From Monads to Effects and Back

Niki Vazou and Daan Leijen

[1] UC San Diego
[2] Microsoft Research

Abstract. The combination of monads and effects leads to a clean and easy to reason about programming paradigm. Monadic programming is easy to reason about, but can be cumbersome, as it requires explicit lifting and binding. In this paper, we combine monads and effects within a single programming paradigm: we use monads to *define* the semantics of effect types, and then, use the effects to *program* with those monads. We implemented an extension to the effect type system of Koka [15] with *user defined effects*. We use a type-directed translation to automatically lift effectful into monadic programs, by inserting bind- and unit operations.

1. Introduction

Monads (proposed by Moggi and others [17, 28]) are used in programming languages to wrap effectful computations, but they can be cumbersome to program with as they require explicit lifting and binding. In this paper, we combine *monads* and *effect typing* (proposed by Gifford and others [9, 25]) within a single programming paradigm: we use monads to *define* the semantics of effect types, and then, use the effect types to *program* with those monads.

We implemented these ideas as an extension of the effect type system of Koka [15] – a strict, JavaScript-like, strongly typed programming language that automatically infers the type and *effect* of functions. Koka has a type inference system for a set of standard effects like divergence, exceptions, heap operations, input/output, etc. Here we extend the effect system with *monadic user defined effects*, where we can define our own effect in terms of any monad. As an example, we define an *amb* effect for ambiguous computations [13]. 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* are the usual list-monad definitions.

Using the above effect definition we can write functions that have the *amb*iguous effect. For example we can write a *flip* primitive that returns either true or false and use it to compute the truth table of *xor*:

```
function flip : () → amb bool
```

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

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.

Translation. Koka uses 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 xor() : list⟨bool⟩ {
  bind( flip(), fun(p) {
    bind( flip(), fun(q) {
      unit( (p || q) && not(p&&q) )
})})}
```

Here we see how the *unit* and *bind* of the effect declaration are used: *bind* is inserted whenever a monadic value is returned and passed the current continuation at that point.

The capture of the continuation at every *bind* makes monadic effects very expressive. For example, 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 semi-colon".

Our *contributions* are summarized as follows:

- Using the correspondence between monads and effects [29], 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 [15] 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 [24] and automatically lifts and binds computations.
- 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, as we further explain in § 2.2.1.

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.

Koka is a strict programming language with a type system that tracks effects. The type of a function in Koka is 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. We can leave out the effect and write $\tau \to \tau'$ as a shorthand for the total function without any side effect: $\tau \to \langle\rangle\ \tau'$. A key observation on Moggi's early work on monads [17] was that *values* and *computations* should be assigned a different type. Koka applies 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.

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 [15] 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).

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

In the introduction we saw how one can define and use the ambiguous *amb* effect with *flip* and *xor* operations. We now discuss the definition and translation in more detail. The *amb* effect is 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)
  }
}
```

Defining the *amb* effect amounts to defining the standard list monad, which can be further simplified by removing 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, and a monadic type alias:

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

that allow us to go from monadic values to effect types and vice versa. These are typed versions of the *reify* and *reflect* methods of Filinski's embedding [6].

We use the above primitives to define *flip* that creates the *amb*iguous effect and *main* that evaluates the effectful computation:

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

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

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 [13, 24].

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) )
}                                         })})}
```

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 typed 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 [28], 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 that 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)
}
```

```
// 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*

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

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 between user defined effects Koka allows combination of user defined effects. Consider a behavior user defined effect, which repre-

sents computations whose value varies with time, as in functional reactive programs [5]. We encode the *beh* effect as a function from *time* to *a*. Since a *beh*aviour is a function, it may have effects itself: it can diverge or throw exceptions for example. This means that we need to parameterize the *beh* effect with two type parameters one for the value *a* and one for the effect *e*:

```
effect beh⟨e, a⟩ = time → e a { ... }
```

With the above definition, Koka automatically creates a type alias for the behavioral monad and the respective *unit* and *bind* operators:

```
alias M_beh⟨e, a⟩ = time → e a;
ub : a → e M_beh⟨e, a⟩;
bb : (M_beh⟨e, a⟩, a → e M_beh⟨e, b⟩) → e M_beh⟨e, b⟩
```

As with the *amb* effect, the user can define primitives that, for example, return the temperature and humidity over time:

```
temp : () → beh int;
hum : () → beh int;
```

We use these primitives to define a function that states that one goes out when temperature is more than 70°F and humidity less than 80%. Koka automatically translates the effectful function to its monadic version:

```
function go_out() : beh bool{
  val t = temp()
  val h = hum()
  (t ≥ 70 && h ≤ 80)
}
```

⇝

