Control structures, seventh lecture

Static typing of effects

Xavier Leroy

2024-03-07

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

Static typing

A widely-used way to guarantee useful properties of programs, by static analysis before execution.

- Data integrity.
  (Absence of errors caused by wrong use of data, such as “returning a pair when a triple is expected”, or “uninitialized field in a record”.)
- Exhaustiveness of control.
  (Does a pattern matching covers all possible cases? Are all raised exceptions caught? Are all algebraic effects handled?)
- Termination of programs.
  (Crucial for logical formalisms such as Coq, Lean, Agda. Irrelevant for Turing-complete languages.)

Static typing

A widely-used way to describe the interface of software components (functions, classes, modules, libraries, ...).

Must not reveal too many details of the implementation, so that it can evolve later.

$\rightarrow$ Types as an abstraction barrier.

Two questions in this lecture

When we have advanced control structures such as exceptions, control operators, algebraic effects and effect handlers:

Q1: Does “classic” static typing still guarantee data integrity? Which types can we give to these control structures?

Q2: Can static typing be extended to guarantee exhaustiveness of control?

Typing values in the presence of advanced control structures

A type system for a language with exceptions

A type `exn` of exception values.

$$\frac{\Gamma \vdash e : \mathtt{exn}}{\Gamma \vdash \mathtt{raise}\ e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \mathtt{exn} \vdash e_2 : \tau}{\Gamma \vdash \mathtt{try}\ e_1\ \mathtt{with}\ x \to e_2 : \tau}$$

`raise` e never returns, therefore it has all possible types τ.
(Alternative: a function $\mathtt{raise} : \forall\alpha,\ \mathtt{exn} \to \alpha$.)

Declaring an exception, functional view =
adding a constructor to the type `exn`.

Declaring an exception, object-oriented view =
define a sub-class of type `exn` (which is called `Throwable` in Java).

Type safety

Either by a direct proof of safety, using the reduction semantics under contexts of lecture #5.

Or by noticing that the ERS (*Exception-Returning Style*) transformation from lecture #5 preserves typing:

$$\text{if} \quad \Gamma \vdash e : \tau \quad \text{then} \quad \Gamma^* \vdash \mathcal{E}(e) : \tau^* + \mathtt{exn}$$

Translating the types of values:

$$\begin{aligned} \iota^* &= \iota \quad \text{for all base types } \iota \\ (\sigma \to \tau)^* &= \sigma^* \to \tau^* + \mathtt{exn} \end{aligned}$$

A type system for a language with effect handlers (a la OCaml 5)

A type α `eff` of effects returning a value of type α.
Declaring an effect = adding a constructor.

$$\frac{\Gamma \vdash e : \tau\ \texttt{eff}}{\Gamma \vdash \texttt{perform}\ e : \tau}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash e_{ret} : \sigma \to \tau \qquad \Gamma \vdash e_{eff} : \forall \alpha,\ \alpha\ \texttt{eff} \to (\alpha \to \tau) \to \tau}{\Gamma \vdash \texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff} : \tau}$$

The e_{eff} part of the handler receives as arguments an arbitrary effect (type α `eff`) and a delimited continuation that expects a value of type α.

Typing of `callcc`

`callcc` $(\lambda k.\, e)$ binds *k* to the continuation of `callcc`, then evaluates *e*.
If *e* terminates normally, its value is that of the `callcc`.
If *e* applies *k* to *v*, the `callcc` terminates with value *v*.

Hence the following type for the `callcc` operator:

$$\texttt{callcc} : \ \forall\alpha,\ (\underbrace{\alpha\ \texttt{cont}}_{\text{type of } k} \to \alpha) \to \alpha$$

type of *e*

Typing of `callcc`

$$\texttt{callcc} : \forall\alpha,\ (\alpha\ \texttt{cont} \to \alpha) \to \alpha$$

The type α `cont` is the type of continuations that expect a value of type α.

