A type system for algebraic effects and handlers with dynamic instances

Albert ten Napel

Contents

Chapter 1

Introduction

Side-effects are ubiquitous in programming. Examples include mutable state, exceptions, nondeterminism, and user input. Side-effects often make functions hard to understand, test and debug. This is because every invocation of the same function with the same arguments may yield different results. Furthermore side-effectful programs can also be difficult to optimize, since the compiler does not have much freedom in rearranging parts of the program.

Any function that includes such side-effects is called *impure*, while functions whose only effect is computing a result are called *pure*. Pure functions on the other hand do not rely on any global state and thus can be reasoned about in isolation of the rest of the program. Every time a pure function is called with the same input, it will return the same output. This means those functions are easier to understand, test, and debug.

There has been a lot of work on programming languages that allow more control over the pure and impure parts of a program. Examples include Haskell [12], Eff [1], Koka [5], and Links [3]. These languages, in one way or another, give the programmer more control over which parts of their program are pure and which parts are impure. By factoring out the pure parts from the impure parts, we can still gain the benefits of pure functions for many parts of our programs. In addition these languages allow to keep track of which effects exactly are used by which function. They also allow some side-effects to be encapsulated, meaning that the use of a particular side-effect

can be completely hidden such that the function still appears to be pure to the outside world.

Type systems play an essential role in enforcing the distinction between pure and impure code. By extending type systems to also show which effects a function may use, we can statically enforce which functions are pure and which are not. This gives insight to the user to what a function may do when called, and also allows a compiler to do more interesting optimizations. For example pure function calls may be reordered in any way that the compiler sees fit, while impure function calls may not, since the effects may interact. These *effect systems* can have different levels of granularity. For example one system could only keep track of a single bit per function, whether the function is impure or not. More fine-grained systems are also possible, where each function is annotated with a set of effects that is used, where the set of possible effects is defined by the language. For example in Koka a function which prints something to the console may be given the type:

```
string -> <console> ()
```

Where CONSOLE shows the use of the console. User-defined effects are also supported in languages such as Koka and Eff. Users, in those systems, can define the effects with which the functions are annotated.

Algebraic effects and handlers [13] are an approach to programming with side-effects that has many of the desirable properties previously described. Algebraic effects provide a way to factor out the pure parts, the operation calls, from the impure parts. Users can define effects and easily use them in functions, with different effects composing without any extra effort. Algebraic effects also easily admit typing, with different type-and-effect already proposed [2, 4, 3]. Each effect is defined as a set of operations, for example nondeterminism can be represented by an operation which takes to values and chooses one. Similarly, state can be defined as two operations, get and put, where get is meant to return the current value of the state and put is meant to change this value. Functions are tagged by the set of effects they may use. These operations can then be called anywhere in a function. Handlers take a program that calls operations and for each operation call defines how to proceed. For example the following piece of code defines an effect called *State* which simulates a single mutable state cell. The function `postInc` increments the current value in the state cell and returns the

previous value.

```
effect State {
  get : () -> Int
  put : Int -> ()
}

postInc : Int!{State}
postInc =
  x <- get ();
  put (x + 1);
  return x
```

While algebraic effects and handlers have many of the desirable properties we would like, they are is unable to express multiple mutable state cells. In the previous example it can be seen that `postInc` does not refer to any variables, but instead can only manipulate the mutable state using the get and put operations. In Haskell the the so-called "ST monad" [14] can be used to safely implement multiple mutable state cells in such a way that stateful computations can be encapsulated and that the references to the mutable objects are not leaked outside of the function. A feature called dynamic instances was introduced by the Eff programming language [1]. With dynamic instances multiple different instances of the same effect can be dynamically created. Using this multiple state cells can be implemented. Unfortunately there is no type-and-effect for dynamic instances, using them can results in runtime errors when instances do not have an associated handler.

In this thesis we define a calculus based on algebraic effects and handlers which allows for the definition of side-effects such as local references, local exceptions, and the dynamic opening of channels. Using this system we can implement a system similar to the "ST monad" in Haskell. This system gives full control of which parts of a program are pure and impure. Functions also compose easily, irrelevant of which side-effects they use. Using a type-and-effect system every function keeps track of which effects it may use. We also statically ensure that side-effects are encapsulated. We give examples of programs using these side-effects in our system and show how to implement local mutable references in this system. We give a formal description of the syntax, typing rules and semantics of the system.

Contributions

- **Language.** We define a language based on algebraic effects and handlers that can handle a form of dynamic effect instances.
- **Mutable references.** We give examples in our language that would be difficult or impossible to express with just algebraic effects.
- **Operational semantics and type system.** We define a core calculus of our language together with a small-step operational semantics and a type system.
- **Type soundness.** We prove type soundness of our type system with respect to the operational semantics via type preservation and progress for the core calculus.

Thesis structure

The thesis is structured as follows. Chapter 2 gives an introduction to algebraic effects and handlers, and static and dynamic instances. Chapter 3 gives an introduction to our proposed language. Chapter 4 gives formal definitions of systems with algebraic effecst and handlers, and static instances. Chapter 5 gives a formal account of X. Chapter 6 discusses related work. Chapter 7 concludes the thesis and discusses future work.

Chapter 2

An introduction to algebraic effects and handlers

Side-effects are an essential part of a programming language. Without side-effects the program would have no way to print a result to the screen, ask for user input or change global state. We consider a function pure if it does not perform any side-effects and unpure if it does. A pure function always gives the same result for the same inputs. A pure function can be much easier to reason about than an unpure one because you know that it won't do anything else but compute, it won't have any hidden inputs or outputs. Because of this property testing pure functions is also easier, we can just give dummy inputs to the functions and observe the output. As already said programs without side-effects are useless, we would not be able to actually observe the result of a function call without side-effects such as printing to the screen. So we would the benefits of pure functions but still have side-effects. We could give up and simply add some form of side-effects to our language but that would immediately make our function impure, since any function might perform side-effects. This would make us lose the benefits of pure functions.

Algebraic effects and handlers are a structured way to introduce side-effects to a programming language. The basic idea is that side-effects can be described by sets of operations, called the interface of the effect. Operations from different effects can then be called in a program. These operations will stay abstract though, they will not actually do anything. Instead, similar to

exceptions where exceptions can be thrown and caught, operations can be "caught" by handlers. Different from exceptions however the handler also has access to a continuation which can be used to continue the computation at the point where the operation was called.

In this chapter we will introduce algebraic effects and handlers through examples. Starting with simple algebraic effects and handlers (§2.1). After we will continue with static instances (§2.2) which allows for multiple static instances of the same effect to be used in a program. We end with dynamic instances (§2.3) which allows for the dynamic creation of effect instances. The examples are written in a statically typed functional programming language with algebraic effects and handlers with syntax reminiscent to Haskell but semantically more similar to Koka[5].

2.1 Algebraic effects and handlers

We will start with the familiar exceptions. We define an `Exc` effect interface with a single operation `throw`.

```
effect Exc {
  throw : String -> Void
}
```

For each operation in an effect interface we specify a parameter type (on the left of the arrow) and a return type (on the right of the arrow). The parameter type is the type of a value that is given when the operation is called and that the handler also has access too. The return type is the type of a value that has to be given to the continuation in the handler, this will be shown later. This return value is received at the point where the operation was called. In the case of `Exc` we take `String` as the parameter type, this is the error message of the exception. An exception indicates that something went wrong and that we cannot continue in the program. This means we do not want the program to continue at the point where the exception was thrown, which is the point where the `throw` operation was called. So we do not want to be able to call the continuation with any value. To achieve this we specify `Void` as the return type of `throw`. This is a type with no values at all, which means that the programmer will never be able to conjure up a

suitable value when a value of type `Void` is requested. By using `Void` as the return type we can ensure that the continuation cannot be called and so that the program will not continue at the point where `throw` was called. To make the code more readable we assume `Void` implicitly coerces to any other type.

We can now write functions that use the `Exc` effect. For example the following function `safeDiv` which will throw an error if the right argument is 0. We assume here that `Void` is equal to any type.

```
safeDiv : Int -> Int -> Int!{Exc}
safeDiv a b =
  if b == 0 then
    throw "division by zero!"
  else
    return a / b
```

We can call this function like any other function, but no computation will actually be performed. The effect will remain abstract, we still need to give them a semantics.

```
result : Int!{Exc}
result = safeDiv 10 2
```

In order to actually "run" the effect we will need to handle the operations of that effect. For example, for `Exc` we can write a handler that returns 0 as a default value if an exception is thrown.

```
result : Int
result = handle (safeDiv 10 0) {
  throw err k -> return 0
  return v -> return v
} -- results in 0
```

For each operation we write a corresponding case in the handler, where we have access to the argument given at operation call and a continuation, which expects a value of the return type of the operation. There is also a case for values `return`, which gets as an argument the final value of a computation and has the opportunity to modify this value or to do some final computation. In this case we simply ignore the continuation and exit the computation early with a 0, we also return any values without modification.

We can give multiple ways of handling the same effect. For example we can also handle the `Exc` effect by capturing the failure or success in a sum type `Either`.

```
data Either a b = Left a | Right b

result : Either String Int
result = handle (safeDiv 10 0) {
  throw err k -> return (Left err)
  return v -> return (Right v)
} -- results in (Left "division by zero!")
```

Here we return early with `Left err` if an error is thrown, otherwise we wrap the resulting value using the `Right` constructor.

