(Early) Effect Systems and Exception Inference— Lecture 1/3

Martin Elsman, DIKU

Seminar at Chalmers University of Technology — December 18–19, 2018

- What is an effect system
- Examples of effect systems
- Exception inference for ML
- Polymorphic exception inference
- Generative exceptions in Standard ML
- Exercises

What is an effect system

- Idea: Extend a traditional type system to capture a safe approximation of what happens (the effect) when a program executes.
- In a functional language setting, the arrow type for functions

$$\tau \to \tau'$$

  is extended with an *effect* φ, which captures a safe approximation of what the function does when it is executed:

$$\tau \xrightarrow{\varphi} \tau'$$

CAVEAT!:

Newer type systems for *algebraic effect handling* (such as those found in Eff, Koka, and Frank) build on these early ideas.

Examples of (early) effect systems

- **Call-tracking analysis.**

  Analyse for each subexpression in a program which function abstractions are applied during its evaluation. Useful for higher-order dead code elimination.

- **Region inference.**

  Organize memory as a *stack of regions*. Insert allocation and deallocation directives in the program, based on an analysis that for each subexpression in the program figures out which regions are written into and which regions are read.

- **Exception inference.**

  Analyse for each subexpression in a program which exceptions may be raised during its evaluation.

A basic language with exception support

v	$::=$	$\lambda x.e$	Functions			
	$\mid$	c	Constant	c	$\in$ C	Constants
e	$::=$	x	Variable			
	$\mid$	v	Value	v	$\in$ V	Variables
	$\mid$	$e_1\ e_2$	Application			
	$\mid$	$\texttt{raise}\ \varepsilon$	Raise exception	ε	$\in$ Σ	Exceptions
	$\mid$	$e\ \texttt{handle}\ \varepsilon \Rightarrow e'$	Exception handling			

EXAMPLE PROGRAMS:

- The program "$(\lambda x.\texttt{raise}\ \varepsilon)\ 5$" raises the exception ε.
- The program "$(\lambda x.\texttt{raise}\ \varepsilon)\ 5\ \texttt{handle}\ \varepsilon \Rightarrow 3$" evaluates to the value 3.
- The program "$(\lambda x.\texttt{raise}\ \varepsilon')\ 5\ \texttt{handle}\ \varepsilon \Rightarrow 3$" raises the exception ε'.

Evaluation

Big-step rules

$$e \rightsquigarrow v / \texttt{raise}\ \varepsilon$$

$$\frac{e_1 \rightsquigarrow \texttt{raise}\ \varepsilon}{e_1\ e_2 \rightsquigarrow \texttt{raise}\ \varepsilon}\ (1) \qquad \frac{e_1 \rightsquigarrow \lambda x.e \quad e_2 \rightsquigarrow v \quad e[v/x] \rightsquigarrow w}{e_1\ e_2 \rightsquigarrow w}\ (2)$$

$$\frac{e_1 \rightsquigarrow \lambda x.e \quad e_2 \rightsquigarrow \texttt{raise}\ \varepsilon}{e_1\ e_2 \rightsquigarrow \texttt{raise}\ \varepsilon}\ (3) \qquad \frac{}{w \rightsquigarrow w}\ (4)$$

$$\frac{e \rightsquigarrow \texttt{raise}\ \varepsilon' \quad \varepsilon' \neq \varepsilon}{e\ \texttt{handle}\ \varepsilon \texttt{=>} e' \rightsquigarrow \texttt{raise}\ \varepsilon'}\ (5) \qquad \frac{e \rightsquigarrow \texttt{raise}\ \varepsilon \quad e' \rightsquigarrow w}{e\ \texttt{handle}\ \varepsilon \texttt{=>} e' \rightsquigarrow w}\ (6)$$

$$\frac{e \rightsquigarrow v}{e\ \texttt{handle}\ \varepsilon \texttt{=>} e' \rightsquigarrow v}\ (7)$$

Effects

- An *effect* φ is a set of exceptions.
- Later we refine the notion of effect to allow for effect polymorphism.