Traditionally, it's a type for a function that never returns, such as

$$\alpha\ \texttt{cont} \overset{\textit{def}}{=} \alpha \to \texttt{empty} \quad \text{or} \quad \alpha\ \texttt{cont} \overset{\textit{def}}{=} \alpha \to (\forall\beta.\ \beta)$$

In SML/NJ, `cont` is an abstract type, and a `throw` operator is provided to invoke a continuation:

$$\texttt{throw} : \forall\alpha\,\beta,\ \alpha\ \texttt{cont} \to \alpha \to \beta$$

Safety of this simple typing

Either by a direct proof using the reduction semantics under contexts from lecture #4.

Or by observing that the CPS (*Continuation-Passing Style*) transformation from lecture #4 preserves simple types (no polymorphism):

$$\text{if} \quad \Gamma \vdash e : \tau \quad \text{then} \quad \Gamma^* \vdash \mathcal{C}(e) : (\tau^* \to R) \to R$$

R is the “result type”, i.e. the type of the whole program.

Translation of value types:

$$\begin{aligned} \iota^* &= \iota \quad \text{for all base types } \iota \\ (\sigma \to \tau)^* &= \sigma^* \to (\tau^* \to R) \to R \end{aligned}$$

An issue with parametric polymorphism

In the languages of the ML/Haskell family, a value bound by `let` receives a type schema and can be used with several different instances of this type schema.

```
let x = [] in          (* x : ∀α, α list *)
... 12 :: x ...        (* x used as an int list *)
... "hello" :: x ...   (* x used as a string list *)
```

The issue with polymorphic references

This form of polymorphism is unsafe for polymorphic references:

```
let f = ref (fun x -> x) in
f := (fun x -> x + 1);  !f "hello"
```

If we give `f` the type $\forall \alpha,\ (\alpha \to \alpha)$ `ref`, this code is well typed, but crashes on evaluating `"hello" + 1`.

Hence the restriction of generalization to values:
then, `ref (fun x -> x)` is not a value,
therefore `f` has monomorphic type $(\alpha \to \alpha)$ `ref` for some type α,
and one of the two uses of `f` fails to typecheck.

A similar issue with `callcc`

```
type 'a attempt = { current: 'a; retry: 'a -> unit }
let r =
    callcc (fun k ->
        let rec retry f =
            throw k { curr = f; retry = retry } in
        { curr = (fun x -> x); retry = retry })
in
    r.current "hello";
    r.retry (fun x -> x + 1)
```

It was long believed that this problem with polymorphism was specific to mutable state, until Harper and Lillibridge (1991) showed a counter-example involving only `callcc`.

A similar issue with `callcc`

```
type 'a attempt = { current: 'a; retry: 'a -> unit }
let r =
    callcc (fun k ->
        let rec retry f =
            throw k { curr = f; retry = retry } in
        { curr = (fun x -> x); retry = retry })
in
    r.current "hello";
    r.retry (fun x -> x + 1)
```

If `r` is given the polymorphic type $\forall\alpha,\ (\alpha \to \alpha)$ `attempt`,
`r.current "hello"` is executed twice,
once with `r.current = fun x -> x`,
once with `r.current = fun x -> x + 1`.

A similar issue with `callcc`

```
type 'a attempt = { current: 'a; retry: 'a -> unit }
let r =
    callcc (fun k ->
        let rec retry f =
            throw k { curr = f; retry = retry } in
        { curr = (fun x -> x); retry = retry })
in
    r.current "hello";
    r.retry (fun x -> x + 1)
```

The value restriction avoids this issue:
since `callcc (fun k -> ...)` is not a value, its type is not generalized, and the uses of `r` are rejected by typechecking.

A similar issue with exceptions

Even exceptions would be unsafe without the value restriction!

```
type 'a box =
  { hide: 'a -> (unit -> unit);
    expose: (unit -> unit) -> 'a option }

let makebox (type a) : a box =
  let exception E of a in
  { hide = (fun v -> fun () -> raise (E v));
    expose = (fun f -> try f (); None with E v -> Some v) }

let (x: string option) = makebox.expose (makebox.hide 12)
```

`x` is `Some 12`, but without the value restriction it would have type τ `option` for all τ.