Another effect we might be interested in is non-determinism. To model this we define the `Flip` effect interface which has a single operation `flip`, which returns a boolean when called with the unit value.

```
effect Flip {
  flip : () -> Bool
}
```

Using the `flip` operation and if-expression we can write non-deterministic computations that can be seen as computation trees where `flip` branches the tree off into two subtrees. The following program `choose123` non-deterministically returns either a 1, 2 or 3.

```
choose123 : Bool!{Flip}
choose123 =
  b1 <- flip ();
  if b1 then
    return 1
  else
    b2 <- flip ();
    if b2 then
      return 2
    else
      return 3
```

Here the syntax `(x <- c1; c2)` sequences the computations `c1` and `c2` by

first performing `c1` and then performing `c2`, where the return value of `c1` can accessed in `x`.

Again `choose123` does not actually perform any computation when called, because we have yet to give it a semantics. We could always return `True` when a `flip` operation is called, in the case of `choose123` this will result in the first branch being picked returning 1 as the answer.

```
result : Int
result = handle (choose123) {
  flip () k -> k True
  return v -> return v
} -- returns 1
```

Another handler could try all branches returning the greatest integer of all possibilities.

```
maxresult : Int
maxresult = handle (choose123) {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return (max vtrue vfalse)
  return v -> return v
} -- returns 3
```

Here we first call the continuation `k` with `True` and then with `False`. The we return the maximum between those results.

We could even collect the values from all branches by returning a list.

```
allvalues : List Int
allvalues = handle (choose123) {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return vtrue ++ vfalse
  return v -> return [v]
} -- returns [1, 2, 3]
```

Again we call the continuation `k` twice, but we append the two results in-

stead. For the `return` base case we simply wrap the value in a singleton list.

Algebraic effects have the nice property that they combine easily. For example by combining the `Exc` and `Flip` we can implement backtracking, where we choose the first non-failing branch from a computation. For example we can write a function which returns all even sums of the numbers 1 to 3 by reusing `choose123`.

```
evensums123 : Int!{Flip, Exc}
evensums123 =
  n1 <- choose123;
  n2 <- choose123;
  sum <- return (n1 + n2);
  if sum % 2 == 0 then
    return sum
  else
    throw "not even!"
```

We implement backtracking in `backtrack` by handling both the `flip` and `throw` operations. For `flip` and the `return` case we do the same as in `allvalues`, calling the continuation `k` with both `True` and `False` and appending the results together. For `throw` we ignore the error message and continuation and exit early with the empty list, this means that branches that results in a failure will not actually return any values.

```
backtrack : List Int
backtrack () = handle (handle (evensums123) {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return vtrue ++ vfalse
  return v -> return [v]
}) {
  throw msg k -> return []
  return v -> return v
} -- returns [2, 4, 4, 6]
```

We can also handle the effects independently of each other. For example we could implement a partial version of `backtrack` that only handles the `Flip` effect. Any operation that is not in the handler is just passed through.

```
partlybacktrack : (List Int)!{Exc}
partlybacktrack = handle (evensums123) {
  flip () k ->
    vtrue <- k True;
    vfalse <- k False;
    return vtrue ++ vfalse
  return v -> return [v]
}
```

Now we can factor out the `throw` handler into its own function.

```
fullbacktrack : List Int
fullbacktrack = handle (partlybacktrack) {
  throw msg k -> return []
  return v -> return v
} -- returns [2, 4, 4, 6]
```

Algebraic effects always commute, meaning the effects can be handled in any order. In the backtracking example the order of the handlers does not actually matter, but in general different orders could have different results.

Lastly we introduce the `State` effect, which allows us to implement local mutable state. We restrict ourselves to a state that consists of a single integer value, but in a language with parametric polymorphism a more general state effect could be written.

```
effect State {
  get : () -> Int
  put : Int -> ()
}
```

Our state effect has two operations, `get` and `put`. The `get` operation allows us to retrieve a value from the state and with the `put` operation we can change the value in the state.

We can now implement the familiar "post increment" operation as seen in the C programming language. This function retrieves the current value of the state, increments it by 1 and returns the previously retrieved value.

```
postInc : Int!{State}
postInc =
```

```
  x <- get ();
  put (x + 1);
  return x
```

To implement the semantics of the `State` effect we use parameter-passing similar to how the State monad is implemented in Haskell. We will abstract the implementation of the state handler in a function `runState`.

```
runState : Int!{State} -> (Int -> (Int, Int))
runState comp = handle (comp) {
  get () k -> return (\s -> (f <- k s; return f s))
  put v k -> return (\s -> (f <- k (); return f v))
  return v -> return (\s -> return (s, v))
}
```

`runState` takes a computation that returns an integer and may use the `State` effect, and returns a function that takes the initial value of the state and returns a tuple of the final state and the return value of the computation. Let us take a look at the `return` case first, here we return a function that takes a state value and returns a tuple of this state and the return value. For the `get` case we return a function that takes a state value and runs the continuation `k` with this value, giving access to the state at the point where the `get` operation was called. From this continuation we get back another function, which we call with the current state, continuing the computation without changing the state. The `put` case is similar to the `get` but we call the continuation with the unit value and we continue the computation by calling `f` with the value giving with the `put` operation call.

Using state now is as simple as calling `runState`.

```
stateResult : (Int, Int)
stateResult =
  f <- runState postInc; -- returns a function taking the initial state
  f 42 -- post-increments 42 returning (43, 42)
```

Using the state effect we can implement imperative algorithms such as summing a range of numbers. We first implement a recursive function `sumRangeRec` which uses `State` to keep a running sum. After we define `sumRange` which calls `sumRangeRec` and runs the `State` effect with 0 as the initial value.

```
sumRangeRec : Int -> Int -> Int!{State}
```

```
sumRangeRec a b =
  if a > b then
    (_, result) <- get ();
    return result
  else
    x <- get ();
    put (x + a);
    sumRangeRec (a + 1) b

sumRange : Int -> Int -> Int
sumRange a b =
  f <- runState (sumRangeRec a b);
  f 0 -- initial sum value is 0
```

2.2 Static instances

Static instances extend algebraic effects by allowing multiple instances of the same effect to co-exist. These instances be handled independently of each other. Operations in such a system are always called on a specific instance and handlers also have to note instance they are handling. We will write operation calls as `inst##op(v)` where `inst` is the instance. Handlers are modified to take an instance parameter as follows `handle##inst(comp) { ... }`.

As an example let us take another look at the `safeDiv` function.

```
safeDiv : Int -> Int -> Int!{Exc}
safeDiv a b =
  if b == 0 then
    throw "division by zero!"
  else
    return a / b
```

We can rewrite this to use static instances by declaring an instance of `Exc` called `divByZero` and calling the `throw` operation on this instance. Note that in the we now state the instance used instead of the effect, since multiple instances of the same effect could be used and we would like to know which instances exactly.

```
instance Exc divByZero

safeDiv : Int -> Int -> Int!{divByZero}
safeDiv a b =
  if b == 0 then
    divByZero#throw "division by zero!"
  else
    return a / b
```

Imagine we wanted to also throw an exception in the case that the divisor was negative. Using instances we can easily declare another `Exc` instance, let us call it `negativeDivisor`, and use it in our function. We also have to modify the type to mention the use of `negativeDivisor`.

```
instance Exc divByZero
instance Exc negativeDivisor

safeDivPositive : Int -> Int -> Int!{divByZero, negativeDivisor}
safeDivPositive a b =
  if b == 0 then
    divByZero#throw "division by zero!"
  else if b < 0 then
    negativeDivisor#throw "negative divisor!"
  else
    return a / b
```

We can now see from the type what kind of exceptions are used in the function. We can also handle the exceptions independently. For example we could handle `divByZero` by defaulting to `0`, but leave `negativeDivisor` unhandled.

```
defaultTo0 : Int!{divByZero, negativeDivisor} -> Int!{negativeDivisor}
defaultTo0 c =
  handle#divByZero (c) {
    throw msg -> return 0
    return v -> return v
  }
```

2.3 Dynamic instances

Having to predeclare every instance we are going to use is very inconvenient, especially when we have effects such as reference cells or communication channels. The global namespace would be littered with all references and channels the program would ever use. Furthermore we do not always know how many references we need. Take for example a function which creates a list of reference cells giving a length l. We do not know statically what the length of the list will be and so we do not know ahead how many instances we have to declare. Furthermore because all the instances would be predeclared some information about the implementation of a function would be leaked to the global namespace. This means it is impossible to fully encapsulate the use of an effect when using static instances.

Dynamic instances improve on static instances by allowing instances to be created dynamically. Instances become first-class values, they can be assigned to variables and passed to functions just like any other value. We use `new E` to create a new instance of the `E` effect. The actual implementation of the function can stay exactly the same, as can the handler `defaultTo0`. We can translate the previous example to use dynamic instances by defining the `divByZero` and `negativeDivisor` as top-level variables and assigning newly created instances to them. We omit type annotation, since there does not exist any type system that can type all usages of dynamic instances.

```
divByZero = new Exc
negativeDivisor = new Exc

safeDivPositive a b =
  if b == 0 then
    divByZero#throw "division by zero!"
  else if b < 0 then
    negativeDivisor#throw "negative divisor!"
  else
    return a / b

defaultTo0 c =
  handle#divByZero (c) {
    throw msg -> return 0
```

```
    return v -> return v
  }
```

Using locally created instances we can emulate variables as they appear in imperative languages more easily. We can implement the factorial function in an imperative style using a locally created `State` instance. The `factorial` function computes the factorial of the paramter `n` by creating a new `State` instance named `ref` and calling the helper function `factorialLoop` with `ref` and `n`. The base case of `factorialLoop` retrieves the current value from `ref` and returns it. In the recursive case of `factorialLoop` the value in `ref` is modified by multiplying it by `n` and then we continue by recursing with `n - 1`. The call to `factorialLoop` in `factorial` is wrapped in the `State` handler explained earlier, chosing `1` as the initial value of `ref`. `factorial` thus computes the factorial of a number by using a locally created instance, but the use of this instance or the `State` effect in general never escapes the function, it is completely encapsulated.

```
factorialLoop ref n =
  if n == 0 then
    ref#get ()
  else
    x <- ref#get();
    ref#put (x * n);
    factorialLoop ref (n - 1)

factorial n =
  ref <- new State;
  statefn <- handle#r (factorialLoop ref n) {
    get () k -> return (\s -> (f <- k s; return f s))
    put v k -> return (\s -> (f <- k (); return f v))
    return v -> return (\s -> return v)
  };
  statefn 1 -- use 1 as the initial value of ref
```

