Remarrying Effects and Monads

Niki Vazou
UC San Diego

Daan Leijen
Microsoft Research

Abstract

Sixteen years ago Wadler and Thiemann published "The marriage of effects and monads" [35] where they united two previously distinct lines of research: the *effect typing* discipline (proposed by Gifford and others [11, 31]) and *monads* (proposed by Moggi and others [23, 34]). In this paper, we marry effects and monads again but this time within a single programming paradigm: we use monads to *define* the semantics of effect types, but then use the effect types to *program* with those monads. In particular, we implemented an extension to the effect type system of Koka [19] with *user defined effects*. We use a type-directed translation to automatically lift such effectful programs into monadic programs, inserting bind- and unit operations where appropriate. As such, these effects are not just introducing a new effect type, but enable full monadic abstraction and let us "take control of the semi-colon" in a typed and structured manner. We give examples of various abstractions like ambiguous computations and parsers. All examples have been implemented in the Koka language and we describe various implementation issues and optimization mechanisms.

1. Introduction

Sixteen years ago Wadler and Thiemann published "The marriage of effects and monads" [35] where they united two previously distinct lines of research: the *effect typing* discipline (proposed by Gifford and others [11, 31]) and *monads* (proposed by Moggi and others [23, 34]). In this paper, we marry effects and monads again but this time within a single programming paradigm: we use monads to *define* the semantics of effect types, but then use the effect types to *program* with those monads.

We implemented these ideas as an extension of the effect type system of Koka [19] – a JavaScript-like, strongly typed programming language that automatically infers the type and *effect* of functions. For example, the squaring function:

```
function sqr(x : int) { x*x }
```

gets typed as:

```
sqr : int → total int
```

signifying that *sqr* has no side effect at all and behaves as a total function from integers to integers. However, if we add a *print* statement:

```
function sqr(x : int) { print(x); x*x }
```

the (inferred) type indicates that *sqr* has a console effect:

```
sqr : int → console int
```

There is no need to change the syntax of the original function, nor to promote the expression $x*x$ into the *console* effect as effects are automatically combined and lifted.

Monadic effects. We described before a type inference system for a set of standard effects like divergence, exceptions, heap operations, input/output, etc [19]. Here we extend the effect system with *monadic user defined effects*, where we can define our own effect in terms of any monad. As a concrete example, we define an *amb* effect for ambiguous computations [16]. In particular we would like to have ambiguous operations that return *one* of many potential values, but in the end get a list of *all* possible outcomes of the ambiguous computation. Using our new effect declaration we can define the semantics of the *amb* effect in terms of a concrete list monad:

```
effect amb⟨a⟩ = list⟨a⟩ {
  function unit( x )      { [x] }
  function bind( xs, f ) { xs.concatMap(f) }
}
```

where the *unit* and *bind* correspond to the usual list-monad definitions. Given such effect declaration, our system automatically generates the following two (simplified) primitives:

```
function to_amb    ( xs : list⟨a⟩ )              : amb a
function from_amb ( action : () → amb a ) : list⟨a⟩
```

This is really where the marriage between effects and monads comes into play as it allows us to reify the two representations going from a concrete monad to its corresponding effect and vice versa. Given these primitives, we can now *use* our new *amb* effect to construct truth tables for example:

```
function flip() : amb bool {
  to_amb( [False, True] )
}

function xor() : amb bool {
  val p = flip()
  val q = flip()
  (p || q) && not(p&&q)     // p,q : bool
}

function main() : console () {
  print( from_amb(xor) )
}
```

Note how the result of *flip* is just typed as *bool* (even though *amb* computations internally use a list monad of *all* possible results). Furthermore, unlike languages like Haskell, we do not need to explicitly lift expressions into a monad, or explicitly bind computations using *do* notation. When we evaluate *main* we get a list of *all* possible output values: [*False*, *True*, *True*, *False*]. One can extend such mechanism to, for example, return a histogram of the results, or to general probabilistic results [16, 30].

Translation. It turns out we can use an *automatic type directed translation* that translates a program with user-defined effect types into a corresponding monadic program. Internally, the previous example gets translated into:

```
function flip : list⟨bool⟩ {
  [False, True]
}

function xor() : list⟨bool⟩ {
  bind( flip(), fun(p) {
    bind( flip(), fun(q) {
      unit( (p || q) && not(p&&q) )
})})}

function main { print( xor() ) }
```

Here we see how the *unit* and *bind* functions of the effect declaration are used, where *bind* is inserted whenever a monadic value is returned and passed the current continuation at that point. Moreover, the *to_amb* and *from_amb* both behave like an identity and are removed from the final monadic program.

The capture of the continuation at every *bind* makes monadic effects very expressive. For example, note that the *amb* effect can cause subsequent statements to be executed multiple times, i.e. once for every possible result. This is somewhat dual to the built-in exception effect which can cause subsequent statements to not be executed at all, i.e. when an exception is thrown. As such, this kind of expressiveness effectively let us take "control of the semicolon".

Many useful library abstractions are effect polymorphic, and they work seamlessly for monadic effects as well. In particular, we do not need families of functions for different monadic variants. For example, in Haskell, we cannot use the usual *map* function for monadic functions but need to use the *mapM* function (or a similar variant). In our system, we can freely reuse existing abstractions:

```
function xor() : amb bool {
  val [p,q] = [1,2].map( fun(_) { flip() } )
  (p || q) && not(p&&q)
}
```

Translating such effect polymorphic functions is subtle though and, as we will discuss, requires dictionaries to be passed at runtime.

The marriage of effects and monads. Wadler and Thiemann [35] show that any effectful computation can be transposed to a corresponding monad. If $[\![\tau]\!]$ is the call-by-value type translation of τ and M_u is the corresponding monad of an effect u, they show how an effectful function of type $[\![\tau_1 \to u\ \tau_2]\!]$ corresponds to a pure function with a monadic type $[\![\tau_1]\!] \to M_u\langle[\![\tau_2]\!]\rangle$. In this article, we translate any effectful function of type $[\![\tau_1 \to \langle u|\epsilon\rangle\ \tau_2]\!]$, to the function $[\![\tau_1]\!] \to \epsilon\ M_u\langle\epsilon, [\![\tau_2]\!]\rangle$. This is almost equivalent, except for the ϵ parameter which represents arbitrary built-in effects like divergence or heap operations. If we assume ϵ to be empty, i.e. a pure function like in Wadler and Thiemann's work, then we have an exact match! As we shall see, due to the non-monadic effects as an extra parameter, our monads are effectively indexed- or poly-monads [14] instead of regular monads.

Contributions. In the rest of the paper we treat each the above points in depth and discuss the following contributions in detail:

- Using the correspondence between monads and effects [35], we propose a novel system where you *define* the semantics of an effect in terms of a first-class monadic value, but you *use* the monad using a first-class effect type. We build on the existing Koka type system [19] to incorporate monadic effects with full polymorphic and higher-order effect inference.
- We propose (§ 3) a sound *type directed monadic translation* that transforms a program with effect types into one with corresponding monadic types. This translation builds on our earlier work on monadic programming in ML [30] and automatically lifts and binds computations. Moreover, relying on the monadic laws, and effect types to guarantee purity of the primitives, the translation is robust in the sense that small rewrites or changes to our algorithm will not affect the final semantics of the program.
- The original row-based polymorphic effect type inference system for Koka [19] was created without monadic effects in mind. It turns out that the system can be used *as is* to incorporate monadic effects as well. Instead, the translation is done purely on an intermediate explicitly typed core calculus, $\lambda^{\kappa u}$. This is a great advantage in practice where we can clearly separate the two (complex) phases in the compiler.
- In contrast to programming with monads directly (as in Haskell), programming with monadic effects integrates seamlessly with built-in effects where there is no need for families of functions like *map*, and *mapM*, or other special monadic syntax. Moreover, in contrast to earlier work by Filinksi who showed how to embed monads in ML [9, 10], our approach is strongly typed with no reliance on first-class continuations in the host language.
- In practice, you need to do a careful monadic translation, or otherwise there is the potential for code blowup, or large performance penalties. We present (§ 4) how we optimize effect polymorphic functions and report on various performance metrics using our Koka to JavaScript compiler, which can run programs both in a browser as well as on NodeJS [33].
- A restriction in this paper is that there can be at most one monadic effect present in a type, e.g., we can have many monadic effects in a program but we must use them independently of each other and cannot compose them automatically. In order to combine two monadic effects, one needs to define a new effect that combines them explicitly. This is just like programming with monads in Haskell where one needs to define explicitly how different monads compose. Nevertheless, by building on earlier work [30], we hope to extend our system in the future to automatically insert morphisms between different monadic effects.

2. Overview

Types tell us about the behavior of functions. For example, the ML type $int \to int$ of a function tells us that the function is well defined on inputs of type *int* and returns values of type *int*. But that is only one part of the story, the ML type tells us nothing about all *other* behaviors: i.e. if it accesses the file system perhaps, or throws exceptions, or never returns a result at all.

In contrast, the type of a function in Koka is always of the form $\tau \to \epsilon\ \tau'$ signifying a function that takes an argument of type τ, returns a result of type τ' *and may* have a side effect ϵ. Sometimes we leave out the effect and write

$\tau \rightarrow \tau'$ as a shorthand for the total function without any side effect: $\tau \rightarrow \langle\rangle\ \tau'$. A key observation on Moggi's early work on monads [23] was that *values* and *computations* should be assigned a different type. Here we apply that principle where effect types only occur on function types; and any other type, like *int*, truly designates an evaluated value that cannot have any effect[1].