Typing delimited continuations

The “cupto” approach by Gunter, Rémy, Riecke (1995):

- an extensible type α `prompt` of “prompts” of type α (prompts $\approx$ program points $\approx$ labels);
- two primitive functions

$$\begin{aligned} \texttt{set}: &\quad \forall \alpha,\ \alpha\ \texttt{prompt} \to (\texttt{unit} \to \alpha) \to \alpha \\ \texttt{cupto}: &\quad \forall \alpha\, \beta,\ \alpha\ \texttt{prompt} \to ((\beta \to \alpha) \to \alpha) \to \beta \end{aligned}$$

Intuition: `cupto` *p* $(\lambda k.\, e)$ captures the continuation that goes “up to” the nearest delimiter `set` *p* (nearest in the call stack), bind it to *k*, returns to delimiter `set` *p*, then evaluates *e*.

Tracking exceptions via types

The issue with uncaught exceptions

Many programming languages featuring exceptions and exception handlers do not statically guarantee that a raised exception is always handled.

(Lisp family, ML family, Ada, Python, C#, ...)

Therefore, a program can terminate abruptly on an uncaught exception (an exception that is raised but not handled).

Our experience with large ML applications is that uncaught exceptions are the most frequent mode of failure.

(Pessaux & Leroy, 1998)

Declaring and verifying exceptions

An approach known as checked exceptions:

- have programmers annotate each function, procedure, method with a set of exceptions;
- verify (statically or dynamically) that all the exceptions a function can raise (itself or via one of the function it calls) and doesn't handle itself are in this set.

Corollary: a function annotated with the empty set of exceptions never raises an exception.

General idea: the exceptions that a function can raise are part of its interface, just like the types of its arguments and results.

The CLU language (B. Liskov *et al*, MIT, circa 1975)

While it is appropriate for the caller to know about the exceptions signaled by the procedure (and these are part of the abstraction implemented by that procedure), the callee should know nothing about the exceptions signaled by procedures used in the implementation of the invoked procedure.

(Liskov & Snyder, Exception Handling in CLU, 1979)

CLU imposes a strict discipline on the use of exceptions:

- Any exception that escapes from a function must be declared in the type of the function.
- Such an exception must be handled by the caller function. (The caller can elect to re-throw the exception, but there is no automatic propagation of exceptions.)

An example of an exception declaration in CLU

(Liskov & Snyder, *Exception Handling in CLU*, 1979)

```
sign = proc (x: int) returns(int) signals(zero, neg(int))
    if x < 0 then signal neg(x)
    elseif x = 0 then signal zero
    else return(x)
    end
end sign
```

Preferred implementation: using multiple return points, as in Fortran 77. For each declared exception, the caller passes (as extra argument) the code label for the handler of this exception.

Run-time checking of exceptions

If a function raises an exception that is not declared in its `signals` clause, this exception is turned into a fatal error (or, in later CLU versions, in a special `failure` exception that propagates without being checked).

Idea: allow programmers to not declare nor handle exceptions that are “impossible by design”, such as the “empty stack” exception in the following example.

```
if ~ stack$empty(s) then
    ...
    x := stack$pop(s)
    ...
end
```

Exceptions in C++

Structured exceptions were added to C++ circa 1990, using the familiar model where exceptions automatically propagate towards callers.

Functions and methods can optionally declare a set of exceptions that they (or their callees) can raise:

```
int f(int x) throw(my_exception, some_other_exception)
{
    ...
    if (x < 0) throw myex;
    ...
}
```

No `throw` clauses $\Rightarrow$ can raise any exception.

Run-time checking of exceptions

Like in CLU: no static verification of `throw` clauses; an exception that escapes and is not declared in the `throw` clause is turned into a fatal error (a call to `std::unexpected`).

```
int f(int x) throw(myexception)
{
  if (x < 0) throw myex; else return -x;
}
int g(int x) throw()
{
  return f(x);
}
```

No compile-time warning, but `g(-1)` triggers a call to `std::unexpected`.

Abandoning checked exceptions in C++

In the 2000's, a consensus emerged: these `throw` clauses are barely usable.