Types:

- Types are defined according to the following grammar:

$$\begin{array}{rcll} \tau & ::= & \texttt{int} & \text{Integers} \\ & | & \tau \xrightarrow{\varphi} \tau' & \text{Function types} \end{array}$$

- We use T to denote the set of all types.

Type environments:

- A *type environment* $\Gamma : V \to T$ maps program variables to types.

The TypeOf function

We allow predefined functions and constants by the use of a function

TypeOf : $C \to T$, which maps constants to types.

Examples of predefined functions and constants:

$$\begin{array}{rcll} \mathit{div} & : & \texttt{int} * \texttt{int} \xrightarrow{\{\mathit{Div}\}} \texttt{int} & \text{Integer division} \\ +, -, * & : & \texttt{int} * \texttt{int} \to \texttt{int} & \text{Integer addition, subtraction, ...} \end{array}$$

To model Standard ML precisely...

$$\begin{array}{rcll} \mathit{div} & : & \texttt{int} * \texttt{int} \xrightarrow{\{\mathit{Div},\mathit{Overflow}\}} \texttt{int} & \text{Integer division} \\ +, -, * & : & \texttt{int} * \texttt{int} \xrightarrow{\{\mathit{Overflow}\}} \texttt{int} & \text{Integer addition, subtraction, ...} \end{array}$$

Assumption: Addition of pairs and evaluation rules for application of constants.

A simple type-and-effect system for exceptions

Typing rules allow inference of sentences of the form $\Gamma \vdash e : \tau, \varphi$, which are read: In the type environment Γ, the expression e has type τ and effect φ.

TYPING RULES $\boxed{\Gamma \vdash e : \tau, \varphi}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \varphi} \text{ (8)} \qquad \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\varphi} \tau', \varphi_1 \quad \Gamma \vdash e_2 : \tau, \varphi_2}{\Gamma \vdash e_1\ e_2 : \tau', \varphi_1 \cup \varphi_2 \cup \varphi} \text{ (9)}$$

$$\frac{}{\Gamma \vdash c : \mathsf{TypeOf}(c), \varphi} \text{ (10)} \qquad \frac{\Gamma + \{x : \tau\} \vdash e : \tau', \varphi'}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{\varphi'} \tau', \varphi} \text{ (11)}$$

$$\frac{\varepsilon \in \varphi}{\Gamma \vdash \texttt{raise}\ \varepsilon : \tau, \varphi} \text{ (12)} \qquad \frac{\Gamma \vdash e : \tau, \varphi \quad \Gamma \vdash e' : \tau, \varphi'}{\Gamma \vdash e\ \texttt{handle}\ \varepsilon\ \texttt{=>}\ e' : \tau, \varphi \setminus \{\varepsilon\} \cup \varphi'} \text{ (13)}$$

Type and effect properties

The type and effect system approximates the dynamic effects of programs:

Lemma 0 (Value substitution) If $\Gamma + \{x : \tau\} \vdash e : \tau', \varphi$ and $\vdash v : \tau, \emptyset$ then $\Gamma \vdash e[v/x] : \tau', \varphi$.

Proof: By induction on the derivation $\Gamma + \{x : \tau\} \vdash e : \tau', \varphi$.

Lemma 1 (Type and effect soundness)

If $\vdash e : \tau, \varphi$ then either

① $e \rightsquigarrow v$ and $\vdash v : \tau, \emptyset$; or

② $e \rightsquigarrow \texttt{raise}\, \varepsilon$ and $\varepsilon \in \varphi$

Proof: By induction on the derivation $\vdash e : \tau, \varphi$ using Lemma 0 in the case of rule 9.

Type and effect polymorphism

- Allow a function to take a different type in different contexts.
- Effect polymorphism allows a higher-order function to take functions with different effects as arguments.
- Without effect polymorphism, the type of a let-bound function depends on the call sites.

