0))
```

Or, we can install a handler at stage 0 before the code composition:

```
fun f2 ls = ‘(a * ,(g1 ls)) handle Zero => ‘0
```

Note that `f1` and `f2` behave differently. When the input `ls` has `‘0`, `f1` generates `‘(a * 0)` while `f2` generates `‘0`. ■

We extend the effect type system [12–14] for exception analysis of ML [4, 9] to have staged effect types. The extension consists of annotating the box type constructor $\Box$ for the code type with the set of possible exceptions that may be raised during the code execution. Every exception effect has an associated non-negative integer that denotes the number of stages that the raised exception must escape to be handled at stage 0. For example, $\texttt{Zero}^n$ in an effect means that uncaught exception `Zero` can be handled at stage 0 after escaping n stages.

The type of a code with `raise` c (c for an exception name) would be:

$$\texttt{‘(raise } c\texttt{)} : \Box(\varnothing \rhd A, \{c^0\}), \emptyset.$$

The box type $\Box(\varnothing \rhd A, \{c^0\})$ means that the above expression is a closed code of type A, and may raise an exception c when evaluated. The empty effect $\emptyset$ means that this code does not raise any exception. The type of executing the above code template by `eval` would be:

$$\texttt{eval ‘(raise } c\texttt{)} : A, \{c^0\}.$$

The effect $\{c^0\}$ means that the above expression may raise exception c. The superscript 0 means that the raised exception c can be handled by a proper handler at the current stage.

Example 3. We will explain such exceptions and their corresponding types by an example program in Fig. 1. The function `codegen` compiles a program in language L into an ML program of `int` type. During compilation, it may raise an exception `CompileError`. The type of codegen would be:

$$\texttt{codegen} : L \xrightarrow{\{\texttt{CompileError}^0\}} \Box(\varnothing \rhd \texttt{int}), \emptyset.$$

```
exception CompileError

type L = CONST of int | PLUS of L * L | ···

fun codegen e =
  case e of
    CONST x => ‘,x
  | PLUS(e1,e2) => ‘(,(codegen e1) + ,(codegen e2))
    ···
  | _ => raise CompileError

fun compile program =
    (codegen (parse program))
    handle CompileError => print "Compile Error"; ‘0
```

Fig. 1. An example: compiling L into ML

It means that exception `CompileError` may be raised when we apply `codegen`. Hence, the type of application `(codegen e1)` is:

$$\texttt{(codegen e1)} : \Box(\varnothing \rhd \texttt{int}, \emptyset), \{\texttt{CompileError}^0\}.$$

In order to plug the above code inside another code (i.e., inside a backquote expression), we have to comma it: `‘(,(codegen e1))`. The type of `,(codegen e1)` is:

$$\texttt{,(codegen e1)} : \texttt{int}, \{\texttt{CompileError}^1\}.$$

Because this expression is inside a code template (stage 1), the raised exception `CompileError` cannot be handled at the current stage: it can be handled only after escaping 1 stage. The superscript 1 in $\texttt{CompileError}^1$ describes this situation. The type of the enclosing code template of the above expression is:

$$\texttt{‘(,(codegen e1))} : \Box(\varnothing \rhd \texttt{int}, \emptyset), \{\texttt{CompileError}^0\}.$$

It means that the above expression may raise exception `CompileError`. The superscript 0 in $\texttt{CompileError}^0$ means that `CompileError` can be handled at the current stage. Hence a handler at stage 0 can catch it

$$\texttt{‘(,(codegen e1))}\ \texttt{handle CompileError => ‘0} : \Box(\varnothing \rhd \texttt{int}, \emptyset), \emptyset$$

while a handler inside code template (at stage > 0) cannot handle it

$$\texttt{‘(,(codegen e1)}\ \texttt{handle CompileError => 0)} : \Box(\varnothing \rhd \texttt{int}, \emptyset), \{\texttt{CompileError}^0\}$$

■

Recently, Nanevski has proposed an exception type for staged language in a different formulation [8]: his language requires programmer to explicitly name

each code composition, while our (in an implicit style) allows unnamed code composition[1]. Though explicitly naming code composition may make type system simple (such that it is not necessary to annotate effects with stage levels), we chose to use the implicit style. Our reason is pragmatic: to have an exception type system to support Lisp's quasi-quote system. Lisp's quasi-quote system is an implicit multi-staged language that has evolved to comply with the demands from multi-staged programming practices. Moreover, our type system enjoys the advantage of [6] that supports open code as first-class objects.

In the rest of our paper, we introduce the syntax and semantics of our language (Section 2), define the exception types and effects (Section 3.1), describe typing rules (Section 3.2), and prove the soundness of our effect type system (Section 3.3).

2 Language

2.1 Syntax

Our language λ_{exn}^{stage} has the staging constructs á la λ_{open}^{sim} [6] with the exception-raise and -handle constructs. We exclude references and gensyms from λ_{open}^{sim} in order to focus on exceptions.

$$\begin{aligned} e \in Exp ::= & \; i \mid c \mid x \mid \lambda x.e \mid e_1 e_2 \\ & \mid \texttt{box}\, e \mid \texttt{eval}\, e \mid \texttt{unbox}_k\, e \\ & \mid \texttt{raise}\, e \mid \texttt{handle}\, e_1\, c\, e_2 \end{aligned}$$

Expression i is an integer constant, c is an exception name. Expression $\texttt{box}\, e$, $\texttt{unbox}_k\, e$ $(k > 0)$, and $\texttt{eval}\, e$ are for manipulating code templates that respectively correspond to the backquote(‘), the comma(,) ..... k stages, and the `eval` in Lisp's quasi-quote notation. At stage 0, $\texttt{raise}\, e$ raises an exception returned from evaluating e. Handle expression $\texttt{handle}\, e_1\, c\, e_2$ evaluates e_1 first. If it does not raise an exception, its result is the handle expression's result. If it raises exception c, then the handler catches it and evaluates e_2. If it raises an exception other than c, then the raised exception is the result.

2.2 Operational Semantics

Fig. 2 shows a big-step operational semantics of our language λ_{exn}^{stage}. Evaluation

$$e \xrightarrow{n} r$$

denotes that expression e is evaluated to result r at stage n.

Values V^n are the values of stage n. In multi-staged languages, values exists at every stage. Values at stage 0 are normal ones plus code. Values at stages> 0 are code only. A staged value v^n $(n > 0)$ is an expression that is to be evaluated

[1] Davies and Pfenning have shown that both explicit and implicit formulations are inter-translatable [2].