> *The biggest problem with exception-specifications is that programmers use them as though they have the effect the programmer would like, instead of the effect they actually have.*
> *(Boost library requirements and guidelines)*

C++ 2011 deprecates `throw` clauses and introduces a simplified declaration `noexcept`.

C++ 2017 removes `throw` clauses.

Checked exceptions in Java

`throws` clauses on method definitions and declarations.

```
public void writeList() throws IOException {
    PrintWriter out = new PrintWriter(new FileWriter(...));
    ...
    out.close();
}
```

No `throws` clause $\Rightarrow$ no checked exception can escape.

Special case: exceptions of `RuntimeException` and `Error` classes are unchecked: they do not need to be declared, and they propagate freely.

Checked exceptions in Java

The compiler checks that all exceptions raised by a method *M* or mentioned in the `throws` clauses of called methods are either handled or declared by *M*.

✓
```
void f() throws Exception {
    ... writeList() ...
}
```

✓
```
void f() {
    try { ... writeList() ... }
    catch (IOException e) { }
}
```

✗
```
void f() {
    ... writeList() ...
}
```

Practical difficulties with exception specifications

More difficult to evolve a library without changing its API: newly-introduced exception cannot escape, they must be handled locally or converted into pre-existing exceptions…

Poor scaling: `throws` clauses get very long when we combine several large libraries.

Knee-jerk reactions from programmers: `throws Exception` everywhere; over-use of `RuntimeException` or `Error` unchecked exceptions.

As a consequence, several “post-Java” languages abandon exception specifications: C#, Scala (initially), Kotlin, …

The capability-based approach

(M. Odersky *et al*, in Scala 3; J. Brachthäuser *et al*, in Effekt.)

A shift of perspective: to avoid uncaught exceptions, let's prohibit raising an exception if we're not in the scope of a handler for that exception.

- To raise exception E, we must possess the capability to do so: a special value of type `CanThrow`$[E]$.
- The `try` e_1 `catch case` E construct produces a `CanThrow`$[E]$ capability and makes it available to e_1.
- This capability propagates to the places where we raise exception E using implicit function arguments.

Revisiting exception declarations

```
def m(x: T) : U throws E
```

Don’t read this declaration as “`m` can raise exception `E`”.
Do read it as “`m` needs capability `CanThrow[E]` to possibly raise exception `E`”.

This declaration expands to

```
def m(x: T) (using CanThrow[E]): U
```

It says that `m` has an implicit argument of type `CanThrow[E]`: `m` can be called only from a context containing a value of this type.

Generating, propagating, using the capabilities

```
def m(x: T) : U throws E = // receives a value CanThrow[E]
    ... throw E ...        // uses it to raise E

def p(x: T) : V throws E = // receives a value CanThrow[E]
    ... m(x) ...           // passes it to m

def q(x: T) : V =
    try         // a value CanThrow[E] appears here
        p(x)    // and is passed to p
    catch
        case e : E => 0
```

Capabilities and higher-order functions

With Java-style exception declarations, a higher-order function such as `List.map` must account for the exceptions raised by its argument function `f`:

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

In the capability model, `map` needs no `CanThrow` permissions, since it raises no exceptions itself!

```
class List[A]
    def map[B](f: A => B): List[B]    // no “throws” clause
```

We can evaluate `xs.map(f)` where `f` is a function that can raise `E`. It suffices to have a `CanThrow[E]` capability in the context.

A delicate issue with the capability approach

A capability `CanThrow`[*E*] must not escape the scope of the `try...catch` that created it:

- the capability must not be returned as a result,
- nor stored in a global variable.

$\Rightarrow$ Requires some support for second-class values, whose only possible use is to be passed as implicit arguments.

From exceptions to algebraic effects

An uncaught exception is a problem;
an effect without a handler is probably a bigger problem.

OCaml 5 (until now)

No declarations and no static checks for effects,
no more than for exceptions…

Eff (Pretnar & Bauer)**, Koka** (Leijen)

Use a type and effect system (see next section).