```
function go_out() : time → bool{
  bb( temp(), fun(t) {
    bb( hum(), fun(h) { ub(
      (t ≥ 70 && h ≤ 80)
    })})}
```

Next, we want to insert ambiguity into the above function. Following Swamy et al. [24] we combine the ambiguous and behavioral effects by tupling them

```
effect ⟨amb, beh⟩⟨e, a⟩ = time → e list⟨a⟩ { ... }
```

and Koka creates the appropriate monadic operators

```
alias M_ab⟨e, a⟩ = time → e list⟨a⟩;
uab : a → e M_ab⟨e, a⟩;
bab : (M_ab⟨e, a⟩, a → e M_ab⟨e, b⟩) → e M_ab⟨e, b⟩;
```

Then, we define morphisms to lift from a single to the joined effect

```
morphism amb ⟨amb, beh⟩{ fun(xs){ fun(t){ xs }} }
morphism beh ⟨amb, beh⟩{ fun(b){ fun(t){ [b(t)] }} }
```

With the above morphism definitions, Koka derives internal morphism functions

```
a2ab :: M_amb ⟨e, a⟩ → e M_⟨amb, beh⟩⟨e, a⟩;
b2ab :: M_beh ⟨e, a⟩ → e M_⟨amb, beh⟩⟨e, a⟩;
```

and use them to automatically translate our modified *go_out* function that combines the two user defined effects:

```
function go_out()                          function go_out()
{                                          {
  val t = temp()                             bab( b2ab(temp()), fun(t) {
  val h = hum()                                bab( b2ab(hum()), fun(h) {
  val u = flip()                       ⇝         bab( a2ab(flip()), fun(u) {
                                                   uab(
  (u || (t >= 70 && h <= 80))                    (u || (t >= 70 && h <= 80))
}                                          })})}
```

This technique for combining monads by tupling is taken from [24]. But, as further discussed in § 4 our current work, though highly inspired, crucially differs from [24] in that the use of effect polymorphism (instead of effect subtyping that was previously used) makes types much simpler.

There are various language design aspects with regard to morphism declarations – here we highlight the most important ones and defer to [24] and [12] for a more in-depth discussion. First of all, since effect rows are equivalent up to re-ordering of labels, we can only declare one combined monad for a specific set of user-defined effects. For example, we can combine $\langle amb, beh \rangle$ in only one of the two possible ways (within the scope of a module). Moreover, the compiler rejects duplicate definitions. Finally, if we assume that the morphism laws hold, the compiler could derive morphisms from the existing ones, i.e. morphisms from *ma* to *mb*, and *mb* to *mc* can be combined to give rise to a morphism from *ma* to *mc*. Currently, we assume that the user provides all required morphisms explicitly, but we plan to implement automatic morphism derivation.

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.

3.1. Syntax

Figure 2 defines the syntax of expressions and types of $\lambda^{\kappa\mathsf{u}}$, a polymorphic explicitly typed λ-calculus. $\lambda^{\kappa\mathsf{u}}$ is System F [10, 21] extended with 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 $\Lambda\alpha^{\kappa}\, .\, e$. 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.

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 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 kind system. We annotate the type τ with its kind κ, as τ^{κ}. We have the kind of types $(*)$ and the kind of functions $\to$, kinds for effect rows (e), effect constants (k), and user-defined

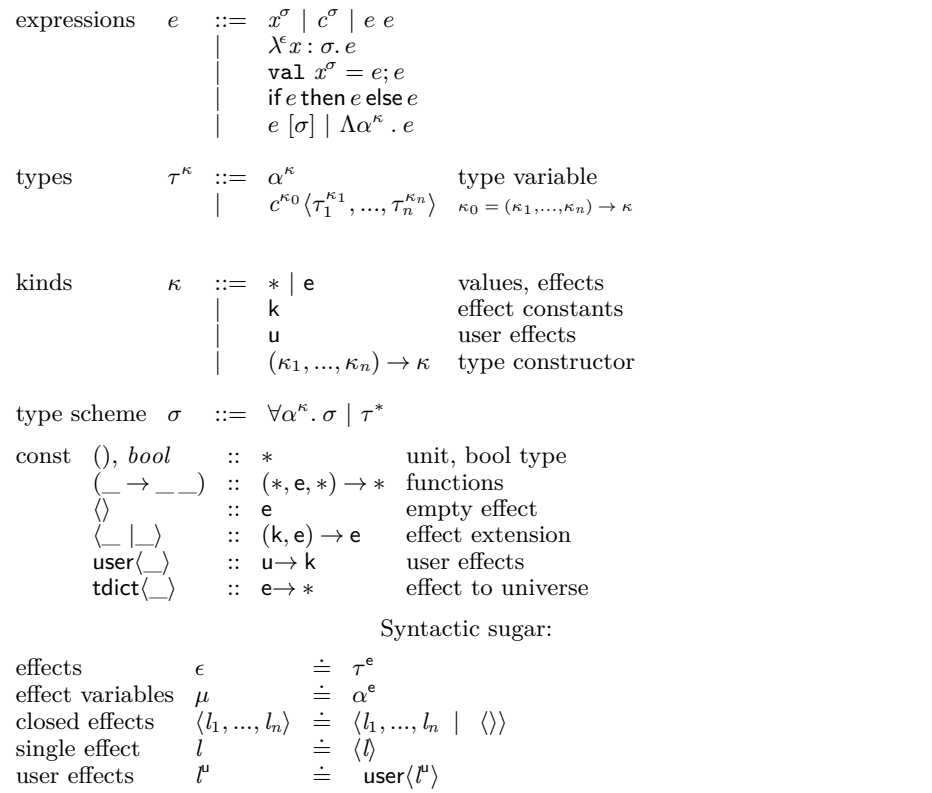

Figure 2. Syntax of explicitly typed Koka, $\lambda^{\kappa\mathsf{u}}$.

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 2 includes definition of type constants and syntactic sugar required to simplify the rest of this section.

Type rules. Figure 3 describes type rules for $\lambda^{\kappa\mathsf{u}}$ where the judgment $\vdash e : \sigma|\,\epsilon$ assigns type σ and effect ϵ to 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. $\lambda^{\kappa\mathsf{u}}$ is explicitly typed, in that variables are annotated with their type and functions with their effect, hence, type checking does not require an environment. By construction,

Type and Effect Checking $\boxed{\vdash e \,:\, \sigma \,|\, \epsilon}$

$$\frac{}{\vdash c^{\sigma} \,:\, \sigma \,|\, \epsilon} \qquad \frac{\vdash e \,:\, \forall\alpha.\, \sigma \,|\, \epsilon}{\vdash e[\tau] \,:\, \sigma[\alpha \mapsto \tau] \,|\, \epsilon} \qquad \frac{\vdash e \,:\, \sigma \,|\, \langle\rangle}{\vdash \Lambda\alpha^{\kappa}.\, e \,:\, \forall\alpha^{\kappa}.\, \sigma \,|\, \epsilon}$$

$$\frac{}{\vdash x^{\sigma} \,:\, \sigma \,|\, \epsilon} \qquad \frac{\vdash e \,:\, \tau_2 \,|\, \epsilon}{\vdash \lambda^{\epsilon} x : \tau_1.\, e \,:\, \tau_1 \to \epsilon\ \tau_2 \,|\, \epsilon'} \qquad \frac{\vdash e_1 \,:\, \tau_1 \to \epsilon\ \tau_2 \,|\, \epsilon \quad \vdash e_2 \,:\, \tau_1 \,|\, \epsilon}{\vdash e_1\ e_2 \,:\, \tau_2 \,|\, \epsilon}$$

$$\frac{\vdash e_1 \,:\, \sigma_1 \,|\, \epsilon \quad \vdash e_2 \,:\, \sigma_2 \,|\, \epsilon}{\vdash \mathsf{val}\ x \,=\, e_1;\ e_2 \,:\, \sigma_2 \,|\, \epsilon} \qquad \frac{\vdash e \,:\, \mathsf{bool} \,|\, \epsilon \quad \vdash e_1 \,:\, \sigma \,|\, \epsilon \quad \vdash e_2 \,:\, \sigma \,|\, \epsilon}{\vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \,:\, \sigma \,|\, \epsilon}$$

Type Checking $\boxed{\vdash e \,:\, \sigma}$

$$\frac{\vdash e \,:\, \sigma \,|\, \epsilon}{\vdash e \,:\, \sigma}$$

Figure 3. Type rules for explicitly typed Koka.

Declarations of User Effects and Morphisms $\boxed{\Gamma \vdash \mathit{def} \,:\, \Gamma}$

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