	Normal computations (at stage 0) and code compositions (at stage $n > 0$)	Propagation of raised exceptions
(EINT)	$i \xrightarrow{n} i \ (n \geq 0)$	
(EEXN)	$c \xrightarrow{n} c \ (n \geq 0)$	
(EVAR)	$x \xrightarrow{n} x \ (n > 0)$	
(EABS)	$\lambda x.e \xrightarrow{0} \lambda x.e$	
	$\dfrac{e \xrightarrow{n} v}{\lambda x.e \xrightarrow{n} \lambda x.v}(n > 0)$	$\dfrac{e \xrightarrow{n} \overline{c}}{\lambda x.e \xrightarrow{n} \overline{c}}(n > 0)$
(EAPP)	$\dfrac{e_1 \xrightarrow{0} \lambda x.e \quad e_2 \xrightarrow{0} v_2 \quad [x \overset{0}{\mapsto} v_2]e \xrightarrow{0} v}{e_1\ e_2 \xrightarrow{0} v}$	$\dfrac{e_1 \xrightarrow{n} \overline{c}}{e_1\ e_2 \xrightarrow{n} \overline{c}}(n \geq 0)$
	$\dfrac{e_1 \xrightarrow{n} v_1 \quad e_2 \xrightarrow{n} v_2}{e_1\ e_2 \xrightarrow{n} v_1\ v_2}(n > 0)$	$\dfrac{e_2 \xrightarrow{n} \overline{c}}{e_1\ e_2 \xrightarrow{n} \overline{c}}(n \geq 0)$
(EBOX)	$\dfrac{e \xrightarrow{n+1} v}{\texttt{box}\ e \xrightarrow{n} \texttt{box}\ v}(n \geq 0)$	$\dfrac{e \xrightarrow{n+1} \overline{c}}{\texttt{box}\ e \xrightarrow{n} \overline{c}}(n \geq 0)$
(EUNBOX)	$\dfrac{e \xrightarrow{0} \texttt{box}\ v}{\texttt{unbox}_n\ e \xrightarrow{n} v}(n > 0)$	
	$\dfrac{e \xrightarrow{n-k} v}{\texttt{unbox}_k\ e \xrightarrow{n} \texttt{unbox}_k\ v}(n > k > 0)$	$\dfrac{e \xrightarrow{n-k} \overline{c}}{\texttt{unbox}_k\ e \xrightarrow{n} \overline{c}}(n \geq k > 0)$
(EEVAL)	$\dfrac{e \xrightarrow{0} \texttt{box}\ v^1 \quad v^1 \xrightarrow{0} v^0}{\texttt{eval}\ e \xrightarrow{0} v^0}$	
	$\dfrac{e \xrightarrow{n} v}{\texttt{eval}\ e \xrightarrow{n} \texttt{eval}\ v}(n > 0)$	$\dfrac{e \xrightarrow{n} \overline{c}}{\texttt{eval}\ e \xrightarrow{n} \overline{c}}(n \geq 0)$
(ERAISE)	$\dfrac{e \xrightarrow{0} c}{\texttt{raise}\ e \xrightarrow{0} \overline{c}}$	
	$\dfrac{e \xrightarrow{n} v}{\texttt{raise}\ e \xrightarrow{n} \texttt{raise}\ v}(n > 0)$	$\dfrac{e \xrightarrow{n} \overline{c}}{\texttt{raise}\ e \xrightarrow{n} \overline{c}}(n \geq 0)$
(EHANDLE)	$\dfrac{e_1 \xrightarrow{0} v}{\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{0} v}$	$\dfrac{e_1 \xrightarrow{n} \overline{c}}{\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{n} \overline{c}}(n > 0)$
	$\dfrac{e_1 \xrightarrow{0} \overline{c} \quad e_2 \xrightarrow{0} v}{\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{0} v}$	$\dfrac{e_1 \xrightarrow{n} \overline{c}}{\texttt{handle}\ e_1\ c'\ e_2 \xrightarrow{n} \overline{c}}(n > 0)$
	$\dfrac{e_1 \xrightarrow{0} \overline{c}}{\texttt{handle}\ e_1\ c'\ e_2 \xrightarrow{0} \overline{c}}$	$\dfrac{e_2 \xrightarrow{n} \overline{c}}{\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{n} \overline{c}}(n > 0)$
	$\dfrac{e_1 \xrightarrow{n} v_1 \quad e_2 \xrightarrow{n} v_2}{\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{n} \texttt{handle}\ v_1\ c\ v_2}(n > 0)$	$\dfrac{e_2 \xrightarrow{n} \overline{c}}{\texttt{handle}\ e_1\ c'\ e_2 \xrightarrow{n} \overline{c}}(n > 0)$

Fig. 2. Operational semantics of λ_{exn}^{stage}.

$$
\begin{array}{lll}
[x \overset{n}{\mapsto} v]i & = i & \\
[x \overset{n}{\mapsto} v]c & = c & \\
[x \overset{n}{\mapsto} v]y & = v, & \text{if } x = y \text{ and } n = 0 \\
 & = y, & \text{otherwise} \\
[x \overset{n}{\mapsto} v](\lambda y.e) & = \lambda y.e, & \text{if } x = y \text{ and } n = 0 \\
 & = \lambda y.([x \overset{n}{\mapsto} v]e), & \text{otherwise} \\
[x \overset{n}{\mapsto} v](e_1\ e_2) & = ([x \overset{n}{\mapsto} v]e_1)\ ([x \overset{n}{\mapsto} v]e_2) & \\
[x \overset{n}{\mapsto} v](\texttt{box}\ e) & = \texttt{box}\ ([x \overset{n+1}{\mapsto} v]e) & \\
[x \overset{n}{\mapsto} v](\texttt{unbox}_k\ e) & = \texttt{unbox}_k\ ([x \overset{n-k}{\mapsto} v]e) & \\
[x \overset{n}{\mapsto} v](\texttt{eval}\ e) & = \texttt{eval}\ ([x \overset{n}{\mapsto} v]e) & \\
[x \overset{n}{\mapsto} v](\texttt{raise}\ e) & = \texttt{raise}\ ([x \overset{n}{\mapsto} v]e) & \\
[x \overset{n}{\mapsto} v](\texttt{handle}\ e_1\ c\ e_2) & = \texttt{handle}\ ([x \overset{n}{\mapsto} v]e_1)\ c\ ([x \overset{n}{\mapsto} v]e_2) &
\end{array}
$$

Fig. 3. Substituting v for free variable x of stage 0 at stage n.

later when it is demoted to stage 0 by the `eval` construct. Results R^n at stage n are either values at stage n or raised exceptions. We write $\overline{c}$ for a raised c exception.