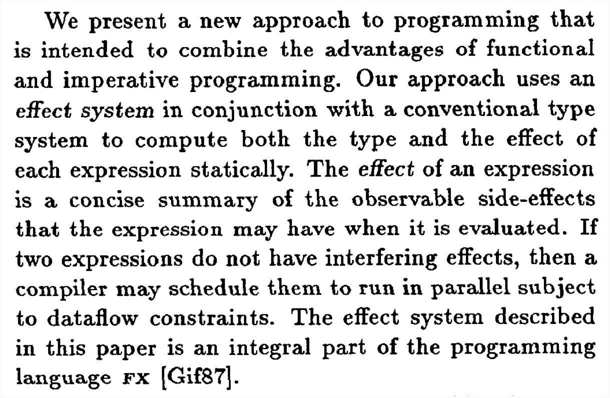
Effekt (Brachthäuser *et al*)

Uses capabilities.

Type and effect systems

Type and effect systems

Use static typing techniques to describe the effects produced by the evaluation of an expression.



We present a new approach to programming that is intended to combine the advantages of functional and imperative programming. Our approach uses an *effect system* in conjunction with a conventional type system to compute both the type and the effect of each expression statically. The *effect* of an expression is a concise summary of the observable side-effects that the expression may have when it is evaluated. If two expressions do not have interfering effects, then a compiler may schedule them to run in parallel subject to dataflow constraints. The effect system described in this paper is an integral part of the programming language FX [Gif87].

(J. Lucassen & D. Gifford, POPL 1988)

Example: typing a language with exceptions

A functional language with `raise` and `try...with`.

Value types:

$$\begin{array}{rcll} \tau, \sigma & ::= & \texttt{int} \mid \texttt{bool} & \text{base types} \\ & \mid & \texttt{exn} & \text{the type of exceptions} \\ & \mid & \sigma \xrightarrow{\varphi} \tau & \text{function type } (\varphi = \text{latent effect}) \end{array}$$

Effect types:

$$\begin{array}{rcll} \varphi & ::= & 0 & \text{pure computation (no exceptions)} \\ & \mid & 1 & \text{computation that can raise an exception} \end{array}$$

Typing judgment:

$$\Gamma \vdash e : \tau \ ! \ \varphi$$

Read: in an environment of type Γ, expression *e* produces a value of type τ and effects of type φ.

Selected typing rules

$$\frac{n \in \{0, 1, 2, 3, \dots\}}{\Gamma \vdash n : \texttt{int} \,!\, \varphi} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash b : \texttt{bool} \,!\, \varphi}$$

Constants are pure (effect $\varphi = 0$), but can also be viewed as impure (effect $\varphi = 1$) if the context demands it.

$$\frac{\Gamma, x : \sigma \vdash e : \tau \,!\, \varphi}{\Gamma \vdash \lambda x.\, e : \sigma \xrightarrow{\varphi} \tau \,!\, \varphi'}$$

A function abstracition is pure. The effect of the function body becomes the latent effect in the type of the function.

Selected typing rules

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \,!\, \varphi \qquad \Gamma \vdash e_2 : \tau \,!\, \varphi \qquad \Gamma \vdash e_3 : \tau \,!\, \varphi}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau \,!\, \varphi}$$

A conditional is pure only if e_1, e_2, e_3 are pure.
The rule forces the 3 expressions to have the same effect type φ.

$$\frac{\Gamma \vdash e_1 : \sigma \xrightarrow{\varphi} \tau \,!\, \varphi \qquad \Gamma \vdash e_2 : \sigma \,!\, \varphi}{\Gamma \vdash e_1\, e_2 : \tau \,!\, \varphi}$$

The application $e_1\ e_2$ combines three effects: the effect of evaluating e_1, the effect of evaluating e_2, and the latent effect of the function.

Typing exception raising and exception handling

$$\frac{\Gamma \vdash e : \texttt{exn} \,!\, \varphi}{\Gamma \vdash \texttt{raise}\; e : \tau \,!\, 1}$$

The `raise` *e* expression has all value types, but only effect 1.

$$\frac{\Gamma \vdash e_1 : \tau \,!\, \varphi_1 \qquad \Gamma, x : \texttt{exn} \vdash e_2 : \tau \,!\, \varphi_2}{\Gamma \vdash \texttt{try}\; e_1 \;\texttt{with}\; x \to e_2 : \tau \,!\, \varphi_2}$$

Exceptions raised by e_1 are caught. Only the exceptions raised by e_2 (effect φ_2) escape the `try...with`.