In contrast to many other effect systems, the effect types are not just labels that are propagated but they truly describe the semantics of each function. As such, it is essential that the basic effects include exceptions (*exn*) and divergence (*div*). The deep connection between the effect types and the semantics leads to strong reasoning principles. For example, Koka's soundness theorem [19] implies that if the final program does not have an *exn* effect, then its execution *never* results in an exception (and similarly for divergence and state).

Example: Exceptions. Exceptions in Koka can be raised using the primitive *error* function:

```
error : string → exn a
```

The type shows that *error* takes a string as an argument and may potentially raise an exception. It returns a value of any type! This is clearly not possible in a strongly typed parametric language like Koka, so we can infer from this type signature that *error always* raises an exception. Of course, effects are properly propagated so the function *wrong* will be inferred to have the *exn* type too:

```
function wrong() : exn int { error("wrong"); 42 }
```

Exceptions can be detected at run-time (unlike divergence) so we can discharge exceptions using the *catch* function:

```
function catch( action : () → exn a,
                handler : exception → a) : a
```

To catch exceptions we provide two arguments: an *action* that may throw an exception and an exception *handler*. If *action*() throws an exception the *handler* is invoked, otherwise the result of the *action*() is returned. In both cases *catch* has a *total* effect: it always returns a value of type *a*. For example, function *pure* always returns an *int*:

```
function pure() : int { catch( wrong, fun(err){ 0 } ) }
```

Effect polymorphism. In reality, the type of *catch* is more polymorphic: instead of just handling actions that can at most raise an exception, it accepts actions with any effect that includes *exn*:

```
function catch( action : () → ⟨exn | e⟩ a,
                handler : exception → e a ) : e a
```

The type variable *e* applies to any effect. The type expression $\langle exn \mid e\rangle$ stands for the effect row that extends the effect *e* with the effect constant *exn*. Effectively, this type captures that given an action that can potentially raise an exception, and perhaps has other effects *e*, *catch* will handle that exception but not influence any of the other effects. In particular, the *handler* has at most effect *e*. For example, the result effect of:

```
catch( wrong, fun(err) { print(err); 0 } )
```

is *console* since the handler uses *print*. Similarly, if the handler itself raises an exception, the result of *catch* will include the *exn* effect:

```
catch( wrong, fun(err) { error("oops") } )
```

Apart from exceptions Koka supplies more built-in effects: we already mentioned *div* that models divergence; there is also *io* to model interaction with input-output, *ndet* to model non-determinism, heap operations through *alloc*, *read*, and *write*, and the list goes on. For all built-in effects, Koka supplies primitive operators that *create* (e.g. *error*, *random*, *print*, etc) and sometimes *discharge* the effect (e.g. *catch*, *timeout*, or *runST*).

The main contribution of this paper is how we extend Koka so that the user can define her own effects, by specifying the type and meaning of new effects and defining primitive operations on them.

2.1. The ambiguous effect

Already in the introduction we saw how we could define and use the ambiguous *amb* effect with *flip* and *xor* operations. We will now discuss the definition and translation in more detail. The *amb* effect was defined using an effect declaration:

```
effect amb⟨a⟩ = list⟨a⟩ {
  function unit( x : a ) : list⟨a⟩ { [x] }
  function bind( xs : list⟨a⟩, f : a → e list⟨b⟩ ) : e list⟨b⟩ {
    xs.concatMap(f)
  }
}
```

As we can see, defining the *amb* effect basically amounts to defining the standard list monad, and is surprisingly easy, especially if we remove the *optional* type annotations. Given the above definition, a new effect type *amb* is introduced, and we know:

1. how to *represent* (internally) ambiguous computations of *a* values: as a $list\langle a\rangle$
2. how to *lift* plain values into ambiguous ones: using *unit*, and
3. how to *combine* ambiguous computations: using *bind*.

Moreover, with the above definition Koka *automatically* generates the *to_amb* and *from_amb* primitives:

```
function to_amb   ( xs : list⟨a⟩ )            : amb a
function from_amb ( action : () → amb a ) : list⟨a⟩
```

that allow us to go from monadic values to effect types and vice versa. These are basically typed versions of the *reify* and *reflect* methods of Filinski's monadic embedding [9]. Later we discuss the more interesting effect of parsers (§ 2.3), but before that, let's discuss in more detail how we do a monadic translation of effects.

2.2. Translating effects

Koka uses a *type directed* translation to internally translate effectful to monadic code. As shown in the introduction, the *xor* function is translated as:

```
function xor() : amb bool        function xor() : list⟨bool⟩
{                                {
  val p = flip()                   bind( flip(), fun(p) {
  val q = flip()          ⇝          bind( flip(), fun(q) {
                                       unit(
  (p || q) && not(p&&q)                  (p || q) && not(p&&q) )
}                                  })})}
```

[1] In contrast to Haskell for example, where *Int* really stands for $Int_\perp$, i.e. referring to a value of such type may still diverge or raise an exception.

In particular, *bind* is inserted at every point where a monadic value is returned, and passed the current continuation at that point. Since *flip* has an ambiguous result, our type-directed translation binds its result to a function that takes p as an argument and similarly for q. Finally, the last line returns a pure boolean value, but *xor*'s result type is ambiguous. We use *unit* to lift the pure value to the ambiguous monad. We note that in Koka's actual translation, *xor* is translated more efficiently using a single *map* instead of a *unit* and *bind*.

The translation to monadic code is quite subtle and relies crucially on type information provided by type inference. In particular, the intermediate core language is explicitly type à la System F (§ 3.1). This way, we compute effects precisely and determine where *bind* and *unit* get inserted (§ 3.3). Moreover, we rely on the user to ensure that the *unit* and *bind* operations satisfy the monad laws [34], i.e. that *unit* is a left- and right identity for *bind*, and that *bind* is associative. This is usually the case though; in particular because the effect typing discipline ensures that both *unit* and *bind* are *total* and cannot have any side-effect (which makes the translation semantically robust against rewrites).

2.2.1. Translating polymorphic effects

One of the crucial features of Koka is effect polymorphism. Consider the function *map*

```
function map(xs : list⟨a⟩, f : (a) → e b) : e list⟨b⟩ {
  match(xs) {
    Nil → Nil
    Cons(y, ys) → Cons( f(y), map(ys,f) )
  }
}
```

The function *map* takes as input a function f with some effect e. Since it calls f, *map* can itself produce the effect e, *for any* effect e. This means we can use such existing abstractions on user defined effects too:

```
function xor() {
  val [p,q] = map( [1,2], fun(_) { flip() } )
  (p || q) && not(p&&q)
}
```

Unfortunately, this leads to trouble when doing a type directed translation: since the function passed to *map* has a monadic effect, we need to *bind* the call $f(y)$ *inside* the *map* function! Moreover, since we can apply *map* to *any* monadic effect, we need to be able to dynamically call the right *bind* function.

The remedy is to pass Haskell-like *dictionaries* or monad interfaces to effect polymorphic functions. In our case, a dictionary is a structure that wraps the monadic operators *bind* and *unit*. The dictionaries are transparent to the user and are automatically generated and inserted. During the translation, every effect polymorphic function takes a dictionary as an additional first argument. Figure 1 shows how the *map* function gets translated.

Now that internally every effect polymorphic function gets an extra dictionary argument, we need to ensure the corresponding dictionary is supplied at every call-site. Once again, dictionary instantiation is type-directed and builds upon Koka's explicitly typed intermediate core language. Whenever a polymorphic effect function is instantiated with a specific effect, the type directed translation automatically inserts the corresponding dictionary argument. Figure 1 shows this in action when we call *map* inside the *xor* function. We can still use *map* with code that has a

```
// source effectful code
function map(xs, f) {
  match(xs) {
    Nil → Nil
    Cons(y, ys) →
      val z  = f(y)
      val zs = map(ys, f)
      Cons(z, zs)

  }
}

function xor() {
  val [p,q] =
    map( [1,2],
      fun(_) { flip() })


  (p || q) && not(p&&q)

}
```

```
// translated monadic code
function map(d : dict⟨e⟩, xs, f) {
  match(xs) {
    Nil → d.unit(Nil)
    Cons(y, ys) →
        d.bind( f(y), fun(z) {
          d.bind( map(ys, f),
            fun(zs) {
              d.unit( Cons(z, zs)
  })})}
}

function xor() {
  dict_amb.bind(
    map( dict_amb, [1,2],
        fun(_) { flip() }),
  fun([p,q]) {
    dict_amb.unit(
      (p || q) && not(p&&q))
  })
}
```

Figure 1. Dictionary translation of *map* and *xor*

non-monadic effect and in that case the translation will use the the dictionary of the primitive identity monad, e.g. *map*(*dict_id*, [1,2], *sqr*).

A naive translation is not very efficient though: always using the monadic version of *map* introduces a performance penalty to all code, even code that doesn't use any monadic effect. As shown in § 4.1, we avoid this by careful translation. For every effect polymorphic function, we generate two versions: one that takes a monad dictionary, and another that has no monadic translation at all. When instantiating *map* we use the efficient non-monadic version unless there is monadic effect. This way the performance of code with non-monadic effects is unchanged.