Next we will implement references more generally similar to the ones available in Standard ML[11], in our case specialized to `Int`. In the previous example we see a pattern of creating a `State` instance and then calling some function with it wrapped with a handler. This is the pattern we want to use when implementing references. To implement this pattern more generally this we

first introduce a new effect named `Heap`. `Heap` has one operation called `ref` which takes an initial value `Int` and returns a `State` instance. `Heap` can be seen as a collection of references. We then define a handler `runRefs` which takes a `Heap` instance and a computation, and creates `State` instances for every use of `ref`. After we call the continuation with the newly created instance and wrap this call in the usual `State` handler, giving the argument of `ref` as the initial value.

```
effect Heap {
  ref : Int -> Inst State
}

runRefs inst c =
  handle#inst (c) {
    ref v k ->
      r <- new State;
      statefn <- handle#r (k r) {
        get () k -> return (\s -> (f <- k s; return f s))
        put v k -> return (\s -> (f <- k (); return f v))
        return v -> return (\s -> return v)
      };
      statefn v
    return v -> return v
  }
```

By calling `runRefs` at the top-level we will have the same semantics for references as Standard ML. In the following example we create two references and swap their values using a `swap` function. First `main` creates a new `Heap` instance `heap` and then calls `runRefs` with this instance. The computation given to `runRefs` is the function `program` called with `heap`.

```
swap r1 r2 =
  x <- r1#get ();
  y <- r2#get ();
  r1#put(y);
  r2#put(x)

program heap =
  r1 <- heap#ref 1;
```

```
  r2 <- heap#ref 2;
  swap r1 r2;
  x <- r1#get ();
  y <- r2#get ();
  return (x, y)

main =
  heap <- new Heap;
  runRefs heap (program heap) -- returns (2, 1)
```

In the Haskell programming language the ST monad?? can be used to implement algorithms that internally use mutable state. The type system, using the `runST` function, will make sure that the mutable state does not leak outside of the function. For example the following function `fibST` implements the Fibonacci function in constant space by creating two mutable references.

```
fibST :: Integer -> Integer
fibST n =
    if n < 2 then
      n
    else runST $ do
        x <- newSTRef 0
        y <- newSTRef 1
        fibST' n x y

    where fibST' 0 x _ = readSTRef x
          fibST' n x y = do
              x' <- readSTRef x
              y' <- readSTRef y
              writeSTRef x y'
              writeSTRef y $! x' + y'
              fibST' (n - 1) x y
```

Using dynamic instances we can implement the same algorithm, named `fib` below. Our `fib` takes a parameter `n` and returns the `n`th Fibonacci number. First we check if `n` is smaller than 2, in which case we can return `n` as the result, since nth Fibonacci number is n, if $n < 2$. Else we create a new `Heap` instance named `heap` and use the `runRefs` function defined earlier to

run a computation on this heap. We create two `State` instances on `heap`, `x` and `y` initialized with `0` and `1` respectively and call the auxillary function `fibRec` with `n` and the two instances `x` and `y`. `fibRec` implements the actual algorithm. It works by (recursively) looping on `n`, subtracting by `1` each recursive call. `x` and `y` store the current and next Fibonacci respectively and each loop they are moved one Fibonacci number to the right. When `n` is `0` we know `x` contains the `n`th (for the initial value of `n`) Fibonacci number and we can just get the current value from `x` and return it. Even though this algorithm uses the `Heap` and `State` effects, their uses are completely encapsulated by the `fib` function. The `fib` function does not leak the fact that it's using those effects to implement the algorithm.

```
fib n =
  if n < 2 then
    n
  else
    heap <- new Heap;
    runRefs heap (
      x <- heap#ref 0;
      y <- heap#ref 1;
      fibRec n x y
    )

fibRec n x y =
  if n == 0 then
    x#get ()
  else
    x' <- x#get ();
    y' <- y#get ();
    x#put(y');
    y#put(x' + y');
    fibRec (n - 1) x y
```

Dynamic instances have one big problem though: they are too dynamic. Similar to how in general it is undecidable to know whether a reference has escaped its scope, it is also not possible to know whether an instance has a handler associated with it. This makes it hard to think of a type system for dynamic instances which ensures that there are no unhandled operations. Earlier versions of the Eff programming language[1] had dynamic instances

but its type system underapproximated the uses of dynamic instances which meant you could still get a runtime error if any operation calls were left unhandled.

Chapter 3

Introduction to X

In Chapter 2.3 we saw how dynamic instances allow us to implement mutable references in a system with algebraic effects. This system is untyped however, meaning that you can get runtime errors if an operation is unhandled. This can happen if an operation is called on a dynamic instance outside of a handler. In this chapter we introduce X, which combines algebraic effects and dynamic instances while still remaining typeable. In order to achieve this we introduce the notion of an *effect scope*. An effect scope groups together instances. When creating an instance we give an effect scope to create the instance in. Every instance belongs to a specific effect scope. Different from the system with dynamic instances from Chapter 2.3, we always have to specify a handler when creating an instance. This to make sure every instance has a handler associated with it. Performing effects is done with a new runscope construct, similar to how the handle construct performed effects in Chapter 2. runscope creates a fresh scope and makes it available for use in a given computation. After all the instances created on this new scope will actually be instantiated and all the operations called on these instances will be handled. In order to allow computations to be polymorphic over effect scopes we also introduce *effect scope polymorpishm* together with *effect scope abstraction* and *effect scope application*.

We start with explaining all the novel concepts in Section 3.1, using the example of mutable references. Then we will show how mutable vectors can be defined, followed by an implementation of a list shuffling algorithm in

Section 3.2.

We build on the language used in Chapter 2.1. We use a language reminiscent of Haskell with algebraic data types and pattern matching. Type constructors and effect names are uppercase while type variables are lowercase.

3.1 Effects, effect scopes and instances

In Figure 3.1 we give an example containing all the novel constructs of X.

3.1.1 Effects

To start off we define a `State` effect specialized to `Int`s. `State` is meant to represent a mutable reference to a single value of type `Int`. This definition is exactly the same as the `State` effect definition in Chapter 2.1, in the basic algebraic effects system. With the `effect` keyword we declare a new effect called `State` with two operations: `get` and `put`. For each operation we give parameter and return types. For `get` we give the unit type `()` as the parameter type. As the return type we give `Int`, meaning that calling the `get` operation will return a integer value. For the `put` operation it is the other way around. As the parameter type we have `Int`, meaning that `put` requires an integer value when called, and the return type is `()`, calling `put` will give back the unit value.

3.1.2 Effect instances

In Figure 3.1 the function `ref` creates a new *effect instance* of the `State` effect. We can create a fresh instance of an effect using the `new` keyword. The construct `new State@s { ... }` can be read as "Create a fresh `State` instance in the effect scope `s`". Here we have to give a specific scope `s` to create the instance on. The instance can only be used within this given scope. The newly-created instance is available in the body of the `new` construct.

Figure 3.1: Example of all the novel constructs

```
effect State {
  get : () -> Int
  put : Int -> ()
}

ref : forall s. Int -> (Inst s State)!{s}
ref [s] v =
  new State@s {
    get () k -> \st -> k st st
    put st' k -> \st -> k () st'
    return x -> \st -> return x
    finally f -> f v
  } as x in return x

postInc : forall s. Inst s State -> Int!{s}
postInc [s] inst =
  x <- inst#get();
  inst#put(x + 1);
  return x

result : Int!{}
result =
  runscope(s1 ->
    r1 <- ref [s1] 10;
    runscope(s2 ->
      r2 <- ref [s2] 20;
      x <- postInc [s2] r2;
      r1#put(x);
      return x);
    y <- r1#get ();
    return y) -- result is 20
```

When creating an instance we have to give a *handler*. The handler specifies what should happen when the operations are called. The handler is defined within curly braces and consists of a case for each operation of the effect, plus a `return` case and a `finally` case. The handler given in the example is the same as the handler given in Chapter 2.1, implementing a single mutable reference. In addition we also have to define a `finally` case. In there we can perform an extra computation after the handler is performed. In the case of the handler given this is necessary because the return type of the handler is `Int -> T` for some return type `T`. The handler transforms a computation in to a function expected the initial value for the mutable reference. Using the `finally` case we can call this function and get back the return value of the computation (of type `T`. In the example we simply call the function `f` with the argument `v` given to `ref`.

We can now understand the type of `ref`:

```
ref : forall s. Int -> (Inst s State)!{s}
```

We can see from the `forall s.` that `ref` is polymorphic over scopes. That means that this function works for any scope `s`.

The type variable `s` here is an *effect scope variable*. An effect scope variable can be seen as the name of a collection of instances that we call an effect scope. Such a scope can contain zero or more instances, where each instance can be of any effect. A scope restricts instances in such a way that they cannot escape that scope and instances from one scope cannot be used in another. This also means that we can never get a runtime error because of an unhandled operation call.

In order to apply `ref` we have to give an explicit scope `s` using the syntax `ref [s]`. We call this *effect scope application*. In the definition we show that this function has a scope parameter using angle brackets `[s]`. We call this *effect scope abstraction*. The second parameter is a value of type `Int` which we call `v`. This is the initial value that we want our mutable reference to have. The return type is `Inst s State`. This is an effect instance of the `State` effect in the scope `s`. From this type we can see that `ref` gives back an instance on the given scope `s`. The effect annotation of `ref` is `!{s}`, which shows that we actually perform effects in the scope `s`. We can see from the function implementation that the only effect we perform is the creation of

an instance on `s`.