Other algebras of effect types

Effect types are often sets of base effects F:

$$\varphi ::= \{F_1, \ldots, F_n\}$$

with the empty set meaning purity (no effects at all).
Base effects can be:

- Broad groups of effects: $F ::= \texttt{div} \mid \texttt{state} \mid \texttt{exn} \mid \texttt{ctrl}$ (divergence, state, exceptions, control operators, …).
- Individual effects such as exception names E or algebraic effect names F.
- Names with types, such as $E(\tau)$ or $F(\sigma \twoheadrightarrow \tau)$.
- For mutable state effects, names + types + state regions ρ : $\texttt{alloc}(\tau, \rho) \mid \texttt{read}(\tau, \rho) \mid \texttt{write}(\tau, \rho)$.

Too simple types?

```
let use_pure (f: int $\xrightarrow{0}$ int) =
    ... f 0 ... f 1 ...

let use_impure (f: int $\xrightarrow{1}$ int) =
    if ... then f 0 else raise E

let f (x: int) = x + 1 in
use_pure f + use_impure f
```

There is no way to use the pure function `f` in two contexts, one that expects a pure function and the other that expects an impure function.

We need polymorphism over effect types:
either subtyping polymorphism or parametric polymorphism.

Subtyping polymorphism

A computation / a function can be viewed as having more effects than it really has. Hence the rule of subsumption:

$$\frac{\Gamma \vdash e : \tau \mathrel{!} \varphi \qquad \tau <: \tau' \qquad \varphi \subseteq \varphi'}{\Gamma \vdash e : \tau' \mathrel{!} \varphi'}$$

The subtyping relation $\tau <: \tau'$ is defined as

$$\tau <: \tau \qquad\qquad \frac{\sigma' <: \sigma \qquad \tau <: \tau' \qquad \varphi \subseteq \varphi'}{\sigma \xrightarrow{\varphi} \tau <: \sigma' \xrightarrow{\varphi'} \tau'}$$

Note the contravariance in the argument types:

$\mathtt{int} \xrightarrow{0} \mathtt{int}$ can be viewed with type $\mathtt{int} \xrightarrow{1} \mathtt{int}$, but
$(\mathtt{int} \xrightarrow{1} \mathtt{int}) \xrightarrow{0} \mathtt{bool}$ can be viewed with type $(\mathtt{int} \xrightarrow{0} \mathtt{int}) \xrightarrow{0} \mathtt{bool}$.

The problem with higher-order functions

For a higher-order function such as `List.map` in OCaml, we should be able to use it with the two types

$$(\sigma \xrightarrow{\emptyset} \tau) \xrightarrow{\emptyset} (\sigma\ \texttt{list} \xrightarrow{\emptyset} \tau\ \texttt{list})$$
$$(\sigma \xrightarrow{\varphi} \tau) \xrightarrow{\emptyset} (\sigma\ \texttt{list} \xrightarrow{\varphi} \tau\ \texttt{list}) \quad \text{avec } \varphi \neq \emptyset$$

In the first case, we “map” a pure function, this produces no effects.

In the second case, we “map” an impure function, this produces the same effects φ as the function.

Neither type is subtype of the other!

$\rightarrow$ need for parametric polymorphism.

Parametric polymorphism and rows of effects

(M. Wand, 1989; D. Rémy, 1989.)

A common representation for two kinds of sets:

- closed sets of effects $\{F_1, \ldots, F_n\}$
- extensible sets of effects $\{F_1, \ldots, F_n\} \cup \rho$
  where ρ is a row variable.