Being able to reuse any previous abstractions when using monadic effects is very powerful. If we insert user-defined effects to a function, only the type of the function changes. Contrast this to Haskell: when inserting a monad, we need to do a non-trivial conversion of the syntax to *do* notation, but also we need to define and use monadic counterparts of standard functions, like *mapM* for *map*.

2.2.2. Interaction with other effects

User defined effects can be combined with other effects. However, in this paper we do not allow multiple user-defined effects to be combined and in our implementation the type-checker enforces this restriction via various checks. Combining multiple monadic effects is for example described by Swamy et al. [30], and generally requires morphisms between different monads, which we leave as a future work.

For now we just consider how user-defined effects, like *amb*, interact with built-in effects like state, divergence, and exceptions. The formal semantics of Koka [19] are unchanged in our system, and we define the semantics of the user-defined effects simply as a monadic transformation. As such, if we viewed the effects as a stack of monad transformers, the user defined effects would be last with all built-in effects transforming it, i.e. something like $div\langle st\langle exn\langle amb\langle a\rangle\rangle\rangle\rangle$. These semantics still require careful compilation; for example, it is important when doing the internal monadic translation to properly capture local variables in the continuation functions passed to *bind*.

Here is an example of a more subtle interaction: if we use mutable variables in the ambiguity monad, we may observe that computations run multiple times:

```
function strange() : amb bool {
  var i := 0
  val p = flip()
  val q = flip()
  i := i+1
  if (i ≥ 4) then True else (p || q) && not(p&&q)
}
```

In this example, we define and increment the mutable variable i. The function *strange* itself does not have a stateful effect ($st\langle h\rangle$) because the mutability is not observable from outside and can be discharged automatically through Koka's higher-ranked type system [17, 19]. However, executing *run*(*strange*) results in [*False*, *True*, *True*, *True*] where inside the body of *strange* we can observe that some statements are executed multiple times. This shows the importance of strong typing: in an IDE one would see that the *flip*() invocations have an *amb* effect that causes the following statements to potentially execute more than once. This is similar for exceptions, where statements following invocations of functions that may raise exceptions, may not execute at all.

Under the monadic semantics, the interaction with built-in effects is more or less what one would expect, with one exception: the exception effect does not play nice with certain user defined effects due to the (expected) lexical scoping of *catch*. Exceptions interact with *amb* as expected, but this is not the case in the context of the parser effect and we discuss this further in the next section.

2.3. The parser effect

We conclude the overview with a more advanced example in the form of monadic parsers. A parser can be defined as a function that consumes the input string and returns a list of (all possible) pairs of parsed tokens and the remaining input: $: string \rightarrow list\langle(a,string)\rangle$. This representation is quite standard but many other designs are possible [15, 20]. Since a parser is a function, it may have effects itself: parsers can *diverge* or *throw exceptions* for example. This means that we need to parameterize the parser effect with two type parameters (instead of one):

```
effect parser⟨e,a⟩ = string → e list⟨(a,string)⟩ {
  function unit( x ) { return fun(s) { [(x,s)] } }
  function bind( p, f ) {
    return fun(s) {
      p(s).concatMap( fun(r) { f(r.fst)(r.snd) } )
  }}
}
```

Given the above definition, Koka automatically derives the conversion functions:

```
function to_parser(   p : string → e list⟨(a,string)⟩ )
                      : ⟨parser | e⟩ a
function from_parser( action : () → ⟨parser | e⟩ a)
                      : e (string → e list⟨(a,string)⟩)
```

which can be used by the parser-library developer to build primitive parsing operators as shown in Figure 2: *parse* that takes a parsing computation and an input string and runs the parser; *succeed*(x) that returns its argument x, without consuming the input; *satisfy*(p) that parses the string *iff* it satisfies p; and *choice*(p_1, p_2) that chooses between two parsers p_1 or p_2.

```
function parse( p : () → ⟨parser | e⟩ a, input : string
              ) : e list⟨(a,string)⟩
{
  from_parser(p)(input)
}

function succeed( x : a ) : parser a {
  to_parser fun(input) { [(x,input)] }
}

function satisfy( pred : (string) → maybe⟨(a,string)⟩ ) {
  to_parser fun(input) {
    match(pred(input)) {
      Just((x,rest)) → [(x,rest)]
      Nothing        → []
  }}
}

function choice( p1 : () → ⟨parser | e⟩ a,
                 p2 : () → ⟨parser | e⟩ a ) : ⟨parser | e⟩ a {
  to_parser fun(input) {
    match (parse(p1,input)) {
      Nil → parse(p2,input)
      res → res
  }}
}
```

Figure 2. Parser primitives

Note how the effect e in *from_parser* occurs both as the effect of the function, but also in the returned parser function. Essentially this is because we cannot distinguish at the type level whether an effect occurs when constructing the parser (i.e. before the first *bind*), or whether it occurs when running the parser.

Having set up the parser effect and its primitives, we can easily construct other parsers. For example, *many*(p) is a parser that applies p zero or more times. A *digit* can be parsed as a character where *satisfy isDigit*. Combining these two, *many*(*digit*) gives a list of parsed digits.

```
function main(input: string): div list⟨int, string⟩ {
  parse(integer, input)
}

function integer() : ⟨parser,div⟩ int {
  val ds = many(digit)
  ds.foldl(0,fun(i,d) { i*10 + d })
}

function digit() : parser int { satisfy( ... ) }

function many( p : () → ⟨parser,div | e⟩ a )
             : ⟨parser,div | e⟩ list⟨a⟩ {
  choice { Cons(p(),many(p)) } { succeed(Nil) }
}
```

Running *main*("12a") now results in [(12,"a")]. Note also how in the *integer* function we can very easily combine parser results (*many*(*digit*)) with pure library functions (*foldl*).

2.3.1. Interaction with exceptions

Because the parser monad is defined as function we need to be careful on how exception handling is defined. Take for example the following parser that may raise an exception:

```
function division() : ⟨parser,exn⟩ int {
  val i = integer(); keyword("/"); val j = integer()
  if (j==0) then error("divide by zero") else i/j
}
```

Suppose now that we catch errors on parsers, as in the following *safe* version of our parser:

```
function safe() : parser int { catch( division, fun(err) { 0 } ) }
```

If *catch* is implemented naively this would not work as expected. In particular, if *catch* just wraps a native `try`-*catch* block, then the exceptions raised inside *division* are not caught: after the monadic translation, *division* would return immediately with a parser function: only invoking that function would actually raise the exception (i.e. when the parser is run using *parse*). Effectively, the lexical scoping expectation of the *catch* would be broken.

Our (primitive) *catch* implementation takes particular care to work across monadic effects too. Since *catch* is polymorphic in the effect, the type directed translation will actually call the specific monadic version of *catch* and pass a dictionary as a first argument. The primitive monadic *catch* is basically implemented in pseudo-code as:

```
function catch_monadic( d : dict⟨e⟩, action, handler ) {
  catch( { d.bind_catch( action, handler ) }, handler)
}
```

Besides catching regular exceptions raised when executing *action*(), it uses the special *bind_catch* method on the dictionary that allows any user-defined effect to participate in exception handling. This is essential for most effects that are implemented as functions. For our parser, we can implement it as:

```
effect parser⟨e,a⟩ = string → e list⟨(a,string)⟩ {
  ...
  function bind_catch( p, handler ) {
    fun(s) { catch( { p(s) }, fun(err) { handler(err)(s) }) }
  }
}
```

With this implementation in place, the parser effect participates fully in exception handling and the *safe* parser works as expected, where any exception raised in *division* is handled by our handler, i.e. the expression *parse*(*safe*,"1/0") evaluates to 0.

Here is the type of *bind_catch* and its default implementation:

```
function bind_catch( action : () → ⟨exn | e⟩ m⟨⟨exn | e⟩,a⟩,
                     handler : exception → e m⟨e,a⟩
                   ) : e m⟨e,a⟩
{ catch(action,handler) }
```

In the above type we write m for the particular monadic type on which the effect is defined. A nice property of this type signature and default implementation is that Koka type inferencer requires you to define *bind_catch*, *only when* needed for your particular monad. For example, the default works as is for the *amb* effect since its monad disregards the e parameter, but the default is correctly rejected by the type checker for the *parser* since the signature of *catch* requires the $m\langle\langle exn \mid e\rangle,a\rangle$ to be unified with $m\langle e,a\rangle$.

expressions	e	$::=$	$x^\sigma \mid c^\sigma \mid e\ e$	
		$\mid$	$\lambda^\epsilon x : \sigma.\, e$	
		$\mid$	$\mathtt{val}\ x^\sigma = e; e$	
		$\mid$	$\mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e$	
		$\mid$	$e\ [\sigma] \mid \Lambda\alpha^\kappa .\, e$	
types	τ^κ	$::=$	α^κ	type variable
		$\mid$	$c^{\kappa_0}\langle\tau_1^{\kappa_1}, ..., \tau_n^{\kappa_n}\rangle$	$\kappa_0 = (\kappa_1, ..., \kappa_n) \to \kappa$
kinds	κ	$::=$	$* \mid \mathsf{e}$	values, effects
		$\mid$	k	effect constants
		$\mid$	u	user effects
		$\mid$	$(\kappa_1, ..., \kappa_n) \to \kappa$	type constructor
type scheme	σ	$::=$	$\forall\alpha^\kappa.\, \sigma \mid \tau^*$	

const	$()$	$::$	$*$	unit type
	$bool$	$::$	$*$	bool type
	$(\_ \to \_\ \_)$	$::$	$(*, \mathsf{e}, *) \to *$	functions
	$\langle\rangle$	$::$	e	empty effect
	$\langle\_ \mid \_\rangle$	$::$	$(\mathsf{k}, \mathsf{e}) \to \mathsf{e}$	effect extension
	exn, div	$::$	k	partial, divergent
	$\mathsf{user}\langle\_\rangle$	$::$	$\mathsf{u} \to \mathsf{k}$	user effects
	$\mathsf{tdict}\langle\_\rangle$	$::$	$\mathsf{e} \to *$	effect to universe

Syntactic sugar:

effects	ϵ	$\doteq$	τ^{e}
effect variables	μ	$\doteq$	α^{e}
closed effects	$\langle l_1, ..., l_n\rangle$	$\doteq$	$\langle l_1, ..., l_n \mid \langle\rangle\rangle$
user effects	l^{u}	$\doteq$	$\mathsf{user}\langle l^{\mathsf{u}}\rangle$

Figure 3. Syntax of explicitly typed Koka, $\lambda^{\kappa\mathsf{u}}$.