$$
\begin{array}{rcll}
v^n \in V^n & ::= & i \mid c \mid \lambda x.e \mid \texttt{box}\ v^1 & \text{if } n = 0 \\
 & ::= & i \mid c \mid x \mid \lambda x.v^n \mid v^n v^n & \\
 & & \mid \texttt{box}\ v^{n+1} \mid \texttt{eval}\ v^n \mid \texttt{unbox}_k\ v^{n-k} & \\
 & & \mid \texttt{raise}\ v^n \mid \texttt{handle}\ v_1^n\ c\ v_2^n & \text{if } n > k \geq 0
\end{array}
$$

$$
r^n \in R^n ::= v^n \mid \overline{c}
$$

Staging semantics of λ^{stage}_{exn} is the same as in λ^{sim}_{open} [6], conservatively extended with exceptions.

At stage 0, computation include, in addition to normal computation, generating code and executing generated code. `(EINT)`, `(EEXN)`, `(EABS)`, and `(EAPP)` are as usual. `(EAPP)` defines the beta reduction. The definition of the staged substitution operator $[x \overset{n}{\mapsto} v]$ is in Fig. 3. `(EBOX)` defines code generation. `(EEVAL)` at stage 0 executes generated code: a code template $\texttt{box}\ v^1$ becomes an expression v^1 then is evaluated. By the type system, v^1 is restricted to closed code. (See section 3). Because only closed code can be evaluated at stage 0, we don't have an evaluation rule for variable at stage 0.

At stage > 0, only meaningful computation is code substitution. It consists of just visiting every sub-expressions and substitute code into code when appropriate. Code substitution is by the $\texttt{unbox}_k$ expression. At stage n, expression $\texttt{unbox}_n\ e$ executes the sub-expression e at stage 0 then substitute its result code for the $\texttt{unbox}_n$ expression: `(EUNBOX)`.

`(ERAISE)` raises an exception only at stage 0. The right side of Fig. 2 shows that the propagation of raised exception $\overline{c}$. A raised exception $\overline{c}$ is propagated to the nearest handler that handles exception c at stage 0: `(EHANDLE)`. Raised exceptions can escape any control structure including stages.

Example 4. In the following expression, an exception c is initially raised at stage 0, is "promoted" to stage 2 (by $\texttt{unbox}_2$), and then escapes to stage 0 (by two `box`es).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{c \xrightarrow{0} c}{\texttt{raise}\, c \xrightarrow{0} \overline{c}}}{\texttt{unbox}_2\, \texttt{raise}\, c \xrightarrow{2} \overline{c}}}{\texttt{box}\, (\texttt{unbox}_2\, \texttt{raise}\, c) \xrightarrow{1} \overline{c}}}{\texttt{box}\, (\texttt{box}\, (\texttt{unbox}_2\, \texttt{raise}\, c)) \xrightarrow{0} \overline{c}}$$

$\overline{c}$ is raised at stage 0

$\overline{c}$ is promoted to stage 2

$\overline{c}$ is demoted to stage 1

$\overline{c}$ is demoted to stage 0

■

Like exceptions raised during normal computation, stage-escaping exceptions (raised during code composition) are handled only at stage 0. Hence, `handle` expressions at stage>0 cannot handle a raised exceptions.

Example 5. The following expression evaluates to $(\texttt{box}\, 0)$ because raised exception $\overline{c}$ is propagated to stage 0, and handled there to evaluate into code `0`.

$$\dfrac{\dfrac{\dfrac{\dfrac{c \xrightarrow{0} c}{\texttt{raise}\, c \xrightarrow{0} \overline{c}}}{\texttt{unbox}_1\, \texttt{raise}\, c \xrightarrow{1} \overline{c}}}{\texttt{box}\, (\texttt{unbox}_1\, \texttt{raise}\, c) \xrightarrow{0} \overline{c}} \quad \dfrac{0 \xrightarrow{1} 0}{\texttt{box}\, 0 \xrightarrow{0} \texttt{box}\, 0}}{\texttt{handle}\, (\texttt{box}\, (\texttt{unbox}_1\, \texttt{raise}\, c))\, c\, (\texttt{box}\, 0) \xrightarrow{0} \texttt{box}\, 0}$$

`handle` at stage 0 catches $\overline{c}$

■

Example 6. The following expression raises uncaught exception $\overline{c}$ because `handle` expression inside the code template cannot handle $\overline{c}$ and just propagates it to stage 0 escaping the code template of stage 1.

$$\dfrac{\dfrac{\dfrac{\dfrac{c \xrightarrow{0} c}{\texttt{raise}\, c \xrightarrow{0} \overline{c}}}{\texttt{unbox}_1\, (\texttt{raise}\, c) \xrightarrow{1} \overline{c}}}{\texttt{handle}\, (\texttt{unbox}_1\, (\texttt{raise}\, c))\, c\, 0 \xrightarrow{1} \overline{c}}}{\texttt{box}\, (\texttt{handle}\, (\texttt{unbox}_1\, (\texttt{raise}\, c))\, c\, 0) \xrightarrow{0} \overline{c}}$$

`handle` at stage 1 cannot catch c

■

As in λ^{sim}_{open} [6], at stages>0 (at code composition stages) no alpha-equivalence is supported, i.e., variable-capturing substitution is allowed. If we change a bound name in expressions of stages>0, the resulting program's semantics changes. On the other hand at stage 0 (at the normal computation stage) alpha-equivalence is preserved as usual. (We enforce only closed code to be evaluated at stage 0. See `(TEVAL)` in Section 3).

3 Effect Type System

3.1 Exception Types and Effects

We use A, B for types, φ for effects, and ψ for a set of exceptions.

$$A, B \in \mathit{Type} ::= \texttt{int} \mid \texttt{exn}(\psi) \mid A \xrightarrow{\varphi} B \mid \Box(\Gamma \rhd A, \varphi)$$

Exception type $\texttt{exn}(\psi)$ has a set of exceptions that an expression of that type can have. As in usual effect systems, function type $A \xrightarrow{\varphi} B$ has a latent effect φ that describes exceptions that may be raised during the evaluation of the function's body. Code type $\Box(\Gamma \rhd A, \varphi)$ is a conditional modal type in which condition Γ specifies the types of free variables in the code template of type A. Our code type is also annotated by a latent effect φ that describes exceptions that may be raised when the code template of that type is evaluated by `eval` .