This makes it possible to take the union of two extensible sets by instantiating their row variables and unifying them:

$$\begin{tikzcd}
\forall \rho_1,\ \{F; F_1\} \cup \rho_1 \arrow[dr, "\text{inst } \rho_1 \mapsto \{F_2\} \cup \rho"] & & \\
& \{F; F_1; F_2\} \cup \rho \arrow[r, "\text{gen}"] & \forall \rho,\ \{F; F_1; F_2\} \cup \rho \\
\forall \rho_2,\ \{F; F_2\} \cup \rho_2 \arrow[ur, "\text{inst } \rho_2 \mapsto \{F_1\} \cup \rho"'] & &
\end{tikzcd}$$

This gives a form of subsumption that is covariant!

An algebra of rows

(D. Rémy, 1989, 1990, 1993.)

To type-check some operations, and to define unification precisely, it is useful to represent the absence of an element, not just its presence.

Rows:	$\varphi ::= \rho$	row variable
	$\mid \emptyset$	empty raw
	$\mid F : \pi; \varphi$	φ plus element F with presence π
Presence:	$\pi ::= \theta$	presence variable
	$\mid \texttt{Abs}$	absent
	$\mid \texttt{Pre}(\tau)$	present with type τ

Rows are treated modulo permutation and absorption:

$$F_1 : \pi_1; F_2 : \pi_2; \varphi = F_2 : \pi_2; F_1 : \pi_1; \varphi \qquad F : \texttt{Abs}; \emptyset = \emptyset$$

Typing algebraic effects and effect handlers

(D. Hillerström, S. Lindley, 2016, 2018.)

Value types:

$\tau, \sigma ::= \alpha$	variable
$\mid \texttt{int} \mid \texttt{bool}$	base types
$\mid \sigma \xrightarrow{\varphi} \tau$	function types
$\mid \forall\alpha, \tau \mid \forall\rho, \tau \mid \forall\theta, \tau$	polymorphism

Effect types:

$\varphi \quad ::= \rho \mid \emptyset \mid F : \pi; \varphi$	rows of effects F

Presence types:

$$\pi \quad ::= \theta \mid \texttt{Abs} \mid \texttt{Pre}(\sigma \twoheadrightarrow \tau)$$

The notation $\texttt{Pre}(\sigma \twoheadrightarrow \tau)$ denotes the presence of an effect carrying an argument of type σ and producing a result of type τ.

Typing rules for raising and handling effects

$$\frac{\Gamma \vdash e : \sigma \,!\, \varphi \qquad \varphi = F : \mathtt{Pre}(\sigma \twoheadrightarrow \tau); \varphi'}{\Gamma \vdash \mathtt{perform}\ F\ e : \tau \,!\, \varphi}$$

A given effect *F* can be performed with an argument *e* of any type σ and an expected result of any type τ.
Instead of constraining σ and τ by a prior declaration of *F*, we record them in the effect type φ.

$$\frac{\Gamma \vdash e : \sigma \,!\, \psi \qquad \Gamma \vdash H : \sigma \,!\, \psi \Rightarrow \tau \,!\, \varphi}{\Gamma \vdash \mathtt{handle}\ e\ \mathtt{with}\ H : \tau \,!\, \varphi}$$

The handler *H* transforms the value type and the effect type of computation *e*.

Typing rules for effect handlers

Consider a handler H for effects $F_1, \ldots, F_n$:

$$H = \{\texttt{val}(x) \to M_{val}; F_1(x,k) \to M_1; \ldots; F_n(x,k) \to M_n\}$$

It transforms value and effect types as follows:

$$\frac{\begin{array}{c} \psi = F_1 : \texttt{Pre}(\sigma_1 \twoheadrightarrow \tau_1); \ldots; F_n : \texttt{Pre}(\sigma_n \twoheadrightarrow \tau_n); \omega \\ \varphi = F_1 : \pi_1; \ldots; F_n : \pi_n; \omega \\ \Gamma, x : \sigma \vdash M_{val} : \tau \mathrel{!} \varphi \\ \Gamma, x : \sigma_i, k : \tau_i \xrightarrow{\varphi} \tau \vdash M_i : \tau \mathrel{!} \varphi \text{ for } i = 1, \ldots, n \end{array}}{\Gamma \vdash H : \sigma \mathrel{!} \psi \Rightarrow \tau \mathrel{!} \varphi}$$

$F_1, \ldots, F_n$ must be present in the initial effect type ψ
but can be absent in the transformed effect type φ.
The other effects are described by the row ω, which is preserved.