3. Formalism

In this section we formalize the type-directed translation using an explicitly typed effect calculus we call $\lambda^{\kappa\mathsf{u}}$. First, we present the syntax (§ 3.1) and typing (§ 3.2) rules for $\lambda^{\kappa\mathsf{u}}$. Then, we formalize our translation (§ 3.3) from effectful to monadic $\lambda^{\kappa\mathsf{u}}$. Finally, we prove soundness (§ 3.4) by proving type preservation of the translation.

3.1. Syntax

Figure 3 defines the syntax of expressions and types of $\lambda^{\kappa\mathsf{u}}$, a polymorphic explicitly typed λ-calculus. It is very similar to System F [12, 27] except for the addition of effect types.

Expressions. $\lambda^{\kappa\mathsf{u}}$ expressions include typed variables x^σ, typed constants c^σ, λ-abstraction $\lambda^\epsilon x : \sigma.\, e$, application $e\ e$, value bindings $\mathtt{val}\ x^\sigma = e; e$, if combinators $\mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e$, type application $e\ [\sigma]$ and type abstraction $\Lambda\alpha^\kappa .\, e$. Note that each value variable is annotated with its type and each type variable is annotated with its kind. Finally, each λ-abstraction $\lambda^\epsilon x : \sigma.\, e$ is annotated with its result effect ϵ which is necessary to check effect types.

As an aside, often in strict languages one distinguishes between values and expressions, where type abstraction and application can only range over value expressions. In our setting, we do not have to do this since we rely on the Koka type checker to guarantee that we only generalize over expressions with a total effect (and there is no need for the 'value restriction' [19]).

Types and type schemes. Types consist of explicitly kinded type variables α^κ and application of constant type constructors $c^{\kappa_0}\langle\tau_1^{\kappa_1}, ..., \tau_n^{\kappa_n}\rangle$, where the type constructor c has the appropriate kind $\kappa_0 = (\kappa_1, ..., \kappa_n) \to \kappa$. We do

$$(\text{T-CON}) \; \frac{}{\vdash c^{\sigma} \; : \; \sigma} \qquad (\text{T-VAR}) \; \frac{}{\vdash x^{\sigma} \; : \; \sigma}$$

$$(\text{T-TLAM}) \; \frac{\vdash e \; : \; \sigma}{\vdash \Lambda\alpha^{\kappa}.\, e \; : \; \forall\alpha^{\kappa}.\, \sigma} \qquad (\text{T-TAPP}) \; \frac{\vdash e \; : \; \forall\alpha.\, \sigma}{\vdash e[\tau] \; : \; \sigma[\alpha \mapsto \tau]}$$

$$(\text{T-LAM}) \; \frac{\vdash e \; : \; \tau_2}{\vdash \lambda^{\epsilon} x : \tau_1.\, e \; : \; \tau_1 \to \epsilon \; \tau_2}$$

$$(\text{TAPP}) \; \frac{\vdash e_1 \; : \; \tau_1 \to \epsilon \; \tau_2 \quad \vdash e_2 \; : \; \tau_1}{\vdash e_1 \; e_2 \; : \; \tau_2}$$

$$(\text{T-VAL}) \; \frac{\vdash e_1 \; : \; \sigma_1 \quad \vdash e_2 \; : \; \sigma_2}{\vdash \mathsf{val}\; x \; = \; e_1; \; e_2 \; : \; \sigma_2}$$

$$(\text{T-IF}) \; \frac{\vdash e \; : \; \mathsf{bool} \quad \vdash e_1 \; : \; \sigma \quad \vdash e_2 \; : \; \sigma}{\vdash \mathsf{if}\; e \;\mathsf{then}\; e_1 \;\mathsf{else}\; e_2 \; : \; \sigma}$$

Figure 4. Type rules for explicitly typed Koka.

not provide special syntax for function types, as they can be modeled by the constructor $(\_ \to \_\_\_) \; :: \; (*, \mathsf{e}, *) \to *$ that, unlike the usual function type, explicitly reasons for the effect produced by the function. Finally, types can be qualified over type variables to yield type schemata.

Kinds. Well-formedness of types is guaranteed by a simple kind system. We annotate the type τ with its kind κ, as τ^{κ}. Apart from the usual kinds $*$ and $\to$, we have kinds for effect rows (e), effect constants (k), and user-defined effects (u). We omit the kind κ of the type τ^{κ}, and write τ, when κ is immediately apparent or not relevant. For clarity, we use α for regular type variables and μ for effect type variables. Finally, we write ϵ for effects, i.e. types of kind e.

Effects. Effects *are* types. Effect types are defined as a row of effect labels l. Such effect row is either empty $\langle\rangle$, a polymorphic effect variable μ, or an extension of an effect row ϵ with an effect constant l, written as $\langle l|\epsilon\rangle$. Effect constants are either built-in Koka effects, i.e. anything that is interesting to our language like exceptions (*exn*), divergence (*div*) etc. or lifted user-defined monadic effects like the ambiguous effect $\mathsf{amb}^{\mathsf{u}} :: \mathsf{u}$. Note that for an effect row to be well-formed we use the user effect function to lift $\mathsf{user}\langle\mathsf{amb}^{\mathsf{u}}\rangle \; :: \; \mathsf{k}$ to the appropriate kind k. For simplicity, in the rest of this section we omit the explicit lifting and write $\mathsf{amb}^{\mathsf{u}}$ to denote $\mathsf{user}\langle\mathsf{amb}^{\mathsf{u}}\rangle$ when a label of kind k is expected.

Finally, Figure 3 includes definition of type constants and syntactic sugar required to simplify the rest of this section.

Type rules. Figure 4 describes type rules for $\lambda^{\kappa\mathsf{u}}$ where the judgment $\vdash e \; : \; \sigma$ assigns type σ for an expression e. All the rules are equivalent to the System F rules, except for rule (LAM) where the effect of the function in the type is drawn from the effect annotation in the λ-abstraction. Just like System F, there is the implicit assumption that under a lambda $\lambda^{\epsilon} x : \sigma.\, e$ (1) all (bound) occurrences of x are typed as x^{σ}, and (2) in all applications (APP) the effect is ϵ, i.e. $e_1 \; : \; \sigma_1 \to \epsilon \; \sigma$. By construction, the Koka type inference rules always produce well-formed $\lambda^{\kappa\mathsf{u}}$. Soundness of $\lambda^{\kappa\mathsf{u}}$ follows from soundness of Koka as described in [19].

3.2. Type inference for effect declarations

The effect- and type- inference of Koka is presented in previous work [18, 19]. Here we look specifically at how type inference works for effect declarations.

The Identity Effect. Before we look at the general type inference rule for effect declarations (Figure 5) we start with a concrete example, namely the identity effect *uid*:

```
effect uid⟨e,a⟩ = a {
  function unit(x) { x }
  function bind(x,f) { f(x) }
}
```

From the above effect definition, initially, Koka generates a type alias that isolates the first line of the definition and relates the effect name with its monadic representation.

$\mathsf{alias}\; M_{uid}\langle\epsilon, \alpha\rangle \; = \; \alpha$

Then, Koka checks well-formedness of the effect definition, by (type-) checking that *unit* and *bind* are the appropriate monadic operators. Concretely, it checks that

$$\begin{aligned} &unit \; : \; \forall\alpha\mu.\, \alpha \to \mu \; M_{uid}\langle\mu, \alpha\rangle \\ &bind \; : \; \forall\alpha\beta\mu.\, (M_{uid}\langle\mu, \alpha\rangle, \; \alpha \to \mu \; M_{uid}\langle\mu, \beta\rangle) \to \mu \; M_{uid}\langle\mu, \beta\rangle \end{aligned}$$

Given the definitions of *unit* and *bind*, Koka *automatically* constructs the primitives required by the rest of the program to safely manipulate the identity effect:

- uid^{u} – the effect constant that can be used inside types,
- $to_{uid} \; : \; \forall\alpha\mu.\, (M_{uid}\langle\mu, \alpha\rangle) \to \langle uid|\mu\rangle \; \alpha$ – the function that converts monadic computations to effectful ones,
- $from_{uid} \; : \; \forall\alpha\beta\mu.\, (() \to \langle uid|\mu\rangle \; \alpha) \to \mu \; M_{uid}\langle\mu, \alpha\rangle$ – the dual function that converts effectful function to their monadic equivalent, and finally,
- $dict_{uid}$ – the (internal) effect dictionary that stores *uid*'s monadic operators.