$$\begin{array}{ll} \varphi \in \mathit{Effects} & = 2^{\mathit{Exn} \times \mathbb{N}} \\ \psi \in \mathit{Exceptions} & = 2^{\mathit{Exn}} \\ c \in \mathit{Exn} & = \text{set of exception names} \end{array}$$

Effects in our types are sets of exceptions, where each exception has the number of stages to escape. The stage-escaping numbers denote how many stages should those exceptions escape to be handled at stage 0. For $\psi \in \mathit{Exceptions}$, ψ^n means $\{c^n \mid c \in \psi\} \in \mathit{Effects}$

Normal Exceptions vs. Stage-Escaping Exceptions

Normal exceptions, which may be raised during normal computation, and stage-escaping exceptions, which may be raised during code composition, have a different behavior. If they are in a code template, stage-escaping exceptions can escape stages, while normal exceptions cannot. For example, $\texttt{raise}\, c$ raises a normal exception, while $\texttt{unbox}_1\ (\texttt{raise}\, c)$ raises a stage-escaping exception. Hence $\texttt{box}\ (\texttt{raise}\, c)$ does not raise exception c, while $\texttt{box}\ (\texttt{unbox}_1\ (\texttt{raise}\, c))$ raises exception c.

Definition 1. *For an effect φ, and a unary predicate $P : \mathbb{N} \to \{true, false\}$, we define P-restricted effect φ, denoted φ^P, as follows:*

$$\varphi^P \stackrel{\text{def}}{=} \{c^n \mid c^n \in \varphi \wedge P(n)\}$$

We can decompose an effect φ into a normal effect $\varphi^{=0}$ and a stage-escaping effect $\varphi^{>0}$, where "$= 0$" is a unary predicate "is equal to 0" and "> 0" is a unary predicate "is greater than 0". Hence the normal effect $\varphi^{=0}$ means exceptions which escape 0 stages (cannot escape stages), and the stage-escaping effect $\varphi^{>0}$ means exceptions which escape at least one stage.

Promotion and Demotion of Effects

As shown in Example 4, stage-escaping exceptions can cross stages upwards (by $\texttt{unbox}_k$) or downwards (by `box`). When stage-escaping exceptions are promoted or demoted to other stages, the effects that estimate those exceptions should also be promoted or demoted, respectively.

Definition 2. *A promotion* $\uparrow_k$ *is a function from Effects to Effects such that*

$$\uparrow_k \varphi \stackrel{def}{=} \{c^{n+k} \mid c^n \in \varphi\}, \text{ where } n \geq 0 \text{ and } k > 0.$$

A demotion $\downarrow$ *is a function from Effects to Effects such that*

$$\downarrow \varphi \stackrel{def}{=} \{c^{n-1} \mid c^n \in \varphi\}, \text{ where } n > 0.$$

3.2 Typing Rules

The typing judgment

$$\Gamma_0 \cdots \Gamma_n \vdash e : A, \varphi$$

means that an expression e, under type environment $\Gamma_0 \cdots \Gamma_n$ has type A and effect φ at stage n. $\Gamma_0 \cdots \Gamma_n$ is a sequence of type environments $\Gamma_0, \cdots, \Gamma_n$. Γ_n is the current type environment. Subscripts $0, \cdots, n$ are stage numbers. Fig. 4 shows our typing rules for λ^{stage}_{exn}.

For exception name c, we include it inside its exception type `exn`. For instance, the type of c must be of the form $\texttt{exn}(\psi)$ such that $c \in \psi$:

$$\frac{c \in \psi}{\Gamma_0 \cdots \Gamma_n \vdash c : \texttt{exn}(\psi), \emptyset} \quad \texttt{(TEXN)}$$

The type of raise expression $\texttt{raise}\, e$ can be any arbitrary type A. Because exceptions ψ are raised at the current stage, `(TRAISE)` collects ψ^0 and the effect φ of its sub-expression e.

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash e : \texttt{exn}(\psi), \varphi}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{raise}\, e : A, \psi^0 \cup \varphi} \quad \texttt{(TRAISE)}$$

Handle expression $\texttt{handle}\, e_1\ c\ e_2$ catches exception c of e_1 only when the handle expression is evaluated at stage 0, its effect catches only c^0.

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A, \varphi \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi' \quad \varphi'' = (\varphi \setminus \{c^0\}) \cup \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{handle}\, e_1\ c\ e_2 : A, \varphi''} \quad \texttt{(THANDLE)}$$

For box expression $\texttt{box}\, e$, `(TBOX)` injects normal exceptions $\varphi^{=0}$ of the sub-expression e into the latent effect of the box type, because they can not escape stages: the box expression would not raise them until unboxed or evaluated.

$$\texttt{(TINT)}\quad \Gamma_0 \cdots \Gamma_n \vdash i : \texttt{int}, \emptyset$$

$$\texttt{(TEXN)}\quad \frac{c \in \psi}{\Gamma_0 \cdots \Gamma_n \vdash c : \texttt{exn}(\psi), \emptyset}$$

$$\texttt{(TVAR)}\quad \frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash x : A, \emptyset}$$

$$\texttt{(TABS)}\quad \frac{\Gamma_0 \cdots \Gamma_n + x : A \vdash e : B, \varphi}{\Gamma_0 \cdots \Gamma_n \vdash \lambda x.e : A \xrightarrow{\varphi^{=0}} B, \varphi^{>0}}$$

$$\texttt{(TAPP)}\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \xrightarrow{\varphi''} B, \varphi \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash e_1\ e_2 : B, \varphi \cup \varphi' \cup \varphi''}$$

$$\texttt{(TBOX)}\quad \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A, \varphi}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\, e : \Box(\Gamma \rhd A, \varphi^{=0}), \downarrow \varphi^{>0}}$$

$$\texttt{(TUNBOX)}\quad \frac{\Gamma_0 \cdots \Gamma_{n-k} \vdash e : \Box(\Gamma_n \rhd A, \varphi), \varphi' \quad n \geq k > 0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{unbox}_k\, e : A, \varphi \cup (\uparrow_k \varphi')}$$

$$\texttt{(TEVAL)}\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\varnothing \rhd A, \varphi), \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{eval}\, e : A, \varphi \cup \varphi'}$$

$$\texttt{(TRAISE)}\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \texttt{exn}(\psi), \varphi}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{raise}\, e : A, \psi^0 \cup \varphi}$$

$$\texttt{(THANDLE)}\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A, \varphi \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi' \quad \varphi'' = (\varphi \setminus \{c^0\}) \cup \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{handle}\, e_1\ c\ e_2 : A, \varphi''}$$

$$\texttt{(TSUB)}\quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : A, \varphi \quad \varphi \subseteq \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash e : A, \varphi'}$$

Fig. 4. Typing rules of λ_{exn}^{stage}

Stage-escaping exceptions $\varphi^{>0}$ of e can escape to the outside of the box expression, hence the effect of the box expression must include them. Because the evaluation $(\texttt{box}\, e) \xrightarrow{n} \overline{c}$ and its premise $e \xrightarrow{n+1} \overline{c}$ imply that the raised exception $\overline{c}$ escapes one stage (from $n+1$ to n), stage-escaping exceptions $\varphi^{>0}$ of e should be demoted to $\downarrow \varphi^{>0}$.