Using `ref` we can fully emulate multiple mutable references. We have the added guarantee that the references will not escape their effect scope, they will not escape their corresponding `runscope`. Adding parametric polymorphism to the effects to give `State t` for any type `t` will enable us to emulate references of any type. With references of different types coexisting. This is very similar to how the ST monad works in Haskell [14]. Using mutable references have interesting applications such as meta variables in unification algorithms and typed logic variables [15].

Looking at the type of creating mutable references using the ST monad in Haskell (`newSTRef`), we can see that the type of `ref` is very similar (we explicitly wrote down the quantification and specialized `newSTRef` to `Int`):

```
ref : forall s. Int -> (Inst s State)!{s}
newSTRef :: forall s. Int -> ST s (STRef s Int)
```

Here `ST s` serves the same purpose as `!{s}` in our system. `STRef s Int` is the type of a mutable reference in the ST monad in Haskell. `s` is some "state thread", the purpose if this type variable is to statically ensure that references do not escape their scope. This is exactly like the `s` type variable in our system, but we generalize "state threads" to effect scopes, where any algebraic effect may be performed.

3.1.3 Using effect instances

In Figure 3.1 the function `postInc` shows how an effect instance can actually be used:

```
postInc : forall s. Inst s State -> Int!{s}
postInc [s] inst =
  x <- inst#get();
  inst#put(x + 1);
  return x
```

We can see from the type that this function is polymorphic over some effect scope `s`. It expects some scope `s` and some `State` instance on `s` as its arguments. It returns an integer value and may perform some effects on `s`

(`Int!{s}`. The type of `postInc` can be read as "For any scope `s`, given a `State` instance in `s`, return a value of type `Int` possibly by calling operations on instances in `s`".

Effects can be used by calling operations. Operations are always called on an effect instance. Without an instance we are unable to perform operations. In the case of `postInc` we get an instance as an argument to the function. Operation can be called on an instance using the syntax `instance\#operation(argument)`. We write `instance\#operation()` to mean `instance\#operation(())`, when the unit value `()` is given as the argument. `postInc` implements the traditional "post increment" operation on a mutable reference. In the C language this is written `x++` for some reference `x`. We first call the `get` operation on `inst`. We get back a value of type `Int`, which we name `x`. Then we call `put` on `inst` with the argument `(x + 1)`. Finally we return the previous value of the mutable reference `x`.

We could define functions wrapping the `get` and `put` operations:

```
performGet : forall s. Inst s State -> Int!{s}
performGet [s] inst = inst#get()

performPut : forall s. Inst s State -> Int -> ()!{s}
performPut [s] inst v = inst#put(v)
```

Again we can compare to the corresponding functions in the ST monad in Haskell, `readSTRef` and `writeSTRef`:

```
readSTRef :: forall s. STRef s Int -> ST s Int
writeSTRef :: forall s. STRef s Int -> Int -> ST s ()
```

We can see that the types are very similar, but that we generalize the ST monad to work for any algebraic effect.

3.1.4 Running scopes

The definition `result` shows how the effects in a computation can be performed:

```
result : Int!{}
result =
  runscope(s1 ->
    r1 <- ref [s1] 10;
    ret <- runscope(s2 ->
      r2 <- ref [s2] 20;
      x <- postInc [s2] r2;
      r1#put(x);
      return x);
    y <- r1#get ();
    return y) -- result is 20
```

From the type we can see that `result` is a computation that returns an integer value. We can see from the effect annotation (`!{}`) that `result` does not have any unhandled effects. In the future we will omit writing `!{}` if a computation does not have any unhandled effects.

The `runscope(s' -> ...)` construct provides a new scope, which we named `s1` and `s2` in our case, which can be used in its body. Inside `runscope` we can create and use instances in this new scope. `runscope` will make sure that any instances that are created on its scope will actually be created and that any operation calls on these instances will be handled.

In `result` we use two nested scope. First we create a scope called `s1`. On this scope we call `ref` to create a mutable reference `r1` with `10` as its initial value. Then we create another scope called `s2`. In `s2` we create another mutable reference `r2` with `20` as its initial value. We then call `postInc` on `r2` and store the return value in `x` (20). Then we call `r1##put(x)`, setting `r1` to 20. We then return `x` as the return value of the `s2` scope, storing this value in `ret` in the `s1` scope. At this point the `s2` scope is gone and any effects in it will be handled. The type system will make sure that no instances created in `s2` can escape `s2`.

Note that we also used `r1` inside `s2`. Since `r1` belongs to s1, all the operations called on it will not be handled inside `s2` but these will be *forwarded* instead. This means that these operations will remain unhandled until `s2` is done. Because of this forwarding behaviour we can combine effects from multiple scopes, giving us fine-grained control over where effects may happen.

Continuing in `result` we get the current value of `r1` and return from `s1`. This value is `20` which was set in scope `s2`. After this the scope `s1` is done and any effect in it will be handled. This leaves us with a computation of type `Int!{}` with no remaining effects.

3.2 Mutable vectors

Figure 3.2: Mutable vectors

```
1  -- list of mutable references
2  data Vector s = VNil | VCons (Inst s State) Vector
3
4  -- get the length of a vector
5  vlength : forall s. Vector s -> Int
6  vlength VNil = 0
7  vlength (VCons _ tail) = 1 + (vlength tail)
8
9  -- get the value at the index given as the first argument
10 -- assumes the index is within range of the vector
11 vget : forall s. Int -> Vector s -> Int!{s}
12 vget [s] 0 (VCons h _) = h#get()
13 vget [s] n (VCons _ t) = vget [s] (n - 1) t
14
15 -- set the value at the index given as the first argument
16 -- to the value given as the second argument
17 -- assumes the index is within range of the vector
18 vset : forall s. Int -> Int -> Vector s -> ()!{s}
19 vset [s] 0 v (VCons h _) = h#put(v)
20 vset [s] n v (VCons _ t) = vset [s] (n - 1) v t
```

In the previous section we have defined mutable references using the `ref` function. We will now build on them to define mutable vectors. In Figure 3.2 we define the `Vector` datatype. `Vector` is a list of `State` instances and is indexed by the scope of instances: `s`. We define three functions on `Vector`: `vlength`, `vget`, and `vset`. With `vlength` we can get the length of a

vector. With `vget` we can retrieve a value from a vector by giving an index. We assume the index is within the range of the vector. With `vset` we can set an element of a vector by giving an index and a value. Again we assume the index is within the range of the vector. In order to allow these functions to work for any vector we have to introduce an effect scope variable `s` again. We define both functions by recursion on the index.

As an example application we will write a shuffling algorithm for vectors. This simple algorithm will shuffle a vector by randomly swapping two random elements of the vector and repeating this some amount of times. In Figure 3.3 we show the algorithm. First we define an effect `Rng` in order to abstract out the generation of random numbers. `Rng` has a single operation `rand` which returns a random integer between `0` and `n` given an integer `n`. We define a function `vlength` to get the length of the vector.

We then define the actually shuffling function `shuffleVector`. This function takes two scope variables, `s` and `s'`, for the vector and `Rng` instance respectively. As arguments we take an instance of `Rng`, in order to generate random numbers, an integer, for the amount of times to shuffle, and the vector we want to shuffle. By taking a seperate scope for the `Rng` instance we are more flexible when handling the computation. We can handle the effects on the vector while leaving the `Rng` effects to be handled higher up.

`shuffleVector` proceeds as follows. If the amount of times we want to shuffle is `0` we stop and return the vector. If not then we first get the length of the vector. Then we generate two random numbers, `i` and `j`, between `0` and this length. These two numbers will be the two elements we will swap. We then get the current values at these indeces. And we swap the values at these indeces in the vector. We then recurse, subtracting the amount of times to shuffle by one.

Figure 3.3: Vector shuffling

```
-- random number generation effect
-- the operation `rand` gives back a random integer
-- between 0..n, where n is the argument given (exclusive)
effect Rng {
  rand : Int -> Int
}

-- shuffles a vector given an instance of Rng
-- by swapping two random elements of the vector
-- the second argument to shuffleVector is the amount of times
-- to swap elements
shuffleVector : forall s s'. Inst s' Rng -> Int
  -> Vector s -> ()!{s, s'}
shuffleVector [s] [s'] _ 0 vec = vec
shuffleVector [s] [s'] rng n vec =
  let len = vlength vec;
  i <- rng#rand(len);
  j <- rng#rand(len);
  a <- vget [s] i vec;
  b <- vget [s] j vec;
  vset [s] i b vec;
  vset [s] j a vec;
  shuffleVector [s] [s'] rng (n - 1) vec
```

Using `shuffleVector` we can implement a function to shuffle a list in Figure 3.4. We first define the usual `List` datatype, with `Nil` and `Cons` cases. Then we define two functions `toVector` and `toList` to convert between lists and vectors. `toVector` simply recurses on the list and creates fresh mutable references for each element of the list, initialized with the value of the element. `toList` converts a vector to a list by getting the current values of each reference in the vector. The function `shuffle` implements the actual shuffling. It takes an effect scope, a `Rng` instance `rng` in this scope and a list `lst`. We first convert the list to a mutable vector. Then we use `shuffleVector` to shuffle the vector 100 times, passing `rng` for generating the random numbers. Finally we convert the vector back to a list and return this result. We wrap

this computation in `runscope` to handle the effects of the mutable vector. The use of mutable vectors is not leaked outside of the function, from the type and behaviour of `shuffleVector` we are unable to find out if mutable vectors are used. We say that the use of the `State` effect is completely *encapsulated*. The type system ensures that `runscope` actually does encapsulate all effects in its scope. Note that we do not handle the scope of `rng`, we leave the `Rng` to be handled higher up by the caller of `shuffle`.