EXAMPLES:

- Some generic constants:

$$\begin{array}{rcll} \mathit{hd} & : & \forall\alpha.\alpha\ \texttt{list} \xrightarrow{\{\mathit{Empty}\}} \alpha & \text{Head of a list} \\ \mathit{tl} & : & \forall\alpha.\alpha\ \texttt{list} \xrightarrow{\{\mathit{Empty}\}} \alpha\ \texttt{list} & \text{Tail of a list} \end{array}$$

- The apply function $\lambda f.\lambda x.f\ x$ can be given the type

$$\forall\alpha\beta\epsilon.(\alpha \xrightarrow{\{\epsilon\}} \beta) \xrightarrow{\emptyset} \alpha \xrightarrow{\{\epsilon\}} \beta$$

Refining exception inference to support polymorphism

- To support polymorphism, we assume denumerably sets of *type variables* (α) and *effect variables* (ϵ).
- Effects φ are refined to be sets of exceptions and effect variables.

TYPES AND TYPE SCHEMES:

- Types are now defined according to the following grammar:

$$\begin{array}{rcll} \tau & ::= & \texttt{int} & \text{Integers} \\ & | & \alpha & \text{Type variable} \\ & | & \tau \xrightarrow{\varphi} \tau' & \text{Function types} \\ \sigma & ::= & \forall \vec{\alpha}\vec{\epsilon}.\tau & \text{Type schemes} \end{array}$$

- TS denotes the set of all type schemes and we sometimes write τ to mean $\forall.\tau$.

TYPE ENVIRONMENTS:

- A *type environment* $\Gamma : V \to TS$ now maps program variables to type schemes.

Substitutions

- A *type substitution* S_{t} maps type variables to types.
- An *effect substitution* S_{e} maps effect variables to effects.
- A *substitution* $S = (S_{\mathrm{t}}, S_{\mathrm{e}})$ is a pair of a type substitution and an effect substitution.
- The effect of applying a substitution to some object is the effect of applying the type substitution and the effect substitution on the object simultaneously.

$$S(\tau \xrightarrow{\varphi} \tau') = S(\tau) \xrightarrow{S(\varphi)} S(\tau') \qquad S(\texttt{int}) = \texttt{int} \qquad (S_{\mathrm{t}}, S_{\mathrm{e}})(\alpha) = S_{\mathrm{t}}(\alpha)$$

$$(S_{\mathrm{t}}, S_{\mathrm{e}})(\varphi) = \{\varepsilon \mid \varepsilon \in \varphi\} \cup \bigcup \{\varphi' \mid \varphi' \in S_{\mathrm{e}}(\epsilon) \wedge \epsilon \in \varphi\}$$

INSTANTIATION:

A type τ is an instance of a type scheme $\sigma = \forall \vec{\alpha}\vec{\epsilon}.\tau'$, written $\sigma \succ \tau$, if there exists a substitution $S = ([\vec{\tau}/\vec{\alpha}], [\vec{\varphi}/\vec{\epsilon}])$, for some $\vec{\tau}$ and $\vec{\varphi}$, such that $S(\tau') = \tau$.

Let-polymorphism

To allow polymorphism in programs, we extend the language with a let-construct with appropriate evaluation rules:

$$\boxed{e \rightsquigarrow v / \texttt{raise}\ \varepsilon}$$

$$\frac{e \rightsquigarrow \texttt{raise}\ \varepsilon}{\texttt{let}\ x = e\ \texttt{in}\ e' \rightsquigarrow \texttt{raise}\ \varepsilon}\ (14) \qquad \frac{e \rightsquigarrow v \quad e'[v/x] \rightsquigarrow w}{\texttt{let}\ x = e\ \texttt{in}\ e' \rightsquigarrow w}\ (15)$$

Refined typing rules:

With addition of a typing rule for the `let`-construct, only the typing rule for variables need be modified:

$$\frac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau}\ (16) \qquad \frac{\Gamma \vdash e : \tau, \varphi \quad \{\vec{\alpha}\vec{\epsilon}\} \cap \mathrm{fv}(\Gamma, \varphi, \varphi') = \emptyset \quad \Gamma + \{x : \forall \vec{\alpha}\vec{\epsilon}.\tau\} \vdash e' : \tau', \varphi'}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \tau', \varphi \cup \varphi'}\ (17)$$

Properties of the refined system

Lemma 2 (Instantiation is closed under substitution)

If $\sigma \succ \tau$ then $S(\sigma) \succ S(\tau)$, for any substitution S.