$$\frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A, \varphi}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\, e : \Box(\Gamma \rhd A, \varphi^{=0}), \downarrow \varphi^{>0}} \qquad \texttt{(TBOX)}$$

For unbox expression $\texttt{unbox}_k\ e$ $(k > 0)$, the only normal exceptions the unbox expression may have are exceptions in the latent effect φ of e. Note that the evaluation $(\texttt{unbox}_k\, e) \xrightarrow{n} \overline{c}$ and its premise $e \xrightarrow{n-k} \overline{c}$ imply that the stage of the raised exception $\overline{c}$ would be increased by k: from $n-k$ to n. Hence, to be handled, the uncaught exceptions of the unbox expression should escape k more

stages than those of its sub-expression. Hence we promote the effect φ' of e to $\uparrow_k \varphi'$.

$$\frac{\Gamma_0 \cdots \Gamma_{n-k} \vdash e : \Box(\Gamma_n \rhd A, \varphi), \varphi' \quad n \geq k > 0}{\Gamma_0 \cdots \Gamma_n \vdash \mathtt{unbox}_k\ e : A, \varphi \cup (\uparrow_k \varphi')} \quad \texttt{(TUNBOX)}$$

For eval expression `eval`e, `(TEVAL)` allows only closed code to be evaluated by `eval` construct. When we evaluate a code with free variables, those free variables may cause unintended variable capture, because of the alpha-conversion at stage 0. Recall that we assume that variables in a code template can not be alpha-converted (for the sake of unhygienic macros), but variables at stage 0 can be alpha-converted. Hence we force to evaluate only colosed code: the code template type $\Box(\varnothing \rhd A, \varphi)$ of e should have empty environment. Like unbox expression, the effect of eval expression should have both of the latent effect φ and the effect φ' of e. We don't need to promote the effect φ' because the stage of e and that of `eval` e are the same.

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\varnothing \rhd A, \varphi), \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash \mathtt{eval}\ e : A, \varphi \cup \varphi'} \quad \texttt{(TEVAL)}$$

Abstraction $\lambda x.e$ is a value at stage 0, while it can be an evaluable expression at stage $n > 0$. Hence the normal exception $\varphi^{=0}$ of e should be injected to the latent effect of the function, and might be raised where the function is applied. Stage-escaping exception $\varphi^{>0}$ of e should be propagated to $\lambda x.e$. Note that $c^0 \in \varphi^{=0}$ means that e may raise exception c at stage 0 ($e \xrightarrow{0} \overline{c}$), and $c^n \in \varphi^{>0}$ means that e may be evaluated to the raised exception $\overline{c}$ at stage $n > 0$ ($e \xrightarrow{n} \overline{c}$).

$$\frac{\Gamma_0 \cdots \Gamma_n + x : A \vdash e : B, \varphi}{\Gamma_0 \cdots \Gamma_n \vdash \lambda x.e : A \xrightarrow{\varphi^{=0}} B, \varphi^{>0}} \quad \texttt{(TABS)}$$

For application $e_1\ e_2$, `(TAPP)` is conventional. All effects from evaluating e_1, e_2, and the function's body are collected. The function body's effect is the latent effect in the type of e_1.

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \xrightarrow{\varphi''} B, \varphi \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash e_1\ e_2 : B, \varphi \cup \varphi' \cup \varphi''} \quad \texttt{(TAPP)}$$

The subsumption rule `(TSUB)` allows any expression to be treated as having more effect than it actually does. By applying subsumption rule to the latent effect, abstractions or the code templates can be treated as having more effect than reality. Without the subsumption rule, a value of type $\Box(\Gamma \rhd A, \emptyset)$ and a value of type $\Box(\Gamma \rhd A, \{c^0\})$ could not both be passed as arguments to the same function, because the function and argument types would have to match exactly.

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash e : A, \varphi \quad \varphi \subseteq \varphi'}{\Gamma_0 \cdots \Gamma_n \vdash e : A, \varphi'} \qquad \texttt{(TSUB)}$$

Example 7. A code template $\texttt{box}\,(\texttt{raise}\,c)$, which raises an exception c when evaluated, has the following typing:

$$\dfrac{\dfrac{\varnothing\varnothing \vdash c : \texttt{exn}(\{c\}), \emptyset}{\varnothing\varnothing \vdash \texttt{raise}\,c : A, \{c^0\}}}{\varnothing \vdash \texttt{box}\,(\texttt{raise}\,c) : \square(\varnothing \rhd A, \{c^0\}), \emptyset.}$$

The empty effect $\emptyset$ implies that the code template will not raise any exception, and the type $\square(\varnothing \rhd A, \{c^0\})$ implies that the exception c may be raised when we execute the code template. ■

Example 8. Exceptions may be raised during code composition. Recall the expression in Example 4.