In Figure 3.5 we define the function `runshuffle` which can shuffle a list `lst`. We use a naive implementation of random number generation using a linear congruential generator. Inside of a new scope `s` we first create a mutable reference called `seedref`, which we set to a random intial seed value. This reference will store the current state of the random number generator, which we call the seed. We then create a fresh `Rng` instance called `rng`. We implement the `rand` operation by first getting the current seed value from `seedref`. Then we calculate a new seed value using arbitrarily chosen numbers, store this in `seedref` and call the continuation with it. The return and finally cases do not do anything special. Finally we call `shuffle` with our `Rng` instance and `lst`. In this example we can see how we can use other effects in the handler of an instance. The `Rng` uses an instance of `State` to implement the `rand` operation. Both of these effects exist and are handled in the same scope `s`. From the type of `runshuffle` (`List -> List`) we can see that all the effects are encapsulated and that the function is pure.

Figure 3.4: List shuffling

```
1  -- (linked) list of integer values
2  data List = Nil | Cons Int List
3
4  -- transform a list to a vector by replacing each value
5  -- in the list by a reference initialized with that value
6  toVector : forall s. List -> (Vector s)!{s}
7  toVector [s] Nil = VNil
8  toVector [s] (Cons h t) =
9    h' <- ref [s] h;
10   t' <- toVector [s] t;
11   return (VCons h' t')
12
13 -- transform a vector back to a list by getting the
14 -- current values from the references in the vector
15 toList : forall s. Vector s -> List!{s}
16 toList [s] VNil = Nil
17 toList [s] (VCons h t) =
18   h' <- h#get();
19   t' <- toList [s] t;
20   return (Const h' t')
21
22 -- shuffles a list given an instance of Rng
23 -- by converting it to a vector
24 -- and shuffling 100 times
25 shuffle : forall s'. Inst s' Rng -> List -> List!{s'}
26 shuffle [s'] rng lst =
27   runscope(s ->
28     let vec = toVector [s] lst;
29     shuffleVector [s] [s'] rng 100 vec;
30     return (toList vec))
```

Figure 3.5: List shuffling

```
runshuffle : List -> List
runshuffle lst =
  runscope(s ->
    seedref <- ref [s] 123456789;
    rng <- new Rng@s {
      rand n k ->
        currentseed <- seedref#get();
        let newseed = (1103515245 * currentseed + 12345) % n;
        seedref#put(newseed);
        k newseed
      return x -> return x
      finally x -> return x
    };
    shuffle [s] rng lst)
```

Chapter 4

Semantics and types of algebraic effects and handlers

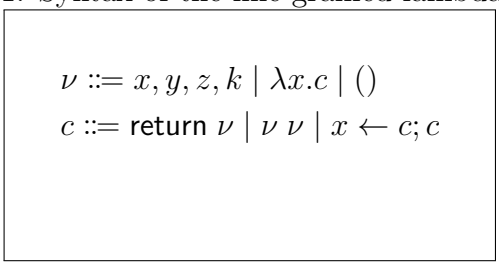
In this chapter we will show the basics of algebraic effects and handlers. We will start with the simply-typed lambda calculus (§4.1) and add algebraic effects (§4.2) and static instances (§4.3 to it.

4.1 Simply-typed lambda calculus

As our base language we will take the fine-grained call-by-value simply-typed lambda calculus (FG-STLC) [8]. This system is a version of the simply-typed lambda calculus with a syntactic distinction between values and computations. Because of this distinction there is exactly one evaluation order: call-by-value. In a system with side effects the evaluation order is very important since a different order could have a different result. Having the evaluation order be apparent from the syntax is thus a good choice for a system with algebraic effects. Another way to look at FG-STLC is to see it as a syntax for the lambda calculus that constrains the program to always be in A-normal form [9].

The terms are shown in Figure 4.1. The terms are split in to values and computations. Values are pieces of data that have no effects, while computations are terms that may have effects.

Figure 4.1: Syntax of the fine-grained lambda calculus



Values We have x, y, z, k ranging over variables, where we will use k for variables that denote continuations later on. Lambda abstractions are denoted as $\lambda x.c$, note that the body c of the abstraction is restricted to be a computation as opposed to the ordinary lambda calculus where the body can be any expression. To keep things simple we take unit () as our only base value, this because adding more base values will not complicate the theory. Using the unit value we can also delay computations by wrapping them in an abstraction that takes a unit value.

Computations For any value ν we have $\mathsf{return}\ \nu$ for the computation that simply returns a value without performing any effects. We have function application $(\nu\ \nu)$, where both the function and argument have to be values. Sequencing computations is done with $(x \leftarrow c; c)$. Normally in the lambda calculus the function and the argument in an application could be any term and so a choice would have to be made in what order these have to be evaluated or whether to evaluate the argument at all before substitution. In the fine-grained calculus both the function and argument in $(\nu\ \nu)$ are values so there's no choice of evaluation order. The order is made explicit by the sequencing syntax $(x \leftarrow c; c)$.

Semantics The small-step operational semantics is shown in Figure 4.2. The relation $\rightsquigarrow$ is defined on computations, where the $c \rightsquigarrow c'$ means c reduces to c' in one step. These rules are a fine-grained approach to the standard reduction rules of the simply-typed lambda calculus. In S-App we apply a lambda abstraction to a value argument, by substituting the value for the variable x in the body of the abstraction. In S-SeqReturn we sequence a computation that just returns a value in another computation by substituting the value for the variable x in the computation. Lastly, in S-Seq we can reduce a sequence of two computations, c_1 and c_2 by reducing the first, c_1.

Figure 4.2: Semantics of the fine-grained lambda calculus

$$\frac{}{(\lambda x.c)\ \nu \rightsquigarrow c[x := \nu]} \quad \text{(S-App)}$$

$$\frac{}{(x \leftarrow \mathsf{return}\ \nu; c) \rightsquigarrow c[x := \nu]} \quad \text{(S-SeqReturn)}$$

$$\frac{c_1 \rightsquigarrow c_1'}{(x \leftarrow c_1; c_2) \rightsquigarrow (x \leftarrow c_1'; c_2)} \quad \text{(S-Seq)}$$

Figure 4.3: Types of the fine-grained simply-typed lambda calculus

$$\tau ::= () \mid \tau \to \underline{\tau}$$
$$\underline{\tau} ::= \tau$$

We define $\rightsquigarrow^*$ as the transitive-reflexive closure of $\rightsquigarrow$. Meaning that c in $c \rightsquigarrow^* c$ can reach c' in zero or more steps, while c in $c \rightsquigarrow c'$ reaches c' in exactly on step.

Types Next we give the *types* in Figure 4.3. Similar to the terms we split the syntax into value and computation types. Values are typed by value types and computations are typed by computation types. A value type is either the unit type () or a function type with a value type τ as argument type and a computation type $\underline{\tau}$ as return type.

For the simply-typed lambda calculus a computation type is simply a value type, but when we add algebraic effects computation types will become more meaningful by recording the effects a computation may use.

Typing rules Finally we give the typing rules in Figure 4.4. We have a typing judgment for values $\Gamma \vdash \nu : \tau$ and a typing judgment for computations $\Gamma \vdash c : \underline{\tau}$. In both these judgments the context Γ assigns value types to variables.

Figure 4.4: Typing rules of the fine-grained simply-typed lambda calculus

$$\frac{\Gamma[x] = \tau}{\Gamma \vdash x : \tau} \quad \text{(T-Var)}$$

$$\frac{}{\Gamma \vdash () : ()} \quad \text{(T-Unit)}$$

$$\frac{\Gamma, x : \tau_1 \vdash c : \underline{\tau_2}}{\Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau_2}} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \nu : \tau}{\Gamma \vdash \mathsf{return}\ \nu : \underline{\tau}} \quad \text{(T-Return)}$$

$$\frac{\Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau_2} \qquad \Gamma \vdash \nu_2 : \tau_1}{\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau_2}} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash c_1 : \underline{\tau_1} \qquad \Gamma, x : \tau_1 \vdash c_2 : \underline{\tau_2}}{\Gamma \vdash (x \leftarrow c_1; c_2) : \underline{\tau_2}} \quad \text{(T-Seq)}$$

The rules for variables (T-Var), unit (T-Unit), abstractions (T-Abs) and applications (T-App) are the standard typing rules of the simply-typed lambda calculus. For $\mathsf{return}\ \nu$ (T-Return) we simply check the type of ν. For the sequencing of two computations $(x \leftarrow c_1; c_2)$ (T-Seq) we first check the type of c_1 and then check c_2 with the type of c_1 added to the context for x.

Examples To show the explicit order of evaluation we will translate the following program from the simply-typed lambda calculus into its fine-grained version:

$$f\ c_1\ c_2$$

Here we have a choice of whether to first evaluate c_1 or c_2 and whether to evaluate $(f\ c_2)$ before evaluating c_2. In the fine-grained system the choice of evaluation order is made explicit by the syntax. This means we can write down three variants for the above program, each having a different evaluation order. In the presence of effects all three may have different results.