$$(\textsc{Morph}) \quad \frac{s \equiv \langle l_1, ..., l_n\rangle \quad t \equiv \langle l_1, ..., l_n, ..., l_m\rangle \quad \Gamma \vdash_{\mathsf{k}} e \,:\, \forall\alpha\mu.\ M_s\langle\mu,\ \alpha\rangle \to \mu\ M_t\langle\mu,\ \alpha\rangle}{\Gamma \vdash \mathsf{morphism}\ s\ t\ \{\ e\ \}\ :\Gamma,\ s \rhd t \,:\, \forall\alpha\mu.\ M_s\langle\mu,\ \alpha\rangle \to \mu\ M_t\langle\mu,\ \alpha\rangle}$$

Figure 4. Type rule for effect and morphism declarations.

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 [15]. Next, we write $\vdash e \,:\, \sigma$ if there exists a derivation $\vdash e \,:\, \sigma \,|\, \epsilon$ for some effect ϵ.

3.2. Type inference for effect and morphism declarations

In this subsection we present how Koka preprocesses the effect and morphism declarations provided by the user. Figure 4 summarizes the rules that the Koka compiler follows to desugar the user definitions to $\lambda^{\kappa\mathsf{u}}$. After desugaring, Koka proceeds to effect- and type- inference as presented in previous work [14, 15].

3.2.1. Effect Declarations Before we look at the general type inference rule for effect declarations we start with describing 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 relates the effect name with its monadic representation.

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

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

$$\begin{array}{l} \mathit{unit} \,:\, \forall\alpha\mu.\, \alpha{\to}\, \mu\ M_{\mathit{uid}}\langle\mu,\alpha\rangle \\ \mathit{bind} \,:\, \forall\alpha\beta\mu.\, (M_{\mathit{uid}}\langle\mu,\alpha\rangle,\ \alpha{\to}\, \mu\ M_{\mathit{uid}}\langle\mu,\beta\rangle) \to \mu\ M_{\mathit{uid}}\langle\mu,\beta\rangle \end{array}$$

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:

- $\mathit{uid}^{\mathsf{u}}$ – the effect constant that can be used inside types,
- $\mathit{to}_{\mathit{uid}} \,:\, \forall\alpha\mu.\, (M_{\mathit{uid}}\langle\mu,\alpha\rangle) \to \langle \mathit{uid}|\mu\rangle\ \alpha$ – the function that converts monadic computations to effectful ones,
- $\mathit{from}_{\mathit{uid}} \,:\, \forall\alpha\beta\mu.\, (() \to \langle \mathit{uid}|\mu\rangle\ \alpha) \to \mu\ M_{\mathit{uid}}\langle\mu,\alpha\rangle$ – the dual function that converts effectful function to their monadic equivalent, and finally,
- $\mathit{dict}_{\mathit{uid}}$ – 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. $\mathit{dict}_{\mathit{uid}}$), is a structure that contains the monadic operators *unit* and *bind* of the effect. It can also include the monadic *map* which will otherwise be automatically derived from *unit* and *bind*. Thus, we define the dictionary structure as a type that is polymorphic on the particular monad, represented as type variable $m :: (\mathsf{e}, *) \to *$:

$$\begin{array}{l} \mathsf{struct}\ \mathit{tdict}\langle m\rangle\ \{ \\ \quad \mathit{unit} \,:\, \forall\alpha\mu.\, \alpha{\to}\, \mu\ m\langle\mu,\alpha\rangle \\ \quad \mathit{map} \,:\, \forall\alpha\beta\mu.\, (m\langle\mu,\alpha\rangle,\ \alpha{\to}\, \beta) \to \mu\ m\langle\mu,\beta\rangle \\ \quad \mathit{bind} \,:\, \forall\alpha\beta\mu.\, (m\langle\mu,\alpha\rangle,\ \alpha{\to}\, \mu\ m\langle\mu,\beta\rangle) \to \mu\ m\langle\mu,\beta\rangle \end{array}$$

With this we can type $\mathit{dict}_{\mathit{uid}} \,:\, \mathit{tdict}\langle M_{\mathit{uid}}\rangle$.

General user-defined effects. Figure 4 generalizes the previous concrete example to any user-defined effect declaration. The judgment:

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

states that under a kind- and type- environment Γ, the effect declaration *eff* results in the type environment Γ' that is extended with the needed types and primitives implied by *eff*. As shown in Figure 4, we first check well-formedness of the effect types, then check that *unit* and *bind* operations have the proper types. Finally, the environment is extended with these types and values.

3.2.2. Morphism Declarations *As a morphism example*, we assume the existence of the two user defined effects from the Overview (§ 2), the *amb*iguous and the *beh*aviour. Moreover, we assume that the user appropriately defined their joined effect $\langle \mathit{amb}, \mathit{beh}\rangle$. These three effect definitions yield three aliases:

$$\begin{array}{l} \mathsf{alias}\ M_{\mathit{amb}}\langle\epsilon,\alpha\rangle\ =\ \mathsf{list}\langle\alpha\rangle \\ \mathsf{alias}\ M_{\mathit{beh}}\langle\epsilon,\alpha\rangle\ =\ \mathsf{time}\to\epsilon\ \alpha \\ \mathsf{alias}\ M_{\langle \mathit{amb},\ \mathit{beh}\rangle}\langle\epsilon,\alpha\rangle\ =\ \mathsf{time}\to\epsilon\ \mathsf{list}\langle\alpha\rangle \end{array}$$

Then, the user can define morphisms that go from *amb* or *beh* to the combined effect $\langle \mathit{amb}, \mathit{beh}\rangle$