$$\dfrac{\dfrac{\dfrac{\dfrac{c \xrightarrow{0} c}{\texttt{raise}\,c \xrightarrow{0} \overline{c}}}{\texttt{unbox}_2\,\texttt{raise}\,c \xrightarrow{2} \overline{c}}}{\texttt{box}\,(\texttt{unbox}_2\,\texttt{raise}\,c) \xrightarrow{1} \overline{c}}}{\texttt{box}\,(\texttt{box}\,(\texttt{unbox}_2\,\texttt{raise}\,c)) \xrightarrow{0} \overline{c}}$$

$\overline{c}$ is raised at stage 0

$\overline{c}$ is promoted to stage 2

$\overline{c}$ is demoted to stage 1

$\overline{c}$ is demoted to stage 0

The above expression has the following typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \vdash c : \texttt{exn}(\{c\}), \emptyset}{\varnothing \vdash \texttt{raise}\,c : \square(\varnothing \rhd A, \emptyset), \{c^0\}}}{\varnothing\varnothing\varnothing \vdash \texttt{unbox}_2\,\texttt{raise}\,c : A, \{c^2\}}}{\varnothing\varnothing \vdash \texttt{box}\,(\texttt{unbox}_2\,\texttt{raise}\,c) : \square(\varnothing \rhd A, \emptyset), \{c^1\}}}{\varnothing \vdash \texttt{box}\,(\texttt{box}\,(\texttt{unbox}_2\,\texttt{raise}\,c)) : \square(\varnothing \rhd \square(\varnothing \rhd A, \emptyset), \emptyset), \{c^0\}.}$$

This typing means that the expression is a code template of a code template, and may raise uncaught exception c. The stage-escaping numbers n of c^n in the proof tree exactly capture the dynamic stages of $\overline{c}$ ($0 \to 2 \to 1 \to 0$). ■

Example 9. An exception raised during code composition can be handled by a proper handler installed at stage 0. Recall the expression in Example 5. A raised exception c at stage 0 can be caught by a handler at stage 0:

$$\texttt{handle}\,(\texttt{box}\,(\texttt{unbox}_1\,\texttt{raise}\,c))\,c\,(\texttt{box}\,0) \xrightarrow{0} \texttt{box}\,0$$

Our effect type system decides that the above expression has no uncaught exception:

$$\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \vdash c : \mathtt{exn}(\{c\}), \emptyset}{\varnothing \vdash \mathtt{raise}\, c : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \{c^0\}}}{\varnothing\varnothing \vdash \mathtt{unbox}_1\, \mathtt{raise}\, c : \mathtt{int}, \{c^1\}}}{\varnothing \vdash \mathtt{box}\, (\mathtt{unbox}_1\, \mathtt{raise}\, c) : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \{c^0\}} \quad \dfrac{\varnothing\varnothing \vdash 0 : \mathtt{int}, \emptyset}{\varnothing \vdash \mathtt{box}\, 0 : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \emptyset}}{\varnothing \vdash \mathtt{handle}\, (\mathtt{box}\, (\mathtt{unbox}_1\, \mathtt{raise}\, c))\, c\, (\mathtt{box}\, 0) : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \emptyset}$$

■

Example 10. A raised exception c at stage 0 can not be caught by a handler in a code template (at stage 1). Recall the expression in Example 6. It raises exception c in the code template, and does not handle the exception because the handler is inside the code template:

$$\mathtt{box}\, (\mathtt{handle}\, (\mathtt{unbox}_1\, \mathtt{raise}\, c)\, c\, 0) \xrightarrow{0} \overline{c}$$

Our effect type system safely estimates that the above expression may raise an uncaught exception c:

$$\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \vdash c : \mathtt{exn}(\{c\}), \emptyset}{\varnothing \vdash \mathtt{raise}\, c : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \{c^0\}}}{\varnothing\varnothing \vdash \mathtt{unbox}_1\, \mathtt{raise}\, c : \mathtt{int}, \{c^1\}} \quad \varnothing\varnothing \vdash 0 : \mathtt{int}, \emptyset}{\varnothing\varnothing \vdash \mathtt{handle}\, (\mathtt{unbox}_1\, \mathtt{raise}\, c)\, c\, 0 : \mathtt{int}, \{c^1\}}}{\varnothing \vdash \mathtt{box}\, (\mathtt{handle}\, (\mathtt{unbox}_1\, \mathtt{raise}\, c)\, c\, 0) : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \{c^0\}}$$

■

Example 11. Let's consider the following code template.

$$\mathtt{box}\, (\mathtt{handle}\, (\mathtt{raise}\, c)\, c\, 0)$$

The handler inside the code template catches exception c when the code template is executed. The above code template has the following typing:

$$\dfrac{\dfrac{\dfrac{\varnothing\varnothing \vdash c : \mathtt{exn}(\{c\}), \emptyset}{\varnothing\varnothing \vdash \mathtt{raise}\, c : \mathtt{int}, \{c^0\}} \quad \varnothing\varnothing \vdash 0 : \mathtt{int}, \emptyset}{\varnothing\varnothing \vdash \mathtt{handle}\, (\mathtt{raise}\, c)\, c\, 0 : \mathtt{int}, \emptyset}}{\varnothing \vdash \mathtt{box}\, (\mathtt{handle}\, (\mathtt{raise}\, c)\, c\, 0) : \Box(\varnothing \rhd \mathtt{int}, \emptyset), \emptyset}$$

Our system gives a correct effect typing for every stage. ■

3.3 Soundness

In our evaluation rule (in Fig. 2), there are two rules which convert values at stage n to values at another stage m. The `eval` at stage 0 converts $\mathtt{box}\, v^1$ into

v^1 and evaluate v^1 at stage 0; it demotes values at stage $n > 0$ to expressions at stage $(n-1)$. The $\texttt{unbox}_k$ at stage $k > 0$ converts $\texttt{box}\, v^1$ into v^k; it promotes values at stage $(n+1)$ to values at stage $(n+k)$. The following lemma shows that such demotion and promotion preserve types and effects. We can freely promote or demote values, because our types and effects only depend on the structure of their sub-expressions and do not depend on the stages where they are. The only restriction of demotion is that Γ_1 should be $\varnothing$, because a value at stage 1 must not have a free variable of stage 1 to be demoted (or to be evaluated by `eval`).

Lemma 1 (Demotion and Promotion). *Suppose* $\varnothing\Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi$.

1. *If* $\Gamma_1 = \varnothing$ *then* $\Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi$.
2. $\varnothing\Gamma'_1 \cdots \Gamma'_m\Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi$ *for all* $\Gamma'_1 \cdots \Gamma'_m$.

Proof. We prove the lemma by induction on the structure of v. ■

Values at stage $n > 0$ may raise exceptions when demoted to stage 0 (or evaluated by `eval`). Hence we can not claim that values at any stage have empty effect. However, any value v^0 at stage 0 has an empty effect (does not raise any exception):

Lemma 2 (Empty Effect of v^0). *If* $\Gamma_0 \vdash v : A, \varphi$ *then* $\Gamma_0 \vdash v : A, \emptyset$.

Proof. We first prove that $\Gamma_0 \cdots \Gamma_n \vdash v : A, \varphi^{<n}$, if $\Gamma_0 \cdots \Gamma_n \vdash v : A, \varphi$. It can be shown by induction on the structure of v. Then the lemma immediately follows from $\forall\varphi : (\varphi^{<0}) = \emptyset$. ■

The soundness theorem shows that every exception that may be raised and uncaught during the evaluation of an expression should be collected inside the expression's effect. For the proof of the soundness theorem, we need Lemma 1 and Lemma 2.

Theorem 1 (Soundness). *Suppose* $\varnothing\Gamma_1 \cdots \Gamma_n \vdash e : A, \varphi$.

1. *If* $e \xrightarrow{n} v$ *then* $\varnothing\Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi$.
2. *If* $e \xrightarrow{n} \overline{c}$ *then* $\varphi \supseteq \{c^n\}$.

Proof. We prove the theorem by induction on the proof tree size of evaluation rule. We show the representative cases `(EUNBOX)`, `(EBOX)`, and `(EHANDLE)` in Appendix A. ■

4 Conclusion

We have presented type and effect system for multi-staged language with exceptions. The proposed type and effect system checks if we safely synthesize complex controls with exceptions (long jumps) in multi-staged programming. The proposed exception constructs in multi-staged programming has no artificial restriction. Exception-raise and -handle expressions can appear in expressions of any stage. Exceptions can be raised during code composition and may escape stages and can be handled only at stage 0. Our effect type system support such features and is proven safe that empty effect means the input program has no uncaught exceptions during its evaluation. The obvious next step is to extend our system to support the let-polymorphism and imperative operations.