Dictionaries. The first three values are user-visible but the final dictionary value is only used internally during the monadic translation. The type of the effect dictionary (e.g. $dict_{uid}$), is a structure that contains the monadic operators *unit* and *bind* of the effect. It can as well include the monadic *map* which will otherwise be automatically derived from *unit* and *bind*, and the $bind_{catch}$ method to interact with primitive exceptions. Thus, we define the dictionary structure as a type that is polymorphic on the particular monad, represented as type variable $m \; :: \; (\mathsf{e}, *) \to *$:

$$\begin{aligned} &\mathsf{struct}\; tdict\langle m\rangle \; \{ \\ &\quad unit \; : \; \forall\alpha\mu.\, \alpha \to \mu \; m\langle\mu, \alpha\rangle \\ &\quad map \; : \; \forall\alpha\beta\mu.\, (m\langle\mu, \alpha\rangle, \; \alpha \to \beta) \to \mu \; m\langle\mu, \beta\rangle \\ &\quad bind \; : \; \forall\alpha\beta\mu.\, (m\langle\mu, \alpha\rangle, \; \alpha \to \mu \; m\langle\mu, \beta\rangle) \to \mu \; m\langle\mu, \beta\rangle \\ &\quad bind_{catch} : \; \forall\alpha\mu.\, (m\langle\langle exn|\mu\rangle, \alpha\rangle, \; exc \to \mu \; m\langle\mu, \alpha\rangle) \to \mu \; m\langle\mu, \alpha\rangle \\ &\} \end{aligned}$$

With this we can type $dict_{uid} \; : \; tdict\langle M_{uid}\rangle$.

General user-defined effects. Figure 5 generalizes the previous concrete example to any user-defined effect declaration. The judgment:

$$\Gamma \vdash \mathsf{effect}\; eff\langle\mu, \alpha\rangle \; = \; \tau\langle\mu, \alpha\rangle \; \{ \; unit \; = \; e_1; \; bind \; = \; e_2 \; \} \; : \; \Gamma'$$

states that under a kind- and type- environment Γ, the effect declaration *eff* results in a new type environment Γ' that is extended with the needed types and primitives implied by *eff*. As shown in Figure 5, we first check well-formedness of the effect types, and then check that *unit* and *bind* operations have the proper types. Finally, the environment is extended with the corresponding types and values.

3.3. Type-directed monadic translation

Next we define the type-directed monadic translation $e \rightsquigarrow_{\epsilon} e' \mid v$ that takes an effect expression e to the monadic expression e'.

$$
\text{(EFF)}\quad \frac{\begin{array}{c}\Gamma, \mu^{\mathsf{e}}, \alpha^{*} \vdash_{\mathsf{k}} \tau :: \ * \quad \Gamma' \ = \ \Gamma,\ M_{\mathit{eff}}\langle \mu, \alpha\rangle = \tau \\ \Gamma' \vdash e_1 \ :\ \forall \alpha\mu.\ \alpha \rightarrow \mu\ M_{\mathit{eff}}\langle \mu, \alpha\rangle \\ \Gamma' \vdash e_2 \ :\ \forall \mu\alpha\beta.\ (M_{\mathit{eff}}\langle \mu, \alpha\rangle,\ \alpha \rightarrow \mu\ M_{\mathit{eff}}\langle \mu, \beta\rangle) \rightarrow \mu\ M_{\mathit{eff}}\langle \mu, \beta\rangle\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{effect}\ \mathit{eff}\langle \mu, \alpha\rangle \ = \ \tau\ \{\ \mathit{unit} \ = \ e_1;\ \mathit{bind} \ = \ e_2\ \}\ : \\ \quad \Gamma',\ \mathit{eff}^{\mathsf{u}},\ \mathit{edict}_{\mathit{eff}}\ :\ \mathit{tdict}\langle M_{\mathit{eff}}\rangle \\ \quad \mathit{to}_{\mathit{eff}}\ :\ \forall \alpha\mu.\ (M_{\mathit{eff}}\langle \mu, \alpha\rangle) \rightarrow \langle \mathit{eff}|\mu\rangle\ \alpha, \\ \quad \mathit{from}_{\mathit{eff}}\ :\ \forall \alpha\beta\mu.\ (() \rightarrow \langle \mathit{eff}|\mu\rangle\ \alpha) \rightarrow \mu\ M_{\mathit{eff}}\langle \mu, \alpha\rangle\end{array}}
$$

Figure 5. Type rule for effect declarations.

$$
\begin{array}{lcl}
\mathsf{bind}^{\upsilon}_{\langle\rangle}(\epsilon, e_1, x, e_2) & = & \mathsf{val}\ x \ = \ e_1;\ e_2 \\
\mathsf{bind}^{\langle\rangle}_{\upsilon}(\epsilon, e_1, x, e_2) & = & \mathit{dict}_{\upsilon}.\mathit{map}\ \langle \tau_1, \tau_2, \epsilon\rangle(e_1,\ \lambda^{\langle\rangle}\ x\ :\ \tau_1.\ e_2) \\
 & & \text{with} \vdash e_1 : \mathsf{mon}\langle \upsilon, \epsilon, \tau_1\rangle, \vdash e_2 : \tau_2 \\
\mathsf{bind}^{\upsilon}_{\upsilon}(\epsilon, e_1, x, e_2) & = & \mathit{dict}_{\upsilon}.\mathit{bind}\langle \tau_1, \tau_2, \epsilon\rangle(e_1,\ \lambda^{\tilde{\epsilon}}\ x\ :\ \tau_1.\ e_2) \\
 & & \text{with} \vdash e_1 : \mathsf{mon}\langle \upsilon, \epsilon, \tau_1\rangle, \vdash e_2 : \mathsf{mon}\langle \upsilon, \epsilon, \tau_2\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{lift}^{\upsilon}_{\upsilon}(\epsilon, e) & = & e \\
\mathsf{lift}^{\upsilon}_{\langle\rangle}(\epsilon, e) & = & \mathit{dict}_{\upsilon}.\mathit{unit}\langle \tau,\ \epsilon\rangle(e) \quad \text{where } \upsilon \neq \langle\rangle, \vdash e\ :\ \tau
\end{array}
$$

Figure 6. Helper functions for binding and lifting.

Computed effects. Our translation needs two effects ϵ and υ: the *maximum* (inferred) effect ϵ and the *minimum* (computed) effect υ. After type inference, every function body has one unified effect ϵ, that consists of the unification of all the effects in that function. Our translation computes bottom-up the minimal user-defined portion of each separate sub-expression, where υ should always be contained in ϵ. Specifically, we define *computed effects* υ as effect types ϵ that have the following grammar:

$$\upsilon ::= \ \langle l^{\mathsf{u}}\rangle \mid \langle\rangle \mid \mu$$

Note that in this work we do not allow more that one user-defined effects to co-exist, but we defer this to future work (§ 6). We can convert a regular effect type ϵ to a computed effect $\overline{\epsilon}$, and dually, we apply $\tilde{\epsilon}$ to an effect ϵ to remove the user defined effects:

$$
\begin{array}{lclclcl}
\overline{\langle l^{\mathsf{u}} \mid \epsilon\rangle} & = & \langle l^{\mathsf{u}}\rangle \ \text{if}\ \overline{\epsilon} = \langle\rangle & \qquad & \widetilde{\langle l^{\mathsf{u}} \mid \epsilon\rangle} & = & \epsilon \\
\overline{\langle l^{\kappa} \mid \epsilon\rangle} & = & \overline{\epsilon} \quad \text{if}\ \kappa \neq \mathsf{u} & & \widetilde{\langle l^{\kappa} \mid \epsilon\rangle} & = & \tilde{\epsilon} \\
\overline{\langle\rangle} & = & \langle\rangle & & \widetilde{\langle\rangle} & = & \langle\rangle \\
\overline{\mu} & = & \mu & & \tilde{\mu} & = & \mu
\end{array}
$$

Constraining the minimum computed effects greatly simplifies effect operations. We can add computed effects as:

$$
\begin{array}{lcl}
\langle\rangle \oplus \langle\rangle & = & \langle\rangle \\
\langle\rangle \oplus \upsilon & = & \upsilon \\
\upsilon \oplus \langle\rangle & = & \upsilon \\
\upsilon \oplus \upsilon & = & \upsilon
\end{array}
$$