```
morphism amb ⟨amb, beh⟩ {
 fun(xs : list⟨a⟩) : time → e list⟨a⟩{
  fun(t){xs}
 }}
```

```
morphism beh ⟨amb, beh⟩ {
 fun(x : time → e a) : time → e list⟨a⟩{
  fun(t){[x(t)]}
 }}
```

From the above definitions, Koka will generate two morphism functions:

$$amb \rhd \langle amb,\ beh\rangle\ :\ \forall\alpha\mu.\ M_{amb}\langle\mu,\ \alpha\rangle \to \mu\ M_{\langle amb,\ beh\rangle}\langle\mu, \alpha\rangle$$
$$beh \rhd \langle amb,\ beh\rangle\ :\ \forall\alpha\mu.\ M_{beh}\langle\mu,\ \alpha\rangle \to \mu\ M_{\langle amb,\ beh\rangle}\langle\mu, \alpha\rangle$$

The above morphisms are internal Koka functions that will be used at the translation phase to appropriately lift the monadic computations.

General user-defined morphisms. Figure 4 generalizes the previous concrete example to any morphism declaration. The judgment $\Gamma \vdash \mathsf{morphism}\ s\ t\ e\ :\ \Gamma'.$ states that under a kind- and type- environment Γ, the morphism declaration from effect raws s to t results in a new type environment Γ' that is extended with the morphism from the source effect s to the target effect t, when the expression e has the appropriate morphism type. The first premise ensures that s is always a sub-effect of the target effect t.

3.3. Type-directed monadic translation

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

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: $\upsilon ::= \mu \mid \langle l_1^{\mathsf{u}},\ ...,\ l_n^{\mathsf{u}}\rangle \quad (n \geq 0)$

Thus, computed effects can be a row of user-effect labels (including the empty row) or an effect variable. Note that because user effects are always constants, and according to the equivalence rules for rows [15], we consider computed effect rows equal up to reordering. For example $\langle l_2^{\mathsf{u}},\ l_1^{\mathsf{u}}\rangle \equiv \langle l_1^{\mathsf{u}},\ l_2^{\mathsf{u}}\rangle$.

As shown by the grammar, the computed effects are restricted in that a row of monadic effects cannot end with a polymorphic tail μ. This limits the functions that a user can write: no user-defined effects can be in a row that has a polymorphic tail. We believe that this restriction is not too severe in practice since it only restricts functions that are higher-order over user-defined effects where the function parameter is open-ended in the side-effects it can have. This is quite unusual and easy to circumvent by stating the allowed effects explicitly. The advantage of adding this restriction is great though: we completely circumvent the need to add constraints to the type language and can simplify the translation significantly compared to earlier work [24].

We convert a regular effect type ϵ to a computed effect $\bar{\epsilon}$, and dually, we apply $\tilde{\epsilon}$ to an effect ϵ to remove the user defined effects:

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

$$
\begin{array}{lcl}
\mathsf{bind}^{\upsilon}_{\langle\rangle}(\epsilon, e_x, x, e) & = & \mathsf{val}\ x\ =\ e_x;\ e \\
\mathsf{bind}^{\upsilon}_{\upsilon_x}(\epsilon, e_x, x, e) & = & \mathit{dict}_{\upsilon_x}.\mathit{map}\,\langle\sigma_x, \sigma, \epsilon\rangle(e_x,\ \lambda^{\langle\rangle}\ x\ :\ \sigma_x.\ e),\ \mathsf{with} \vdash e_x : \mathsf{mon}\langle\upsilon_x, \epsilon, \sigma_x\rangle, \vdash e : \sigma \\
\mathsf{bind}^{\upsilon}_{\upsilon_x}(\epsilon, e_x, x, e) & = & \mathit{dict}_{\upsilon_x \oplus \upsilon}.\mathit{bind}\langle\sigma_x, \sigma, \epsilon\rangle(e'_x,\ e'),\ \mathsf{with} \vdash e_x : \mathsf{mon}\langle\upsilon_x, \epsilon, \sigma_x\rangle, \vdash e : \mathsf{mon}\langle\upsilon, \epsilon, \sigma\rangle \\
\multicolumn{3}{l}{\quad \mathsf{where}\ e'_x\ =\ \mathsf{lift}^{(\upsilon_x \oplus \upsilon)}_{\upsilon_x}(\epsilon, e_x),\ e'\ =\ \lambda^{\tilde{\epsilon}}\ x\ :\ \sigma_x.\ (\mathsf{lift}^{(\upsilon_x \oplus \upsilon)}_{\upsilon}(\epsilon, e))}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{lift}^{\upsilon}_{\upsilon}(\epsilon, e) & = & e \\
\mathsf{lift}^{\upsilon}_{\langle\rangle}(\epsilon, e) & = & \mathit{dict}_{\upsilon}.\mathit{unit}\langle\sigma,\ \epsilon\rangle(e)\ \ \mathsf{where}\ \upsilon \neq \langle\rangle, \vdash e\ :\ \sigma \\
\mathsf{lift}^{\upsilon_t}_{\upsilon_s}(\epsilon, e) & = & \upsilon_s \rhd \upsilon_t\langle\sigma,\ \epsilon\rangle(e) \qquad \mathsf{where} \vdash e\ :\ \mathsf{mon}\langle\upsilon_s, \epsilon, \sigma\rangle
\end{array}
$$