References

1. Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 242–257. ACM, Jan 1996.
2. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.
3. Paul Graham. *On Lisp: an advanced techniques for Common Lisp.* Prentice Hall, 1994.
4. Juan Carlos Guzmán and Ascánder Suárez. A type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1994.
5. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, 1993.
6. Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 257–268. ACM, January 2006.
7. M. Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 137–148. ACM Press, June 1996.
8. Aleksandar Nanevski. A modal calculus for exception handling. In *Proceedings of the Intuitionistic Modal Logic and Applications Workshop*, June 2005.
9. François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 276–290, January 1999.
10. Massimilian Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. C and tcc:a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21:324–369, March 1999.
11. Guy L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.
12. Jean-Piere Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
13. Jean-Pierre Talpin. *Theoretical and Practical Aspects of Type and Effect Inference.* PhD thesis, University of Paris VI, May 1993.
14. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

A Proof of Soundness Theorem

Theorem 1 (Soundness). *Suppose* $\varnothing \Gamma_1 \cdots \Gamma_n \vdash e : A, \varphi$.

1. *If* $e \xrightarrow{n} v$ *then* $\varnothing \Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi$.
2. *If* $e \xrightarrow{n} \overline{c}$ *then* $\varphi \supseteq \{c^n\}$.

Proof. By induction on the proof tree size of evaluation rule $\xrightarrow{n}$. We prove the representative cases `(EEVAL)`, `(EUNBOX)`, `(EBOX)`, and `(EHANDLE)`. We can similarly prove other cases

- `(EEVAL)`

- Case for $\texttt{eval}\ e \xrightarrow{0} v^0$.

(1)	$\varnothing \vdash \texttt{eval}\ e : A, \varphi \cup \varphi'$	Assumption
(2)	$e \xrightarrow{0} \texttt{box}\ v^1$	By (EEVAL)
(3)	$v^1 \xrightarrow{0} v^0$	By (EEVAL)
(4)	$\varnothing \vdash e : \Box(\varnothing \rhd A, \varphi), \emptyset$	By (TEVAL),Lemma 2
(5)	$\varnothing \vdash \texttt{box}\ v^1 : \Box(\varnothing \rhd A, \varphi), \emptyset$	By I.H. (induction hypothesis)
(6)	$\varnothing\varnothing \vdash v^1 : A, \varphi$	By (TBOX)
(7)	$\varnothing \vdash v^1 : A, \varphi$	By Lemma 1
(8)	$\varnothing \vdash v^0 : A, \varphi$	By I.H.
(9)	$\varnothing \vdash v^0 : A, \varphi \cup \varphi'$	By (TSUB)

- Case for $\texttt{eval}\ e \xrightarrow{n} \texttt{eval}\ v$ where $n > 0$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{eval}\ e : A, \varphi \cup \varphi'$	Assumption
(2)	$e \xrightarrow{n} v$	By (EEVAL)
(3)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash e : \Box(\varnothing \rhd A, \varphi), \varphi'$	By (TEVAL)
(4)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash v : \Box(\varnothing \rhd A, \varphi), \varphi'$	By I.H.
(5)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{eval}\ v : A, \varphi \cup \varphi'$	By (TEVAL)

- Case for $\texttt{eval}\ e \xrightarrow{n} \overline{c}$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{eval}\ e : A, \varphi \cup \varphi'$	Assumption
(2)	$e \xrightarrow{n} \overline{c}$	By (EEVAL)
(3)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash e : \Box(\varnothing \rhd A, \varphi), \varphi'$	By (TEVAL)
(4)	$\varphi' \supseteq \{c^n\}$	By I.H.
(5)	$\varphi \cup \varphi' \supseteq \{c^n\}$	By (4)

– (EUNBOX)

- Case for $\texttt{unbox}_n\ e \xrightarrow{n} v$ where $n > 0$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{unbox}_n\ e : A, \varphi \cup (\uparrow_n \varphi')$	Assumption
(2)	$e \xrightarrow{0} \texttt{box}\ v$	By (EUNBOX)
(3)	$\varnothing \vdash e : \Box(\Gamma_n \rhd A, \varphi), \varphi'$	By (TUNBOX)
(4)	$\varnothing \vdash \texttt{box}\ v : \Box(\Gamma_n \rhd A, \varphi), \emptyset$	By I.H.,Lemma 2
(5)	$\varnothing\Gamma_n \vdash v : A, \varphi$	By (TBOX)
(6)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi$	By Lemma 1
(7)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash v : A, \varphi \cup (\uparrow_n \varphi')$	By (TSUB)

- Case for $\texttt{unbox}_k\ e \xrightarrow{n} \texttt{unbox}_k\ v$ where $n > k \geq 0$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{unbox}_k\ e : A, \varphi \cup (\uparrow_k \varphi')$	Assumption
(2)	$e \xrightarrow{n-k} v$	By (EUNBOX)
(3)	$\varnothing\Gamma_1 \cdots \Gamma_{n-k} \vdash e : \Box(\Gamma_n \rhd A, \varphi), \varphi'$	By (TUNBOX)
(4)	$\varnothing\Gamma_1 \cdots \Gamma_{n-k} \vdash v : \Box(\Gamma_n \rhd A, \varphi), \varphi'$	By I.H.
(5)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{unbox}_k\ v : A, \varphi \cup (\uparrow_k \varphi')$	By (TUNBOX)

- Case for $\texttt{unbox}_k\ e \xrightarrow{n} \overline{c}$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{unbox}_k\ e : A, \varphi \cup (\uparrow_k \varphi')$	Assumption
(2)	$e \xrightarrow{n-k} \overline{c}$	By (EUNBOX)
(3)	$\varnothing\Gamma_1 \cdots \Gamma_{n-k} \vdash e : \Box(\Gamma_n \rhd A, \varphi), \varphi'$	By (TUNBOX)
(4)	$\varphi' \supseteq \{c^{n-k}\}$	By I.H.
(5)	$\uparrow_k \varphi' \supseteq \{c^n\}$	By definition of $\uparrow_k$
(6)	$\varphi \cup (\uparrow_k \varphi') \supseteq \{c^n\}$	By (5)

– (EBOX)

- Case for $\texttt{box}\ e \xrightarrow{n} \texttt{box}\ v$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{box}\ e : \Box(\Gamma \rhd A, \varphi^{=0}), \downarrow \varphi^{>0}$	Assumption
(2)	$e \xrightarrow{n+1} v$	By (EBOX)
(3)	$\varnothing\Gamma_1 \cdots \Gamma_n\Gamma \vdash e : A, \varphi$	By (TBOX)
(4)	$\varnothing\Gamma_1 \cdots \Gamma_n\Gamma \vdash v : A, \varphi$	By I.H.