Strengths and weaknesses of row polymorphism

- + A good match for generic higher-order functions such as `List.map`.
- + Lends itself to ML-style type inference (by unification and generalization; Damas & Milner 1982).
- – Does not account for the "direction of propagation" of effects, which can lead to imprecise types. f
- – Types are hard to read (in typing error messages) and even harder to write (in module interfaces).

Example of an imprecise typing

λf: int $\xrightarrow{\varphi}$ int. λb: bool.

```
    handle (if b then f 0 else perform E ())
    with { val(x) -> x;
           E(_, _) -> f 1 }
```

By “leakage” from `perform E`, the row φ must mention *E* as being present.

Then, the second call `f 1` is seen as possibly raising *E*.

How to present rows to users?

Heuristics for printing: omit ρ, θ variables that do not change the meaning of the type.

Heuristics for writing declarations: by default, all arrows share the same row. For example the declaration

```
val f : ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list
```

is parsed as

$$\mathtt{f} : \forall \alpha\, \beta\, \rho, (\alpha \xrightarrow{\rho} \mathtt{bool}) \xrightarrow{\rho} (\alpha \xrightarrow{\rho} \beta) \xrightarrow{\rho} \alpha\, \mathtt{list} \xrightarrow{\rho} \beta\, \mathtt{list}$$

Simplifying function effects

(D. Leijen, *Type Directed Compilation of Row-Typed Algebraic Effects*, 2017.)

Instantiation and generalization rules make it possible to close a row terminated by a variable ρ.

$$\frac{\Gamma \vdash e : \forall \vec{\alpha}\, \rho, \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\rho} \tau \mathrel{!} \varphi \qquad \rho \notin FV(\sigma, \tau, \pi_1, \ldots, \pi_n)}{\Gamma \vdash e : \forall \vec{\alpha}, \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\emptyset} \tau \mathrel{!} \varphi}$$

We can add a typing rule enabling us to re-open a closed row, thus recovering the flexibility of the $\forall \rho$.

$$\frac{\Gamma \vdash e : \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\emptyset} \tau \mathrel{!} \varphi}{\Gamma \vdash e : \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\varphi'} \tau \mathrel{!} \varphi}$$

Summary

Typing values in the presence of effects

Classic type systems (Hindley-Milner, system *F*, etc.)
easily extend to new control structures
(exceptions, algebraic effects, `call/cc`, delimited continuations)
provided that

- we restrict the generalization of type variables to expressions that are values (the “value restriction” and its variants);
- we declare the types of exceptions, effects, and “prompts” before they are used.

There exists type systems that lift these restrictions, but they are as complex as type and effect systems.

Reflecting effects in types

A wide spectrum of mechanisms ranging from checked exceptions (as in Java) to type and effect systems with row polymorphism.

Issues with practical usability: complicated types, hard to read, hard to write in interfaces.

Issues with software engineering: a risk of “over-constraining” the implementations, restricting their future evolution.

A new perspective based on capabilities that could result in simpler types, but requires language support for “second-class values”.

An alternative: static analysis of control flows, without any type declarations. But this requires the full program to be available (and a lot of RAM).

References

References

Type and effect systems + row types:

- B. C. Pierce, ed: *Advanced Topics in Types and Programming Languages*, MIT Press, 2005. Sections 3.1–3.3 (effects), 10.8 (rows).

A static analysis perspective on type and effect systems:

- F. Nielson, H. R. Nielson, C. Hankin: *Principles of Program Analysis*, chap. 5, *Type and Effect Systems*. Springer, 2005.

The capability-based approach to exceptions in Scala 3:

- M. Odersky, A. Boruch-Gruszecki, J. Brachthäuser, E. Lee, O. Lhoták: *Safer exceptions for Scala*, proceedings Scala 2021. https://doi.org/10.1145/3486610.3486893