Figure 5. Helper functions for binding and lifting.

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

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

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

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

$$
(\textsc{TApp})\quad \frac{e\ \rightsquigarrow_{\epsilon}\ e' \mid \upsilon \quad \kappa \neq \mathsf{e}}{e[\tau^{\kappa}]\ \ \rightsquigarrow_{\epsilon}\ e'\ [[\![\tau^{\kappa}]\!]] \mid \upsilon} \qquad (\textsc{TApp-E})\quad \frac{e\ \rightsquigarrow_{\epsilon}\ e' \mid \upsilon}{e[\epsilon']\ \ \rightsquigarrow_{\epsilon}\ e'[[\![\epsilon']\!]]\ \mathit{dict}_{\overline{[\![\epsilon']\!]}} \mid \upsilon}
$$

$$
(\textsc{App})\quad \frac{e_1\ \rightsquigarrow_{\epsilon}\ e'_1 \mid \upsilon_1 \quad e_2\ \rightsquigarrow_{\epsilon}\ e'_2 \mid \upsilon_2 \quad e_1 \downarrow \upsilon_3 \quad \vdash e_1\ :\ \sigma_1 \to \epsilon\ \sigma_2 \quad ue_3\ =\ \overline{\epsilon} \quad \upsilon =\ \upsilon_1 \oplus \upsilon_2 \oplus \upsilon_3}{e_1\ e_2\ \rightsquigarrow_{\epsilon}\ \mathsf{bind}^{\upsilon_2 \oplus \upsilon_3}_{\upsilon_1}(\epsilon, e'_1, f, \mathsf{bind}^{\upsilon_3}_{\upsilon_2}(\epsilon, e'_2, y, f\,y)) \mid \upsilon}
$$

$$
(\textsc{Val})\quad \frac{e_1\ \rightsquigarrow_{\epsilon}\ e'_1 \mid \upsilon_1 \quad e_2\ \rightsquigarrow_{\epsilon}\ e'_2 \mid \upsilon_2}{\mathsf{val}\ x\ =\ e_1;\ e_2\ \rightsquigarrow_{\epsilon}\ \mathsf{bind}^{\upsilon_2}_{\upsilon_1}(\epsilon, e'_1, x, e'_2) \mid \upsilon_1\ \oplus \upsilon_2}
$$

$$
(\textsc{If})\quad \frac{e_1\ \rightsquigarrow_{\epsilon}\ e'_1 \mid \upsilon_1 \quad e_2\ \rightsquigarrow_{\epsilon}\ e'_2 \mid \upsilon_2 \quad e_3\ \rightsquigarrow_{\epsilon}\ e'_3 \mid \upsilon_3 \quad \upsilon =\ \upsilon_1 \oplus \upsilon_2 \oplus \upsilon_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \rightsquigarrow_{\epsilon}\ \mathsf{bind}^{\upsilon_2 \oplus \upsilon_3}_{\upsilon_1}(\epsilon, e'_1, y, \mathsf{if}\ y\ \mathsf{then}\ \mathsf{lift}^{\upsilon_2 \oplus \upsilon_3}_{\upsilon_2}(\epsilon, e'_2)\ \mathsf{else}\ \mathsf{lift}^{\upsilon_2 \oplus \upsilon_3}_{\upsilon_3}(\epsilon, e'_3)) \mid \upsilon}
$$

Figure 6. Basic translation rules. Any f and y are assumed fresh.

Note that this function is partial: when a type is passed that combines a user-defined effect with a polymorphic tail it fails and the compiler raises an error that the program is *too polymorphic*. To join computed effects we use the $\oplus$ operator:

$$
\begin{array}{lll}
\langle\rangle \oplus \upsilon_2 & = \upsilon_2 & \\
\mu \oplus \mu & = \mu & \\
\langle l \mid \upsilon_1\rangle\ \oplus \langle l \mid \upsilon_2\rangle & = \langle l \mid \upsilon_1\ \oplus \upsilon_2\rangle & \\
\langle l \mid \upsilon_1\rangle\ \oplus \upsilon_2 & = \langle l \mid \upsilon_1\ \oplus \upsilon_2\rangle & \mathsf{if}\ l \notin \upsilon_2
\end{array}
$$

Again, this function is partial but we can show that the usage in the translation over a well-typed koka program never leads to an undefined case.

Type translation. The type operator $[\![\cdot]\!]$ translates effectful- to monadic types.

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

For function types $\tau \to \epsilon\ \tau'$ the effect ϵ is split to the build-in effect portion $\tilde{\epsilon}$ and the user-defined portion $\overline{\epsilon}$. The effect of the translated type is only the build-in portion $\tilde{\epsilon}$ while the result is monadically wrapped according to the user-defined portion $\overline{\epsilon}$ using the mon operator

$$
\begin{array}{lcl}
\mathsf{mon}\langle\langle\rangle, \epsilon, \sigma\rangle & = & \sigma \\
\mathsf{mon}\langle\langle l_1^\mathsf{u},\ \ldots,\ l_n^\mathsf{u}\rangle, \epsilon, \sigma\rangle & = & M_{\langle l_1^\mathsf{u},\ \ldots,\ l_n^\mathsf{u}\rangle}\langle \epsilon,\ \sigma\rangle \ \mathsf{where}\ n \geq 1 \\
\mathsf{mon}\langle \mu, \epsilon, \sigma\rangle & = & (\textit{evaluated at instantiation})
\end{array}
$$

The mon operation derives a monadic result type and effect. For polymorphic effect types, mon cannot be computed until instantiation. We therefore keep this type unevaluated until instantiation time. As such, it is really a *dependent type* (or a type parametrized over types). In our case, this is a benign extension to $\lambda^{\kappa\mathsf{u}}$ since $\lambda^{\kappa\mathsf{u}}$ is explicitly typed. After instantiation, the type argument is not polymorphic, thus mon will return a concrete type.