1. c_1 before c_2, c_2 before $(f\ c_1)$

$$x' \leftarrow c_1; y' \leftarrow c_2; g \leftarrow (f\ x'); (g\ y')$$

2. c_2 before c_1, c_2 before $(f\ c_1)$

$$y' \leftarrow c_2; x' \leftarrow c_1; g \leftarrow (f\ x'); (g\ y')$$

3. c_1 before c_2, $(f\ c_1)$ before c_2

$$x' \leftarrow c_1; g \leftarrow (f\ x'); y' \leftarrow c_2; (g\ y')$$

To give a more concrete example, take a programming language based on the call-by-value lambda calculus that has arbitrary side-effects. Given a function print that takes an integer and prints it to the screen, we can define the following function printRange that prints a range of integers:

```
-- given print : Int -> ()
printRange : Int -> Int -> ()
printRange a b =
  if a > b then
    ()
  else
    (\a b -> ()) (print a) (printRange (a + 1) b)
```

Here we use a lambda abstraction (`(\a b -> ())`) in order to simulate sequencing. Knowing the evaluation order is very important when evaluating the call `(printRange 1 10)`. In the expression
`(\a b -> ()) (print a) (printRange (a + 1) b)`
the arguments can be either evaluated left-to-right or right-to-left, corresponding to (1) and (2) in the list above respectively. This makes a big difference in the output of the program, in left-to-right order the numbers 1 to 10 will be printed in increasing order while using a right-to-left evaluation strategy will print the numbers 10 to 1 in decreasing order. A third option is to first evaluation `(print a)` then the call `(\a b -> ()) (print a)`, resulting in `(\b -> ()) (printRange (a + 1) b)`, after which this application is reduced. This corresponds to (3) in the list above, but has the same result as (1) in this example. From the syntax of the language we are not able to deduce which evaluation order will be used, even worse it may be left undefined in the language definition.

Translating the evaluation order corresponding to (1) to a language that uses a fine-grain style syntax results in:

```
-- given print : Int -> ()
printRange : Int -> Int -> ()
printRange a b =
  if a > b then
    ()
  else
    _ <- print a;
    printRange (a + 1) b
```

Here from the syntax it is made clear that `print a` should be evaluated before `printRange (a + 1) b`, meaning a left-to-right evaluation order. Because the fine-grained lambda calculus has explicit sequencing syntax we do not have to use lambda abstraction (`(\a b -> ())`) for this purpose.

Alternatively a translation that corresponds to evaluation order (2) results in:

```
-- given print : Int -> ()
printRange : Int -> Int -> ()
printRange a b =
  if a > b then
    ()
  else
    _ <- printRange (a + 1) b;
    print a
```

Making clear we want a right-to-left evaluation order, printing the numbers in decreasing order.

Because we have eliminated the lambda abstraction there is no translation corresponding to (3), but semantically it would be identical to the first (left-to-right) translation.

Type soundness In order to prove type soundness for the previously defined calculus we first have define what it means for a computation to be a value. We define a computation c to be a value if c is of the form $\mathsf{return}\ \nu$ for some

value ν.

$$\mathsf{value}(c) \text{ if } \exists \nu. c = \mathsf{return}\ \nu$$

Using this definition we can state the following type soundness theorem for the fine-grained simply typed lambda calculus.

Theorem 1 (Type soundness)**.**

$$\mathsf{if}\ \cdot \vdash c : \underline{\tau}\ \wedge\ c \rightsquigarrow^* c'\ \mathsf{then}\ \mathsf{value}(c') \vee (\exists c''.\ c' \rightsquigarrow c'')$$

This states that given a well-typed computation c and taking some amount of steps then the resulting computation c' will be of either a value or another step can be taken. In other words the term will not get "stuck". Note that this is only true if the computation c is typed in the empty context. If the context is not empty then the computation could get stuck on free variables.

We can prove this theorem using the following lemmas:

Lemma 1 (Progress)**.**

$$\mathsf{if}\ \cdot \vdash c : \underline{\tau}\ \mathsf{then}\ \mathsf{value}(c) \vee (\exists c'.\ c \rightsquigarrow c')$$

Lemma 2 (Preservation)**.**

$$\mathsf{if}\ \Gamma \vdash c : \underline{\tau}\ \wedge\ c \rightsquigarrow c'\ \mathsf{then}\ \Gamma \vdash c' : \underline{\tau}$$

Where the progress lemma states that given a well-typed computation c then either c is a value or c can take a step. The preservation lemma states that given a well-typed computation c and if c can take a step to c' then c' is also well-typed. We can prove both these by induction on the typing derivations. Note again that the context has to be empty for the Progress lemma, again because the computation could get stuck on free variables. For the Preservation lemma the context can be anything however, since the operational semantics will not introduce any new free variables that are not already in the context.

Figure 4.5: Syntax of algebraic effects

$$\nu ::= x, y, z, k \mid \lambda x.c \mid ()$$
$$c ::= \mathsf{return}\ \nu \mid \nu\ \nu \mid x \leftarrow c; c \mid op(\nu) \mid \mathsf{handle}(c)\{h\}$$
$$h ::= op\ x\ k \to c;\ h \mid \mathsf{return}\ x \to c$$

4.2 Algebraic effects

We now extend the previous calculus with algebraic effects and handlers. We assume there is a set of effect names $\mathsf{EffName}$ with $E \subseteq \mathsf{EffName}$, for example $E = \{\mathsf{Flip}, \mathsf{State}, ...\}$. For each effect ϵ there assume there is a non-empty set of operations O^ϵ. For example $O^{\mathsf{Flip}} = \{\mathsf{flip}\}$ and $O^{\mathsf{State}} = \{\mathsf{get}, \mathsf{put}\}$.

Syntax The syntax for the extended system is shown in Figure 4.5, additions are highlighted with a gray background. Values stay the same. We add two forms of computations, operation calls $op(\nu)$ where $op \in O^\epsilon$ for some effect ϵ and we can handle computations using $\mathsf{handle}(c)\{h\}$. Handlers h are lists of operation cases $op\ x\ k \to c;\ h$ ending in the return case $\mathsf{return}\ x \to c$. We assume that operations are not repeated within a handler.

Semantics We give a small-step operational semantics in Figure 4.6. S-App, AlgEff-S-SeqReturn and S-Seq are the same as in the fine-grained system and are left out of the figure. To be able to handle a computation we first transform the computation to the form $\mathsf{return}\ \nu$ or $(x \leftarrow op(\nu); c)$. S-Flatten and S-Op are used to get a computation to those forms. The last four rules are used to handle a computation. S-HandleReturn handles a computation of the form $\mathsf{return}\ \nu$ by substituting ν in the body of the return case of the handler. S-HandleOp and S-HandleOpSkip handle computations of the form $(x \leftarrow op(\nu); c)$. If the operation op is contained in the handler h then the rule S-HandleOp substitutes the value ν of the operation call in the body of the matching operation case c'. We also substitute a continuation in c', which continues with the computation c wrapped by the same handler h. If the operation op is not contained in the handler then we

Figure 4.6: Semantics of algebraic effects

$$\frac{}{(x \leftarrow (y \leftarrow c_1; c_2); c_3) \rightsquigarrow (y \leftarrow c_1; (x \leftarrow c_2; c_3))} \quad (\text{S-Flatten})$$

$$\frac{}{op(\nu) \rightsquigarrow (x \leftarrow op(\nu); \mathsf{return}\ x)} \quad (\text{S-Op})$$

$$\frac{}{\mathsf{handle}(\mathsf{return}\ \nu)\{h; \mathsf{return}\ x \to c\} \rightsquigarrow c[x := \nu]} \quad (\text{S-HandleReturn})$$

$$\frac{op\ x\ k \to c' \in h}{\mathsf{handle}(y \leftarrow op(\nu); c)\{h\} \rightsquigarrow c'[x := \nu, k := (\lambda y.\mathsf{handle}(c)\{h\})]} \quad (\text{S-HandleOp})$$

$$\frac{op \notin h}{\mathsf{handle}(x \leftarrow op(\nu); c)\{h\} \rightsquigarrow (x \leftarrow op(\nu); \mathsf{handle}(c)\{h\})} \quad (\text{S-HandleOpSkip})$$

$$\frac{c \rightsquigarrow c'}{\mathsf{handle}(c)\{h\} \rightsquigarrow \mathsf{handle}(c')\{h\}} \quad (\text{S-Handle})$$

float out the operation call $op(\nu)$ and wrap the handler h around the continuing computation c. Lastly, S-Handle is able to reduce a computation in the handle computation.

Type syntax We now give a type system which ensures that a program reduced by the given semantics will not get "stuck" meaning that the result will be a computation of the form $\mathsf{return}\ \nu$ for some value ν. In Figure 4.7 we give the syntax of the types. Value types τ are the same as in the fine-grained system. Computation types $\underline{\tau}$ are now of the form $\tau\ !\ r$ for some value type

Figure 4.7: Types of algebraic effects

$$\tau ::= () \mid \tau \to \underline{\tau}$$
$$\underline{\tau} ::= \tau\ !\ r$$
$$r ::= \{\epsilon_1, ..., \epsilon_n\}$$

Figure 4.8: Subtyping rules of algebraic effects

$$\frac{}{() <: ()} \quad (\textsc{Sub-Unit})$$

$$\frac{\tau_3 <: \tau_1 \qquad \underline{\tau}_2 <: \underline{\tau}_4}{\tau_1 \to \underline{\tau}_2 <: \tau_3 \to \underline{\tau}_4} \quad (\textsc{Sub-Arr})$$

$$\frac{\tau_1 <: \tau_2 \qquad r_1 \subseteq r_2}{\tau_1 \,!\, r_1 <: \tau_2 \,!\, r_2} \quad (\textsc{Sub-Annot})$$

τ. An annotation $r \subseteq E$ is a set of effect names.

Subtyping It is always valid in the system to weaken a type by adding more effects to an annotation. This is done using subtyping judgments $\tau <: \tau$ and $\underline{\tau} <: \underline{\tau}$. In Figure 4.13 we give the subtyping rules for the system. Subtyping proceeds structurally on the value and computation types. In Sub-Arr we compare function arguments contravariantly. To compare two annotated types we compare the value types and then check that the annotation on the left is a subset of the annotation on the right.