Type translation. Our translation transforms effectful- to monadic expressions and changes the return types in the process. The function

$$
\begin{array}{lcll}
[\![\alpha^{\kappa}]\!] & = & \alpha^{\kappa} & \\
[\![\tau \rightarrow \epsilon\ \tau']\!] & = & [\![\tau]\!] \rightarrow \tilde{\epsilon}\ \mathsf{mon}\langle \overline{\epsilon}, \epsilon, [\![\tau']\!]\rangle & \\
[\![c^{\kappa}\langle \tau_1,\ \ldots,\ \tau_n\rangle]\!] & = & c^{\kappa}\langle [\![\tau_1]\!],\ \ldots,\ [\![\tau_n]\!]\rangle & \mathit{with}\ c \neq \rightarrow \\
[\![\forall \alpha^{\kappa}\,.\,\sigma]\!] & = & \forall \alpha^{\kappa}\,.\,[\![\sigma]\!] & \mathit{with}\ \kappa \neq \mathsf{e} \\
[\![\forall \alpha^{\mathsf{e}}\,.\,\sigma]\!] & = & \forall \alpha^{\mathsf{e}}\,.\,\mathsf{tdict}\langle \overline{\alpha^{\mathsf{e}}}\rangle \rightarrow \langle\rangle [\![\sigma]\!] &
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{mon}\langle \langle\rangle, \epsilon, \tau\rangle & = & \tau \\
\mathsf{mon}\langle \langle l^{\mathsf{u}}\rangle, \langle l^{\mathsf{u}}|\epsilon\rangle, \tau\rangle & = & M_l\langle \epsilon, \tau\rangle \\
\mathsf{mon}\langle \mu, \epsilon, \tau\rangle & = & \text{(evaluated at instantiation)}
\end{array}
$$

The mon operation derives a monadic result type and effect. This cannot be computed though for polymorphic effect types since it is not known whether it will be instantiated to a built-in effect or user-defined effect. We therefore keep this type unevaluated until instantiation time. As such, it is really a *dependent type*. In our case, this is a benign extension to $\lambda^{\kappa\mathsf{u}}$ since $\lambda^{\kappa\mathsf{u}}$ is explicitly typed. There is one other dependent type for giving the type of a polymorphic dictionary (see Figure 7):

$$
\begin{array}{lcl}
\mathsf{tdict}\langle \langle\rangle\rangle & = & \mathit{tdict}\langle M_{\mathit{uid}}\rangle \\
\mathsf{tdict}\langle \langle l^{\mathsf{u}}\rangle\rangle & = & \mathit{tdict}\langle M_l\rangle \\
\mathsf{tdict}\langle \mu\rangle & = & \ldots \text{ (evaluated at instantiation)}
\end{array}
$$

Given the type translation function $[\![\cdot]\!]$ we can now also derive how the *to_eff* and *from_eff* functions are internally implemented. If we apply type translation to their signatures, we can see that both become identity functions. For example, the type translation type of *to_eff* is $[\![M_{\mathit{eff}}\langle \epsilon, \alpha\rangle \rightarrow \langle \mathit{eff}|\epsilon\rangle\ \alpha]\!]$ which is equivalent to $M_{\mathit{eff}}\langle \epsilon, \alpha\rangle \rightarrow \epsilon\ M_{\mathit{eff}}\langle \epsilon, \alpha\rangle$, i.e. we can implement *to_eff* simply as $\lambda x.\ x$. Similarly, *from_eff* is implemented as $\lambda f.\ f()$.

Monadic Abstractions. Figure 6 defines two syntactic abstractions that are used by our translation to bind and lift effect computations.

- $\mathsf{lift}^{\upsilon_t}_{\upsilon_s}(\epsilon, e)$ lifts the expression e from the source υ_s to the target υ_t computed effect. If the computed effects are different $\upsilon_s \neq \upsilon_t$ the lifting is performed via a call to the *unit* field of the dictionary of the target effect $\mathit{dict}_{\upsilon_t}$. Note that the monadic *unit* operator is effect polymorphic thus *lift* is also parametric on an effect ϵ :: e that we use to instantiate the effect variable of *unit*.
- $\mathsf{bind}^{\upsilon}_{\upsilon_x}(\epsilon, e_x, x, e)$ binds the expression e_x to the variable x that appears in e. The expression e_x (resp. e) has computed (*minimum*) effect υ_x (resp. υ) and ϵ is the combined (*maximum*) effect of the binding. If e_x does not have any computed effect binding is simply a val-binding, otherwise binding is performed via a call in the *bind* field of the dictionary of the target effect $\mathit{dict}_{\upsilon_x}$.

As an optimization, if $\upsilon = \langle\rangle$ our system uses the monadic *map* instead of lifting ϵ to υ and using *bind*. As in *lift* the combined effect ϵ is used to instantiate the effect variable of the monadic operators. This particular optimization is similar to the ones used to avoid unnecessary “administrative” redexes, which customary CPS-transform algorithms go to great lengths to avoid [8, 28].

3.3.1. Monadic Translation

Finally we can define the translation relation $e \rightsquigarrow_{\epsilon} e' \mid \upsilon$ as shown in Figure 7, where ϵ is inherited and υ synthesized.

Values. Values have no effect, and compute $\langle\rangle$. Rules (Con) and (Var) are simple: they only translate the type of the expression and leave the expression otherwise un-

Translation $\boxed{e \rightsquigarrow_\epsilon e \mid \upsilon}$

$$(\textsc{Con})\quad \frac{}{c^\sigma \rightsquigarrow_\epsilon c^{[\![\sigma]\!]} \mid \langle\rangle} \qquad (\textsc{Var})\quad \frac{}{x^\sigma \rightsquigarrow_\epsilon x^{[\![\sigma]\!]} \mid \langle\rangle}$$

$$(\textsc{Lam})\quad \frac{e \rightsquigarrow_\epsilon e' \mid \upsilon}{\lambda^\epsilon x : \sigma.\, e \rightsquigarrow_{\epsilon_0} \lambda^{\bar{\epsilon}} x : [\![\sigma]\!].\, \mathsf{lift}^{\bar{\epsilon}}_{\upsilon}(\epsilon, e') \mid \langle\rangle}$$

$$(\text{TL}\textsc{am})\quad \frac{e \rightsquigarrow_\epsilon e' \mid \langle\rangle \quad \kappa \neq \mathsf{e}}{\Lambda\alpha^\kappa.\, e \rightsquigarrow_\epsilon \Lambda\alpha^\kappa.\, e' \mid \langle\rangle} \qquad (\text{TL}\textsc{am}\text{-E})\quad \frac{e \rightsquigarrow_\epsilon e' \mid \langle\rangle}{\Lambda\mu.\, e \rightsquigarrow_\epsilon \Lambda\mu.\lambda^{\langle\rangle}\, dict_\mu : \mathsf{tdict}\langle\overline{\mu}\rangle.\, e' \mid \langle\rangle}$$

$$(\text{TA}\textsc{pp})\quad \frac{e \rightsquigarrow_\epsilon e' \mid \langle\rangle \quad \kappa \neq \mathsf{e}}{e[\tau^\kappa] \rightsquigarrow_\epsilon e'\,[[\![\tau^\kappa]\!]] \mid \langle\rangle} \qquad (\text{TA}\textsc{pp}\text{-E})\quad \frac{e \rightsquigarrow_\epsilon e' \mid \langle\rangle}{e[\epsilon'] \rightsquigarrow_\epsilon e'[[\![\epsilon']\!]]\; dict_{\overline{[\![\epsilon']\!]}} \mid \langle\rangle}$$

$$(\textsc{App})\quad \frac{e_1 \rightsquigarrow_\epsilon e_1' \mid \upsilon_1 \quad e_2 \rightsquigarrow_\epsilon e_2' \mid \upsilon_2 \quad e_1 \downarrow \upsilon_3 \quad \upsilon = \upsilon_1 \oplus \upsilon_2 \oplus \upsilon_3}{e_1\ e_2 \rightsquigarrow_\epsilon \mathsf{bind}^{\upsilon}_{\upsilon_1}(\epsilon, e_1', f, \mathsf{bind}^{\upsilon}_{\upsilon_2}(\epsilon, e_2', y, \mathsf{lift}^{\upsilon}_{\upsilon_3}(\epsilon, f\,y))) \mid \upsilon}$$

$$(\textsc{Val})\quad \frac{e_1 \rightsquigarrow_\epsilon e_1' \mid \upsilon_1 \quad e_2 \rightsquigarrow_\epsilon e_2' \mid \upsilon_2}{\mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow_\epsilon \mathsf{bind}^{\upsilon_2}_{\upsilon_1}(\epsilon, e_1', x, e_2') \mid \upsilon_1 \oplus \upsilon_2}$$

$$(\textsc{If})\quad \frac{e_1 \rightsquigarrow_\epsilon e_1' \mid \upsilon_1 \quad e_2 \rightsquigarrow_\epsilon e_2' \mid \upsilon_2 \quad e_3 \rightsquigarrow_\epsilon e_3' \mid \upsilon_3 \quad \upsilon = \upsilon_1 \oplus \upsilon_2 \oplus \upsilon_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightsquigarrow_\epsilon \mathsf{bind}^{\upsilon}_{\upsilon_1}(\epsilon, e_1', y, \mathsf{if}\ y\ \mathsf{then}\ \mathsf{lift}^{\upsilon}_{\upsilon_2}(\epsilon, e_2')\ \mathsf{else}\ \mathsf{lift}^{\upsilon}_{\upsilon_3}(\epsilon, e_3')) \mid \upsilon}$$

Figure 7. Basic translation rules. Any f and y are assumed fresh.

$$(\text{O}\textsc{pt}\text{-TA}\textsc{pp})\quad \frac{\vdash e : \forall\mu, \alpha_1, ..., \alpha_m.\, \sigma_1 \to \langle l_1, ..., l_n | \mu\rangle\ \sigma_2}{e[\epsilon, \alpha_1, ..., \alpha_m] \downarrow \overline{\langle l_1, ..., l_n\rangle}}$$

$$(\text{O}\textsc{pt}\text{-D}\textsc{efault})\quad \frac{\vdash e : \sigma_1 \to \epsilon\ \sigma_2}{e \downarrow \overline{\epsilon}}$$

Figure 8. Computing minimal effects of function expressions.

changed. In rule (Lam) we see that when translating $\lambda^\epsilon x : \sigma.\, e$ the type σ of the parameter is also translated. Moreover, the effect ϵ dictates the maximum effect in the translation of the body e. Finally, we *lift* the body of the function from the computed minimum effect to ϵ.