tdict$\langle\rangle$ is a dependent type used to get the type of a polymorphic dictionary:

$$
\begin{array}{lcl}
\mathsf{tdict}\langle\langle\rangle\rangle & = & \mathit{tdict}\langle M_{\mathit{uid}}\rangle \\
\mathsf{tdict}\langle\langle l_1^\mathsf{u},\ \ldots,\ l_n^\mathsf{u}\rangle\rangle & = & \mathit{tdict}\langle M_{\langle l_1^\mathsf{u},\ \ldots,\ l_n^\mathsf{u}\rangle}\rangle \ \mathsf{where}\ n \geq 1 \\
\mathsf{tdict}\langle \mu\rangle & = & (\text{evaluated at instantiation})
\end{array}
$$

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

Monadic Abstractions. Figure 5 defines two syntactic abstractions $\mathsf{lift}^{\upsilon_t}_{\upsilon_s}(\epsilon, e)$ lifts the expression e from the source υ_s to the target υ_t computed effect. If the computed effects are the same e is returned. If effects are different $\upsilon_s \neq \upsilon_t$ and the source effect is empty, the lifting is performed via a call to the *unit* field of the dictionary of the target effect $\mathit{dict}_{\upsilon_t}$. Otherwise, the lifting is performed via the morphism $\upsilon_s \rhd \upsilon_t$. Note that the monadic *unit* and the morphism operators are effect polymorphic thus *lift* is also parametric on an effect $\epsilon :: \mathsf{e}$ that is used to instantiate the effect variable of *unit* and $\rhd$. Furthermore, lift fails if the morphism $\upsilon_s \rhd \upsilon_t$ is not defined.

$\mathsf{bind}^{\upsilon}_{\upsilon_x}(\epsilon, e_x, x, e)$ 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 both expressions are lifted to the effect $\upsilon_x \oplus \upsilon$ and binding is performed via a call in the *bind* field of the dictionary of the target effect $\mathit{dict}_{\upsilon_x \oplus \upsilon}$.

As an optimization, if $\upsilon = \langle\rangle$ our system uses the monadic *map* instead of lifting ϵ to υ_x and then using *bind*. As in *lift* the combined effect ϵ is used to

instantiate the effect variable of the monadic operators. This optimization is similar to the ones used to avoid unnecessary "administrative" redexes, which customary CPS-transform algorithms go to great lengths to avoid [22].

3.3.1. Monadic Translation Finally, we define the translation relation $e \rightsquigarrow_\epsilon e' \mid \upsilon$ as shown in Figure 6, 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 unchanged. Rule (Lam) states that when translating $\lambda^\epsilon x : \tau.\, e$ the type τ 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, and the Koka type system guarantees that type abstraction only happens over total expressions. 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, the rule (TApp-E) that translates application of the effect ϵ' 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 set of user-defined effects, say $\langle\mathsf{amb}\rangle$, then the rule directly applies the appropriate dictionary $dict_{amb}$, i.e. the dictionary value that Koka created from the amb effect definition. If the computed effect $\overline{\epsilon'}$ is an effect variable μ, then the rule applies the appropriate variable dictionary $dict_\mu$, i.e. the variable abstracted by a rule (TLam-E) lower in the translation tree. By the way we defined computed effects, the final case is the computed effect $\overline{\epsilon'}$ to be the empty effect $\langle\rangle$, and then the identity dictionary $dict_{uid}$ is applied. This is because in the computed effects world the total effect $\langle\rangle$ *is* the identity effect *uid*. 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 υ_3.

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.

Finally, the rule (Val) translates val-binding $\mathtt{val}\ x = e_1; e_2$ by binding e_1 to x in e_2. Similarly, the rule (If) translates $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ by first binding e_1 to a fresh variable y, 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 [15] 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 $\vdash e : \sigma \mid \epsilon$.

Here, the relation $\Gamma \vdash k : \sigma \mid \epsilon \rightsquigarrow e$ is the type inference relation defined in [15] 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 technical report [27] we give a proof of soundness for a more general Theorem (*General Type Preservation*): if $\vdash e : \sigma$ and $e \rightsquigarrow_\epsilon e' \mid v$, then $\vdash e' : \mathsf{mon}\langle v, \epsilon, [\![\sigma]\!]\rangle$. Theorem 1 follows as a direct implication for a computed effect $v = \langle\rangle$. □