Typing rules Finally we give the typing rules in Figure 4.14. We have three judgements:

1. $\Gamma \vdash \nu : \tau$, which types the value ν with the value type τ
2. $\Gamma \vdash c : \underline{\tau}$, which types the computation c with the computation type $\underline{\tau}$
3. $\Gamma \vdash^{\tau} h : \underline{\tau}$ which types the handler h with the computation type $\underline{\tau}$ given some value type τ

We can get the type of a variable from the context using $\Gamma[x] = \tau$. For each operation *op* we have a parameter type τ^1_{op} and a return type τ^2_{op}. We use the syntax $op \Rightarrow (\epsilon, \tau^1_{op}, \tau^2_{op})$ to retrieve the effect, parameter and return type given an operation *op*.

T-Var, T-Unit, T-Abs, T-App, and T-Seq are the same as in the fine-grained system. We can weaken the type of values and computations using subtyping using the rules T-SubVal and T-SubComp. For return computations return ν we type the value and annotate it with the empty effect set using the rule T-Return. T-Op shows that for operation calls we first

Figure 4.9: Typing rules of algebraic effects

$$\frac{\Gamma[x] = \tau}{\Gamma \vdash x : \tau \,!\, \varnothing} \quad (\text{T-Var})$$

$$\frac{}{\Gamma \vdash () : ()} \quad (\text{T-Unit})$$

$$\frac{\Gamma, x : \tau_1 \vdash c : \underline{\tau}_2}{\Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau}_2} \quad (\text{T-Abs})$$

$$\frac{\Gamma \vdash \nu : \tau_1 \qquad \tau_1 <: \tau_2}{\Gamma \vdash \nu : \tau_2} \quad (\text{T-SubVal})$$

$$\frac{\Gamma \vdash \nu : \tau}{\Gamma \vdash \mathsf{return}\ \nu : \tau \,!\, \varnothing} \quad (\text{T-Return})$$

$$\frac{\Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau}_2 \qquad \Gamma \vdash \nu_2 : \tau_1}{\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}_2} \quad (\text{T-App})$$

$$\frac{\Gamma \vdash c_1 : \tau_1 \,!\, r \qquad \Gamma, x : \tau_1 \vdash c_2 : \tau_2 \,!\, r}{\Gamma \vdash (x \leftarrow c_1; c_2) : \tau_2 \,!\, r} \quad (\text{T-Seq})$$

$$\frac{op \Rightarrow (\epsilon, \tau^1_{op}, \tau^2_{op}) \qquad \Gamma \vdash \nu : \tau^1_{op}}{\Gamma \vdash op(\nu) : \tau^2_{op} \,!\, \{\epsilon\}} \quad (\text{T-Op})$$

$$\frac{\Gamma \vdash c : \tau_1 \,!\, r_1 \qquad op \in h \Leftrightarrow op \in O^{\epsilon} \qquad \Gamma \vdash^{\tau_1} h : \tau_2 \,!\, r_2}{\Gamma \vdash \mathsf{handle}(c)\{h\} : \tau_2 \,!\, ((r_1 \setminus \{\epsilon\}) \cup r_2)} \quad (\text{T-Handle})$$

$$\frac{\Gamma \vdash c : \underline{\tau}_1 \qquad \underline{\tau}_1 <: \underline{\tau}_2}{\Gamma \vdash c : \underline{\tau}_2} \quad (\text{T-SubComp})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\tau_1} h : \tau_2 \,!\, r \\ op \Rightarrow (\epsilon, \tau^1_{op}, \tau^2_{op}) \qquad \Gamma, x : \tau^1_{op}, k : \tau^2_{op} \to \tau_2 \,!\, r \vdash c : \tau_2 \,!\, r\end{array}}{\Gamma \vdash^{\tau_1} (op\ x\ k \to c;\ h) : \tau_2 \,!\, r} \quad (\text{T-HOp})$$

$$\frac{\Gamma, x : \tau_1 \vdash c : \tau_2 \,!\, r}{\Gamma \vdash^{\tau_1} (\mathsf{return}\ x \to c) : \tau_2 \,!\, r} \quad (\text{T-HReturn})$$

lookup the operation in the context to find the effect, parameter and return types. We then check that the argument of the operation call is of the same type as the paramter type of the operation. Finally we type the operation call as an annotated type of the return type and a singleton effect set of the effect of the operation.

For handling we use the rule T-Handle. First we typecheck the type of the computation we are handling as having the computation type $\tau_1 \,!\, r_1$. Then we check that all operations in the handler h are in the set of operations of some effect ϵ, this means that handlers always have to contain exactly the operations of some effect. We then typecheck the handler h, giving it the type of the computation we are handling τ_1 and getting the return type $\tau_2 \,!\, r_2$. The return type of the handling computation is then τ_2 annotated with the effects from the handled computation minus the effect ϵ we handled together with the effects from the handler.

Finally the rules T-HOp and T-HReturn type the two cases of a handler. T-HReturn checks that the computation c of the return case types as $\tau_2 \,!\, r$ after adding x to Γ with the given type τ_1. $\tau_2 \,!\, r$ is the return type of the handler. T-HOp first checks the rest of the handler. Then the parameter and return types of the operation *op* are retrieved. Finally we add the parameter x of the operation and the continuation k to Γ and check that the type of the computation c agrees with the return type of the rest of the handler.

Type soundness TODO
We define a computation c to be a value if c is of the form $\mathsf{return}\ \nu$ for some value ν.

$$\mathsf{value}(c)\ \mathsf{if}\ \exists \nu. c = \mathsf{return}\ \nu$$

Theorem 2 (Type soundness)**.**

$$\mathsf{if}\ \cdot \vdash c : \tau \,!\, \varnothing\ \wedge\ c \rightsquigarrow^* c'\ \mathsf{then}\ \mathsf{value}(c') \vee (\exists c''.\ c' \rightsquigarrow c'')$$

Lemma 3 (Progress)**.**

$$\mathsf{if}\ \cdot \vdash c : \tau \,!\, \varnothing\ \mathsf{then}\ \mathsf{value}(c) \vee (\exists c'.\ c \rightsquigarrow c')$$

Lemma 4 (Preservation)**.**

$$\mathsf{if}\ \Gamma \vdash c : \underline{\tau}\ \wedge\ c \rightsquigarrow c'\ \mathsf{then}\ \Gamma \vdash c' : \underline{\tau}$$

Figure 4.10: Syntax of algebraic effects with static instances

$$\nu ::= x, y, z, k \mid \lambda x.c \mid () \mid \iota$$
$$c ::= \mathsf{return}\ \nu \mid \nu\ \nu \mid x \leftarrow c; c \mid \nu \# op(\nu) \mid \mathsf{handle}^{\nu}(c)\{h\}$$
$$h ::= op\ x\ k \to c;\ h \mid \mathsf{return}\ x \to c$$

4.3 Static instances

Finally extend algebraic effects with static instances. We assume there exists a set of instances $I = \{\iota_1, ..., \iota_n\}$, where each instance belongs to a single effect ϵ, written as $E[\iota] = \epsilon$.

Syntax The syntax of the system with algebraic effects and handlers is extended in Figure 4.10, changes and new additions are shown in gray. For values we add instances, these are taking from the set of instances I. We also change the operation call and handle computations to take an extra value term, which is the instance they are operating on.

Semantics The semantics for static instances are shown in Figure 4.11. The rules from algebraic effects that did not change are left out of this figure. The rules S-Op, S-HandleReturn and S-Handle are, except for the change in syntax with the addition of the value term, identical to the corresponding rules in the previous system. For static instances the only important change is in the S-HandleOp and S-HandleOpSkip rules. In S-HandleOp the instance in the handle and the instance in the operation call have to be the same, besides this the rule is the same as the corresponding rule in the previous system. If the instances do not match or if the operation is not in the handler then the rule S-HandleOpSkip is used to lift the operation call over the handler, also like in the previous system.

Type syntax The updated syntax for types is shown in Figure 4.12. We add instances types, which are just instance names from the set I. The effect annotation on the computation types are now sets of instance names instead effect names.

Figure 4.11: Semantics of algebraic effects with static instances

$$\frac{}{\nu_1 \# op(\nu_2) \rightsquigarrow (x \leftarrow \nu_1 \# op(\nu_2); \mathsf{return}\ x)} \quad (\text{S-Op})$$

$$\frac{}{\mathsf{handle}^{\nu_1}(\mathsf{return}\ \nu_2)\{h; \mathsf{return}\ x \to c\} \rightsquigarrow c[x := \nu_2]} \quad (\text{S-HandleReturn})$$

$$\frac{op\ x\ k \to c' \in h}{\mathsf{handle}^{\iota}(y \leftarrow \iota \# op(\nu); c)\{h\} \rightsquigarrow c'[x := \nu, k := (\lambda y.\mathsf{handle}^{\iota}(c)\{h\})]} \quad (\text{S-HandleOp})$$

$$\frac{op \notin h \wedge \iota_1 \neq \iota_2}{\mathsf{handle}^{\iota_1}(x \leftarrow \iota_2 \# op(\nu); c)\{h\} \rightsquigarrow (x \leftarrow \iota_2 \# op(\nu); \mathsf{handle}^{\iota_1}(c)\{h\})} \quad (\text{S-HandleOpSkip})$$

$$\frac{c \rightsquigarrow c'}{\mathsf{handle}^{\nu}(c)\{h\} \rightsquigarrow \mathsf{handle}^{\nu}(c')\{h\}} \quad (\text{S-Handle})$$

Figure 4.12: Types of algebraic effects with static instances

$$\tau ::= () \mid \mathsf{inst}(\iota) \mid \tau \to \underline{\tau}$$

$$\underline{\tau} ::= \tau\ !\ r$$

$$r ::= \{\iota_1, ..., \iota_n\}$$

Figure 4.13: Subtyping rules of algebraic effects with static instances

$$\frac{}{\mathsf{inst}(\iota) <: \mathsf{inst}(\iota)} \quad (\textsc{Sub-Inst})$$