Type Operations. Type abstraction and application preserve the computed effect of the wrapped expression e – since the Koka type system guarantees that type abstraction only happens over total expressions, the computed effect is always $\langle\rangle$ in these rules. In (TLam-E) we abstract over an effect variable μ, thus we add an extra value argument, namely, the dictionary of the effect that instantiates μ, i.e. $dict_\mu : \mathsf{tdict}\langle\overline{\mu}\rangle$. Symmetrically, rule (TApp-E) that translates application of the effect ϵ' also applies the dictionary $dict_{\overline{\epsilon'}} : \mathsf{tdict}\langle\overline{\epsilon}'\rangle$ of the effect ϵ'. Note that if the computed effect $\overline{\epsilon'}$ is a user-defined effect, say amb, then the rule directly applies the appropriate dictionary $dict_{amb}$, that is the dictionary value Koka created from the amb effect definition. If the computed effect $\overline{\epsilon'}$ is an effect variable μ, then the rule directly applies the appropriate dictionary $dict_\mu$, that is the variable abstracted by a rule (TLam-E) lower in the translation tree. The final case is the computed effect $\overline{\epsilon'}$ to be the empty effect $\langle\rangle$, in that case the dictionary of the identity effect $dict_{uid}$ is applied. This is because in the computed effects world the total effect $\langle\rangle$ *is* the identity effect *uid*. But in our rules we used the $\langle\rangle$ effect as it is more intuitive.

Application. The rule (App) translates the application $e_1\ e_2$. The minimal computed effect of the application is the union of the computed effects of the function e_1 (that is υ_1), the argument e_2 (that is υ_2) and the computed effect of the body of the the function. The maximum effect of the function is ϵ but using this maximum effect would lead to unoptimized translation, since every application would be unnecessarily lifted to its maximum effect. For example, if we wrote:

$choose(id([False, True]))$

then the unified effect for the *id* application would be *amb* and we would unnecessarily pass an *amb* dictionary to *id* and bind the result.

As an optimization, we compute the minimal effect as $e_1 \downarrow \upsilon_3$, which is presented in Figure 8. In this example, we can apply (Opt-TApp) and use a fully pure invocation of the $id([False, True])$ sub-expression. As it turns out, in practice this optimization is very important and saves much unnecessary binding, lifting, and passing of dictionaries. Since a row-based effect system like Koka uses simple unification, all sub expressions get unified with the final type. With this optimization we basically recover some of the subtyping but we can do it separately from the initial type inference. This is important in practice as we can now clearly separate the two complex phases and keep using simple unification during type inference.

Finally, the rule (Val) translates val-binding `val` $x = e_1; e_2$ by binding e_1 to x in e_2. Similarly, the rule (If) translates if e_1 then e_2 else e_3 by first binding e_1 to a fresh variable x, since e_1 may have user-defined effects and then lifting both branches to the computed effect υ that is the union of the computed effects of the guard υ_1 and the two branches υ_2 and υ_3.

3.4. Soundness

From previous work on type inference for Koka [19] we have that the resulting explicitly typed Koka is well-typed, i.e.

Lemma 1. (*Explicit Koka is well-typed*)
If $\Gamma \vdash k : \sigma \mid \epsilon \rightsquigarrow e$ then $\vdash e : \sigma$.

Here, the relation $\Gamma \vdash k : \sigma \mid \epsilon \rightsquigarrow e$ is the type inference relation defined in [19] where the source term k gets type σ with effect ϵ and a corresponding explicitly typed term e. The new part in this paper is that our translation preserves types according to the $[\![\cdot]\!]$ type translation:

Theorem 1. (*Type Preservation*)
If $\vdash e : \sigma$ and $e \rightsquigarrow_\epsilon e' \mid \langle\rangle$, then $\vdash e' : [\![\sigma]\!]$.

Proof. In the supplementary material we give a proof of soundness for a more general Theorem 3 (*General Type Preservation*): if $\vdash e : \sigma$ and $e \rightsquigarrow_\epsilon e' \mid \upsilon$, then $\vdash e' : \mathsf{mon}\langle \upsilon, \epsilon, [\![\sigma]\!]\rangle$. Theorem 1 follows as a direct implication for a computed effect $\upsilon = \langle\rangle$. □

This is a strong property since Koka has explicit effect types, i.e. it is not possible to have a typed translation simply by using $\bot$, and as such it gives high confidence in the faithfulness of the translation. This is related to the types of *bind* and *unit* for example which are both guaranteed to be total functions (since they are polymorphic in the effect).

4. Implementation

We implemented monadic user-defined effects in Koka and the implementation is available at koka.codeplex.com. Koka's compiler is implemented in Haskell and it transforms Koka source code to JavaScript:

- The compiler takes as input Koka source code as specified in [19].
- Next, it performs type inference and transforms the source code the Koka's intermediate representation which is very similar to $\lambda^{\kappa \mathsf{u}}$ of Figure 3.
- Then, it applies the translation rules of Figure 7, i.e. it uses the inferred types and effects to apply our effect to monadic translation.
- Finally, the monadic intermediate Koka is translated to JavaScript.

The goal of our implementation is to build a sound, complete, and efficient translation with minimum run-time overhead. We get soundness by § 3.4, and in § 4.2 we discuss that the translation is complete, modulo the monomorphic restriction. Moreover (§ 4.1), our translation is optimized to reduce any compile- and run-time overhead as far as possible. We conclude this section by a quantitative evaluation (§ 4.3) of our translation.

4.1. Optimizations

We optimized the translation rules of Figure 7 with three main optimization rules: only translate when necessary, generate multiple code paths for effect polymorphic functions, and use monadic laws to optimize bind structures.

Selective Translation. We observed that most of the functions are *user-defined effect free*. A function is user-defined effect free when (1) it does not generate user-defined effects, (2) it is not effect polymorphic (as any abstract effect can be instantiated with a user-defined one), and (3) all the functions that calls or defines are user-defined effect free.

before translation		after translation		percentage increase	
lines	bytes	lines	bytes	lines	bytes
2678	89038	3248	121668	21.28 %	36.65%

Figure 9. Code size of Koka's library before and after translation.

A user-defined effect free function is translated to itself. Thus our first optimization is to skip translation of such functions. This optimization is crucial in code that does not make heavy use of user-defined effects. As it turns out, 229 out of 287 Koka library functions are not translated!

Two versions of effect polymorphic functions. The translation rule (TApp-E) is quite inefficient when the applied effect ϵ is not user-defined: the identity dictionary is applied and it is used to perform identity monadic operators. User-defined effect free functions are the common case in Koka code, and as such they should not get polluted with identity dictionaries.
As an optimization, when translating an effect polymorphic function we create two versions of the function:

- the monadic version, where the effect variables *can* be instantiated with user-defined effects, thus insertion of monadic operators (*unit* and *bind*) and thus the addition of *the monadic dictionary is required*, and
- the non-monadic version, where the effect variables *cannot* be instantiated with user-defined effects, thus *the monadic dictionary is not required*.

To soundly perform this optimization we use an environment with the effect variables that cannot be instantiated with user-defined effects, which is used as a proof that insertion of monadic operators is not required.
Non-monadic versions *are translated*, but using the effect environment that constraints their effect variables to non-user effects. Still translation is required as these functions may use or produce user-defined effects. Though, in practice translation of non-monadic versions of functions is non-required and optimized by our previous optimization. It turns out that in Koka's library there are 58 polymorphic functions for which we created double versions. None of the non-monadic version requires translation.

Monadic Laws. As a final optimization we used the monadic laws to produce readable and optimized JavaScript code. Concretely, we applied the following three equivalence relations to optimize redundant occurrences:

$$\begin{aligned} \mathsf{bind}(\mathsf{unit}\ x,\ f) &\equiv f\,x \\ \mathsf{bind}(f,\ \mathsf{unit}\ x) &\equiv f\,x \\ \mathsf{map}(\mathsf{unit}\ x,\ f) &\equiv \mathsf{unit}\ (f\,x) \end{aligned}$$

4.2. The Monomorphism Restriction

The *Monomorphism Restriction* restricts value definitions to be effect monomorphic. Consider the following program

```
function poly(x : a, g : (a) → e b) : e b
val mr = if (expensive() ≥ 0) then poly else poly
```

How often is *expensive*() executed? The user can naturally assume that the call is done once, at the initialization of *mr*. But, since *mr* is effect polymorphic (due to *poly*), our translation inserts a dictionary argument. Thus, the call is executed as many times as *mr* is referenced. To avoid this situation the Koka compiler rejects such value definitions, similar to the *monomorphism restriction* of Haskell. Aside

program	static time compile	trans. time	trans. %	file size lines
amb	107 ms	2.795 ms	2.60 %	67
parser	89 ms	2.582 ms	2.88 %	72
async	117 ms	3.155 ms	2.67 %	166
core	6057 ms	16.424 ms	0.27 %	2465

Figure 10. Quantitative evaluation of static time for user-defined effects.

from this restriction, our translation is complete, *i.e.,* we accept and translate all programs that plain Koka accepts.