4. Related work

Many *effect typing* disciplines have been proposed that study how to delimit the scope of effects. Early work is by Gifford and Lucassen [9, 16] which was later extended by Talpin [26] and others [18, 25]. 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. [18] also requires the effects to form a complete lattice with meets and joins. Wadler and Thiemann [29] show the close connection between monads [17, 28] and the effect typing disciplines.

Java contains a simple effect system where each method is labeled with the exceptions it might raise [11]. A system for finding uncaught exceptions was developed for ML by Pessaux et al. [19]. A more powerful system for tracking effects was developed by Benton [3] who also studies the semantics of such effect systems [4]. Recent work on effects in Scala [23] shows how restricted polymorphic effect types can be used to track effects for many programs in practice.

Our current work relies heavily on a *type directed* monadic translation. This was also described in the context of ML by Swamy et al. [24], where we also showed how to combine multiple monads using monad morphisms. However, Koka uses row-polymorphism to do the typing, while [24] uses subtyping. A problem with subtyping is that it leads to too complicated types.

A similar approach to [24] is used by Rompf et al. [22] 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 [6, 7] 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. In a later work Filinksi [8] proposes a typed calculus where monads are used to give semantics to effects. This proposal has many similarities with our current work, but does not explore effect inference and polymorphism, which both are features crucial for a usable effect system.

Algebraic effect handlers described by Plotkin et al. [20] 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 [1] and a type checking system [2]. 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.

References

1. Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, 1203.1539, 2012. URL http://arxiv.org/abs/1203.1539.
2. Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10 (4), 2014.
3. Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI*, 2007. doi:10.1145/1190315.1190320.
4. 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.
5. Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, 1997.
6. Andrzej Filinski. Representing monads. In *POPL*, 1994.
7. Andrzej Filinski. Controlling effects. Technical report, 1996.
8. Andrzej Filinski. Monads in action. 2010.
9. David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, 1986. doi:10.1145/319838.319848.
10. Jean-Yves Girard. The System F of variable types, Fifteen years later. *TCS*, 1986.
11. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. 1996.
12. Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. In *MSFP*, 2014.
13. Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*. 2009. doi:10.1007/978-3-642-03034-5_17.
14. Daan Leijen. Koka: Programming with row-polymorphic effect types. Technical Report MSR-TR-2013-79, Microsoft Research, 2013.
15. Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP*, 2014. doi:10.4204/EPTCS.153.8.
16. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.
17. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 1991.
18. Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *LOMAPS*, 1997.
19. François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *POPL*, 1999. doi:10.1145/292540.292565.
20. Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP*, 2009.
21. John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, 1974.
22. Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. ICFP, 2009.
23. Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, 2012. doi:10.1007/978-3-642-31057-7_13.
24. Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011. doi:10.1145/2034773.2034778.
25. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 1994. doi:10.1006/inco.1994.1046.
26. 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.
27. Niki Vazou and Daan Leijen. From monads to effects and back – technical report. Technical report, July 2015. URL http://goto.ucsd.edu/~nvazou/padl16/techrep.pdf.
28. Philip Wadler. The essence of functional programming. POPL, 1992.
29. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *TOLC*, 2003. doi:10.1145/601775.601776.