(5)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{box}\ v : \Box(\Gamma \rhd A, \varphi^{=0}), \downarrow \varphi^{>0}$	By (TBOX)

- Case for $\texttt{box}\ e \xrightarrow{n} \overline{c}$.

(1)	$\varnothing\Gamma_1 \cdots \Gamma_n \vdash \texttt{box}\ e : \Box(\Gamma \rhd A, \varphi^{=0}), \downarrow \varphi^{>0}$	Assumption
(2)	$e \xrightarrow{n+1} \overline{c}$	By (EBOX)
(3)	$\varnothing\Gamma_1 \cdots \Gamma_n\Gamma \vdash e : A, \varphi$	By (TBOX)
(4)	$\varphi \supseteq \{c^{n+1}\}$	By I.H.
(5)	$\varphi^{>0} \supseteq \{c^{n+1}\}$	By definition of $\varphi^{>0}$
(6)	$\downarrow \varphi^{>0} \supseteq \{c^n\}$	By definition of $\downarrow$

– (EHANDLE)

- Case for $\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{0} v$.

(1)	$\varnothing \vdash \texttt{handle}\ e_1\ c\ e_2 : A, \varphi$	Assumption
(2)	$\varnothing \vdash e_1 : A, \varphi_1$	By (THANDLE)
(3)	$\varnothing \vdash e_2 : A, \varphi_2$	By (THANDLE)
(4)	$\varphi = (\varphi_1 \setminus \{c^0\}) \cup \varphi_2$	By (THANDLE)

* $e_1 \xrightarrow{0} v$.
  (5) $\varnothing \vdash v : A, \varphi_1$ By I.H.
  (6) $\varnothing \vdash v : A, \emptyset$ By Lemma 2
  (7) $\varnothing \vdash v : A, \varphi$ By (TSUB)
* $e_1 \xrightarrow{0} \overline{c}$ and $e_2 \xrightarrow{0} v$.
  (5) $\varnothing \vdash v : A, \varphi_2$ By I.H.
  (6) $\varnothing \vdash v : A, \varphi$ By (TSUB)

- Case for $\texttt{handle}\ e_1\ c'\ e_2 \xrightarrow{0} \overline{c}$.
  (1) $\varnothing \vdash \texttt{handle}\ e_1\ c'\ e_2 : A, \varphi$ Assumption
  (2) $e_1 \xrightarrow{0} \overline{c}$ By (EHANDLE)
  (3) $\varnothing \vdash e_1 : A, \varphi_1$ By (THANDLE)
  (4) $\varnothing \vdash e_2 : A, \varphi_2$ By (THANDLE)
  (5) $\varphi = (\varphi_1 \setminus \{c'^0\}) \cup \varphi_2$ By (THANDLE)
  (6) $\varphi_1 \supseteq \{c^0\}$ By I.H.
  (7) $\varphi \supseteq (\varphi_1 \setminus \{c'^0\}) \cup \varphi_2 \supseteq \{c^0\}$ By (6)
- Case for $\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{n} v_2$ where $n > 0$.
  (1) $\Gamma_0 \cdots \Gamma_n \vdash \texttt{handle}\ e_1\ c\ e_2 : A, \varphi$ Assumption
  (2) $e_1 \xrightarrow{n} v_1$ By (EHANDLE)
  (3) $e_2 \xrightarrow{n} v_2$ By (EHANDLE)
  (4) $\Gamma_0 \cdots \Gamma_n \vdash e_1 : A, \varphi_1$ By (THANDLE)
  (5) $\Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi_2$ By (THANDLE)
  (6) $\varphi = (\varphi_1 \setminus \{c^0\}) \cup \varphi_2$ By (THANDLE)
  (7) $\Gamma_0 \cdots \Gamma_n \vdash v_1 : A, \varphi_1$ By I.H.
  (8) $\Gamma_0 \cdots \Gamma_n \vdash v_2 : A, \varphi_2$ By I.H.
  (9) $\Gamma_0 \cdots \Gamma_n \vdash v_2 : A, \varphi$ By (TSUB)
- Case for $\texttt{handle}\ e_1\ c\ e_2 \xrightarrow{n} \overline{c}$ where $n > 0$.
  (1) $\Gamma_0 \cdots \Gamma_n \vdash \texttt{handle}\ e_1\ c\ e_2 : A, \varphi$ Assumption
  (2) $\Gamma_0 \cdots \Gamma_n \vdash e_1 : A, \varphi_1$ By (THANDLE)
  (3) $\Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi_2$ By (THANDLE)
  (4) $\varphi = (\varphi_1 \setminus \{c^0\}) \cup \varphi_2$ By (THANDLE)
  * $e_1 \xrightarrow{n} \overline{c}$.
    (5) $\varphi_1 \supseteq \{c^n\}$ By I.H.
    (6) $\varphi = (\varphi_1 \setminus \{c^0\}) \cup \varphi_2 \supseteq \{c^n\}$ By (5)
  * $e_2 \xrightarrow{n} \overline{c}$.
    (5) $\varphi_2 \supseteq \{c^n\}$ By I.H.
    (6) $\varphi = (\varphi_1 \setminus \{c^0\}) \cup \varphi_2 \supseteq \{c^n\}$ By (5)
- Case for $\texttt{handle}\ e_1\ c'\ e_2 \xrightarrow{n} \overline{c}$ where $n > 0$.
  (1) $\Gamma_0 \cdots \Gamma_n \vdash \texttt{handle}\ e_1\ c\ e_2 : A, \varphi$ Assumption
  (2) $\Gamma_0 \cdots \Gamma_n \vdash e_1 : A, \varphi_1$ By (THANDLE)
  (3) $\Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \varphi_2$ By (THANDLE)
  (4) $\varphi = (\varphi_1 \setminus \{c'^0\}) \cup \varphi_2$ By (THANDLE)
  * $e_1 \xrightarrow{n} \overline{c}$.
    (5) $\varphi_1 \supseteq \{c^n\}$ By I.H.
    (6) $\varphi = (\varphi_1 \setminus \{c'^0\}) \cup \varphi_2 \supseteq \{c^n\}$ By (5)
  * $e_2 \xrightarrow{n} \overline{c}$.
    (5) $\varphi_2 \supseteq \{c^n\}$ By I.H.
    (6) $\varphi = (\varphi_1 \setminus \{c'^0\}) \cup \varphi_2 \supseteq \{c^n\}$ By (5)

■