4.3. Evaluation

Finally, we give a quantitative evaluation of our approach that allows us to conclude that monadic user-defined effects can be use to produce high quality code without much affecting the compile- or running-time of your program.

In Figure 10 we present the static metrics and the file size of our four benchmarks: (1) *amb* manipulates boolean formulas using the ambiguous effect, (2) *parser* parses words and integers from an input file using the parser effect, (3) *async* interactively processes user's input using the asynchronous effect, and (4) *core* is Koka's core library that we translate so that all library's functions can be used in effectful code. On these benchmarks we count the file size in lines, the total compilation time, the translation time and we compute the percentage of the compilation time that is spent on translation. The results are collected on a machine with an Intel Core i5.

Static Time. As Figure 10 suggests the compile-time spend in translation is low (less that 3% of compilation time). More importantly, the fraction of the time spend in translation is minor in code that does not use monadic effects, mostly due to our optimizations (§ 4.1).

Run Time. To check the impact of our translation on run-time we created "monadic" versions of our examples, i.e. a version of the *amb* that uses lists instead of the *amb* effect and a version of the *parser* that uses a *parser* data type. We observed that our monadic translation does not have any run-time cost mostly because it optimizes away redundant calls (§ 4.1).

Thus, we conclude that our proposed abstraction provides better quality of code with no run-time and low static-time overhead.

5. Related work

The problems with arbitrary effects have been widely recognized, and there is a large body of work studying how to delimit the scope of effects. There have been many effect typing disciplines proposed. Early work is by Gifford and Lucassen [11, 21] which was later extended by Talpin [32] and others [24, 31]. These systems are closely related since they describe polymorphic effect systems and use type constraints to give principal types. The system described by Nielson et al. [24] also requires the effects to form a complete lattice with meets and joins. Wadler and Thiemann [35] show the close connection between monads [23, 34] and the effect typing disciplines.

Java contains a simple effect system where each method is labeled with the exceptions it might raise [13]. A system for finding uncaught exceptions was developed for ML by Pessaux et al. [25]. A more powerful system for tracking effects was developed by Benton [5] who also studies the semantics of such effect systems [6]. Recent work on effects in Scala [29] shows how restricted polymorphic effect types can be used to track effects for many programs in practice.

Marino et al. created a generic type-and-effect system [22]. This system uses privilege checking to describe analytical effect systems. Their system is very general and can express many properties but has no semantics on its own. Banados et al. [2] layer a gradual type system on top this framework. This may prove useful for annotating existing code bases where a fully static type system may prove too conservative.

This paper relies heavily on a *type directed* monadic translation. This was also described in the context of ML by Swamy et al. [30], where we also showed how to combine multiple monads using monad morphisms. However, Koka we use row-polymorphism to do the typing, while our previous paper used subtyping. A problem with subtyping is that the inferred types can be difficult to understand. For example, in an η-robust version, the type inferred for function composition in [30] is:

$$\forall a\ b\ c\ \mu_1\ \mu_2\ \mu_3\ \mu_4\ \mu.\ (\mu_1 \geq \mu, \mu_2 \geq \mu_)\ \Rightarrow \\ (b \to \mu_2\ c) \to \mu_3\ (a \to \mu_1\ b) \to \mu_4\ (a \to \mu\ c)$$

which is much harder to read than the type inferred in Koka:

$$\forall\langle a, b, c, \epsilon\rangle\ (b \to \epsilon\ c,\ a \to \epsilon\ b) \to \epsilon\ c$$

where no constraints are present. Just as in our system, there is also at most one monad, but you can define the morphisms between monads. However, that means that you may need to define many combinations of smaller monads. For Koka, we wanted fine-grained effect control with effects like *div*, *exn*, *read*$\langle h\rangle$ etc. This would be a problem when users need to define all combinations with their user defined monad (so in [30] we conveniently made all those basic effects part of the basic *Id* or *ML* monad).

A similar approach to [30] is used by Rompf et al. [28] to implement first-class delimited continuations in Scala which is essentially done by giving a monadic translation. Similar to our approach, this is also a *selective* transformation; i.e. only functions that need it get the monadic translation. Both of the previous works are a *typed* approach where the monad is apparent in the type. Early work by Filinksi [9, 10] showed how one can embed any monad in any strict language that has mutable state in combination with first-class continuations (i.e. *callcc*). This work is untyped in the sense that the monad or effect is not apparent in the type.

Algebraic effect handlers described by Plotkin et al. [26] are not based on monads, but on an algebraic interpretation of effects. Even though monads are more general, algebraic effects are still interesting as they compose more easily. Bauer and Pretnar describe a practical programming model with algebraic effects [3] and a type checking system [4]. Even though this approach is quite different than the monadic approach that we take, the end result is quite similar. In particular, the idea of handlers to *discharge* effects, appears in our work in the form of the *from* primitives induced by an `effect` declaration.

Many languages have extensions to support a particular effect or monad. For example, both C# [7] and Scala [1] have extensions to make programming with asynchronous code more convenient using *await* and *async* keywords. In those cases, the compiler needs to be significantly extended to generate state machines under the hood. Similarly, many languages have special support for iterators which can be implemented using a list-like monad.

6. Conclusion & Future Work

Using the correspondence between monads and effects [35], we have shown how you can *define* the semantics of an effect in terms of a first-class monadic value, but you can *use* the monad using a first-class effect type. We provide a prototype implementation that builds on top of Koka's type- and effect inference system.

As future work we would like take the work on monadic programming in ML [30] and explore how multiple user-defined effects can co-exist and how to automatically infer the morphisms between them. Furthermore, we would like to apply the existing system to program larger real-world applications, like an asynchronous web server.

In "The essence of functional programming" [34], Wadler remarks that "*by examining where monads are used in the types of programs, one determines in effect where impure features are used. In this sense, the use of monads is similar to the use of effect systems*". We take the opposite view: by examining the effect types one can determine where monads are to be used. In Wadler's sense of the word, the essence of effectful programming is monads.

References

[1] Scala async proposal. 2013. URL http://docs.scala-lang.org/sips/pending/async.html.

[2] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *ICFP*, 2014. doi:10.1145/2628136.2628149.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, 1203.1539, 2012. URL http://arxiv.org/abs/1203.1539.

[4] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10 (4), 2014. doi:10.2168/LMCS-10(4:9)2014.

[5] Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI*, 2007. doi:10.1145/1190315.1190320.

[6] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*, 2007. doi:10.1145/1273920.1273932.

[7] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause 'N' Play: Formalizing asynchronous C#. In *ECOOP*, 2012. doi:10.1007/978-3-642-31057-7_12.

[8] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass cps transformations. *J. Funct. Program.*, 17 (6): 793–812, November 2007. doi:10.1017/S0956796807006387.

[9] Andrzej Filinski. Representing monads. In *POPL*, 1994.

[10] Andrzej Filinski. Controlling effects. Technical report, U.S. Dept. of Labor, OSHA, 1996.

[11] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, 1986. doi:10.1145/319838.319848.

[12] Jean-Yves Girard. The System F of variable types, fifteen years later. *TCS*, 1986. doi:10.1016/0304-3975(86)90044-7.

[13] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. 1996.

[14] Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. In *MSFP*, 2014.

[15] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.

[16] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*. 2009. doi:10.1007/978-3-642-03034-5_17.

[17] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *ICFP*, 1997. doi:10.1145/258948.258970.

[18] Daan Leijen. Koka: Programming with row-polymorphic effect types. Technical Report MSR-TR-2013-79, Microsoft Research, 2013.

[19] Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP*, 2014. doi:10.4204/EPTCS.153.8.

[20] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001. URL http://www.haskell.org/haskellwiki/Parsec.

[21] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988. doi:10.1145/73560.73564.

[22] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI*, 2009. doi:10.1145/1481861.1481868.

[23] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 1991. doi:10.1016/0890-5401(91)90052-4.

[24] Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *LOMAPS*, 1997. doi:10.1007/3-540-62503-8_8.

[25] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *POPL*, 1999. doi:10.1145/292540.292565.

[26] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP*, volume 5502 of *LNCS*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

[27] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, 1974.

[28] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328, 2009. doi:10.1145/1596550.1596596.

[29] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, 2012. doi:10.1007/978-3-642-31057-7_13.

[30] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011. doi:10.1145/2034773.2034778.

[31] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 1994. doi:10.1006/inco.1994.1046.

[32] J.P. Talpin. *Theoretical and practical aspects of type and effect inference*. PhD thesis, Ecole des Mines de Paris and University Paris VI, Paris, France, 1993.

[33] Stefan Tilkov and Steve Vinoski. NodeJS: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 2010.

[34] Philip Wadler. The essence of functional programming. POPL, 1992. doi:10.1145/143165.143169.

[35] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *TOLC*, 2003. doi:10.1145/601775.601776.