Lemma 3 (Typing judgments are closed under substitution)

If $\Gamma \vdash e : \tau, \varphi$ then $S(\Gamma) \vdash e : S(\tau), S(\varphi)$.

Lemma 4 (Value substitution)

If $\Gamma + \{x : \sigma\} \vdash e : \tau', \varphi$ and $\vdash v : \tau, \emptyset$ and $\sigma \succ \tau$ then $\Gamma \vdash e[v/x] : \tau', \varphi$.

Lemma 5 (Type soundness)

If $\vdash e : \tau, \varphi$ then either

① $e \rightsquigarrow v$ and $\vdash v : \tau, \emptyset$; or

② $e \rightsquigarrow \texttt{raise}\, \varepsilon$ and $\varepsilon \in \varphi$

Proof: By induction on the derivation $\vdash e : \tau, \varphi$.

Possible extensions to basic exception inference

- Recursive functions (use a small-step operational semantics)
- First class exception constructors
- Value-carrying exception constructors
- Generative exceptions as for Standard ML

The first three extensions are covered by:

[1] F. Pessaux and X. Leroy. **Type-based analysis of uncaught exceptions**. POPL '99.

Other related work include:

[2] M. Fähndrich, J. S. Foster, Alexander Aiken, and J. Cu. **Tracking down Exceptions in Standard ML Programs**. Tech Report, UC Berkeley. 1998.

[3] J. C. Guzmán and A. Suárez. **An extended type system for exceptions**. ML '94.

Generative exceptions in Standard ML

- Exception constructors are introduced by explicit exception declarations:

  `exception E`

- When such a declaration is evaluated, a new distinct *exception name* is associated with the exception constructor `E` at runtime.

- Matches in `handle`-constructs look up exception names at runtime to identify exceptions.

EXAMPLE:

```
fun f x = let exception E                                    1
          in  (E, fn y => (raise y) handle E => x)           2
          end                                                3
val a = #1 (f 5)                                             4
val g = #2 (f 10)                                            5
val ? = g a                                                  6
```

Generative exceptions in Standard ML — continued...

- In the precense of locally declared, value carrying exceptions, the generative nature of exceptions is essential for soundness.

EXAMPLE:

```
fun f (x:'a) : exn * (exn -> 'a) =                                    1
  let exception E of 'a                                               2
  in (E x, fn y => (raise y) handle E v => v)                         3
  end                                                                 4
val a : exn = #1 (f 5)                                                5
val g : exn -> bool = #2 (f true)                                     6
val ? : bool = g a                                                    7
```

- The call `g a` in the example above raises an exception `E 5`, which should not be caught in a context where the argument is expected to be a boolean.

Towards a substitution-based exception inference algorithm

Problem with the typing rules:

- In an inference algorithm based on the typing rules, we need a way to unify types, including higher-order types, with effects.

Solution:

- Refine the notion of function types to take the form $\tau \xrightarrow{\epsilon.\varphi} \tau'$.
- Refine the notion of substitution to map effect variables ϵ to arrow effects $\epsilon.\varphi$:

$$S(\epsilon.\varphi) = \epsilon'.(S(\varphi) \cup \varphi') \text{ where } S(\epsilon) = \epsilon'.\varphi'$$

- In this way, substitutions can work as unifiers.

Example:

- The substitution $\{\epsilon \mapsto \epsilon.\{\mathit{Overflow}, \mathit{Div}\}, \epsilon' \mapsto \epsilon.\{\mathit{Overflow}, \mathit{Div}\}\}$ is a unifier for the two arrow effects $\epsilon.\{\mathit{Div}\}$ and $\epsilon'.\{\mathit{Overflow}\}$.

Exercises

① Add pairs and conditionals to the system.

② Prove Lemma 1 on slide 1-9.

③ More advanced: prove the lemmas on slide 1-14.