Figure 4.14: Typing rules of algebraic effects with static instances

$$\frac{}{\Gamma \vdash \iota : \mathsf{inst}(\iota)} \quad (\textsc{T-Inst})$$

$$\frac{\Gamma \vdash \nu_1 : \mathsf{inst}(\iota) \qquad E[\iota] = \epsilon \qquad \Gamma[op] = (\epsilon, \tau^1_{op}, \tau^2_{op}) \qquad \Gamma \vdash \nu_2 : \tau^1_{op}}{\Gamma \vdash \nu_1 \# op(\nu_2) : \tau^2_{op} \,!\, \{\iota\}} \quad (\textsc{T-Op})$$

$$\frac{\begin{array}{c}\Gamma \vdash \nu : \mathsf{inst}(\iota) \\ E[\iota] = \epsilon \qquad \Gamma \vdash c : \tau_1 \,!\, r_1 \qquad op \in h \Leftrightarrow op \in O^{\epsilon} \qquad \Gamma \vdash^{\tau_1} h : \tau_2 \,!\, r_2\end{array}}{\Gamma \vdash \mathsf{handle}^{\nu}(c)\{h\} : \tau_2 \,!\, ((r_1 \setminus \{\iota\}) \cup r_2)} \quad (\textsc{T-Handle})$$

Subtyping For subtyping we keep the rules from the previous system but we add a rule for the instance types (Figure 4.13).

Typing rules The typing rules from the previous system mostly stay the same except for the rules T-OP and T-Handle, they are showin in Figure 4.14. We also had a rule to type instances (T-Inst), this rule simply types an instance as a instance type with the same name. For both T-OP and T-Handle we just have to check that the added value term is an instance and that the effect of that instance matches the operations.

Type soundness TODO, same as algebraic effects TODO
We define a computation c to be a value if c is of the form $\mathsf{return}\ \nu$ for some value ν.

$$\mathsf{value}(c) \text{ if } \exists \nu. c = \mathsf{return}\ \nu$$

Theorem 3 (Type soundness)**.**

$$\text{if } \cdot \vdash c : \tau \,!\, \varnothing \ \wedge\ c \rightsquigarrow^* c' \text{ then } \mathsf{value}(c') \vee (\exists c''.\ c' \rightsquigarrow c'')$$

Lemma 5 (Progress)**.**

$$\text{if } \cdot \vdash c : \tau \,!\, \varnothing \text{ then } \mathsf{value}(c) \vee (\exists c'.\ c \rightsquigarrow c')$$

Lemma 6 (Preservation)**.**

$$\mathsf{if}\ \Gamma \vdash c : \underline{\tau}\ \wedge\ c \rightsquigarrow c'\ \mathsf{then}\ \Gamma \vdash c' : \underline{\tau}$$

Chapter 5

Semantics and types of X

In this chapter we give a formal account of X. We give the syntax, typing rules and a small-step operation semantics. We end the chapter with a type soundness theorem. The system builds on the formal system with algebraic effects, handlers and static instances of Chapter 4.3. We add constructs to handle effect scope polymorphism. We also add a construct to dynamically create new instances. Finally we add constructs to handle effect scopes.

In Section 5.1 we give the syntax of the terms and types of X. In Section 5.2 we give the environments and judgments used in the typing rules and semantics. In Section 5.3 we give subtyping rules for the types. In Section 5.4 we give well-formedness rules for the types. In Section 5.5 we give the typing rules. In Section 5.6 we give a small-step operation semantics for X. Finally in Section 5.7 we give a type-soundness theorem.

5.1 Syntax

Just like in the formal systems of algebraic effects of Chapter 4.2 and Chapter 4.3 we assume there is set of effect names $\mathsf{EffName}$ with $E \subseteq \mathsf{EffName}$. For example $E = \{\mathsf{Flip}, \mathsf{State}, \mathsf{Exc}, ...\}$. Each effect ε has a non-empty set of operation names O^{ε}. For example $O^{\mathsf{Flip}} = \{\mathsf{flip}\}$ and $O^{\mathsf{State}} = \{\mathsf{get}, \mathsf{put}\}$. Every operation name only corresponds to a single effect. Each operation *op*

Figure 5.1: Syntax

$$
\begin{aligned}
s &::= s_{var} \mid s_{loc} \\
\tau &::= \mathsf{Inst}\ s\ \varepsilon \mid \tau \to \underline{\tau} \mid \forall s_{var}.\underline{\tau} \\
\underline{\tau} &::= \tau\ !\ r \\
\nu &::= x, y, z, k \mid \mathsf{inst}(l) \mid \lambda x.c \mid \Lambda s_{var}.c \\
c &::= \mathsf{return}\ \nu \mid \nu\ \nu \mid x \leftarrow c;\ c \mid \nu \# op(\nu) \mid \nu\ [s] \\
&\quad \mid \mathsf{new}\ \varepsilon @ s\ \{h; \mathsf{finally}\ x \to c\}\ \mathsf{as}\ x\ \mathsf{in}\ c \\
&\quad \mid \mathsf{runscope}(s_{var} \to c) \\
&\quad \mid \mathsf{runscope}^{s_{loc}}(c) \\
&\quad \mid \mathsf{runinst}^{l}\{h\}(c) \\
h &::= op\ x\ k \to c; h \mid \mathsf{return}\ x \to c
\end{aligned}
$$

has a parameter type τ^1_{op} and a return type τ^2_{op}. Locations l are modeled by some countable infinite set.

In Figure 5.1 we show the syntax of the types and terms of X. An effect scope s is either a scope variable s_{var} or a scope location s_{loc}. Effect scope variables s_{var} and effects scope locations s_{loc} are both modeled by countable infinite sets.

Like in the systems in Chapter 4 terms and types are both split between values and computations, and value types and computation types. Values are typed by value types and computations are typed by computation types.

Value types τ are either an instance type $\mathsf{Inst}\ s\ \varepsilon$, indexed by an effect scope s and an effect ε. Or a function type $\tau \to \underline{\tau}$ where the paramter type is a value type and the return type is a computation type. Or an universally quantified computation type $\forall s_{var}.\underline{\tau}$, where the domain of quantification are effect scopes.

A computation type $\underline{\tau}$ is always an annotated value type of the form $\tau\ !\ r$.

Annotations r are sets of effect scopes $\{s_1, ..., s_n\}$.

Values are either variables x, k, where we always use k do denote variables that refer to continuations. Or instances $\mathsf{inst}(l)$, indexed by some location l. Instances would not appear in the surface language, but are introduced by the semantics. Or lambda abstractions $\lambda x.c$, where the body is a computation. Or effect scope abstractions $\Lambda s_{var}.c$, where we abstract over a computation c, with the domain of the quantification being effect scopes.

For computations we have $\mathsf{return}\ \nu$, to lift a value ν in to a computation. We have application $\nu\ \nu$ and sequencing $x \leftarrow c;\ c$. We have operation calls $\nu\#op(\nu)$. The new constructs are as follows. We have effect scope application $\nu\ [s]$. We can create new instances with $\mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ x \to c\}\ \mathsf{as}\ x\ \mathsf{in}\ c$, where h is a handler. We can handle computations with $\mathsf{runscope}(s_{var} \to c)$. Finally we have two more constructs which would not appear in the surface language, but are introduced by the semantics. Effect scope handlers $\mathsf{runscope}^{s_{loc}}(c)$ handle a specific scope s_{loc} in the computation c. Instance handlers $\mathsf{runinst}^{l}\{h\}(c)$ handle the operations of a single instance of the location l in the computation c.

Finally we have handlers h which are lists of operation cases ending with a return case. Operation cases are of the form $op\ x\ k \to c; h$, where h is the rest of the handler. Return cases are of the form $\mathsf{return}\ x \to c$.

5.2 Environments and judgments

Figure 5.2: Environments

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$
$$\Delta ::= \cdot \mid \Delta, s_{var}$$
$$\Sigma ::= \cdot \mid \Sigma, s_{loc} \mid \Sigma, l := (s_{loc}, \varepsilon)$$

Environments Before looking at the different judgments, we will introduce the three environments used. The syntax for the environment are shown in figure 5.2.

- Γ is the typing enviroment which assigns variables x to value types τ.
- Δ is the effect scope variable environment which keeps track of the scope variables s_{var} that are in use.
- Σ is the dynamic environment which keeps track of scope location s_{loc} and instance locations l. Instance locations are assigned a tuple of a scope location and an effect (s_{loc}, ε). Σ is used in both the typing rules and the operational semantics.

Judgments There are three kinds of judgments: subtyping, well-formedness and typing.

The subtyping judgments are used to weaken the effect annotation of a computation type. Weakening the effect annotation is sometimes necessary in order to type a program. For example when typing the sequencing of two computations $x \leftarrow c_1;\ c_2$, if the two computations do not agree on the effects then subtyping can be used to weaken both the computations such that the effect annotations agree. There is a subtyping judgment for both the value types τ and the computation types $\underline{\tau}$ these mutually depend on one another:

- $\tau <: \tau'$ holds when the value type τ is a subtype of τ'.
- $\underline{\tau} <: \underline{\tau}'$ holds when the computation type $\underline{\tau}$ is a subtype of $\underline{\tau}'$.

The well-formedness judgments check that scopes used in the types are valid under the scope variable and dynamic environments. There are three judgments of this kind:

- $\Delta; \Sigma \vdash s$ checks that the scope s is either in Δ if it is a scope variable or else in Σ if it is a scope location.
- $\Delta; \Sigma \vdash \tau$ checks that all the scopes in the value type τ are valid under the environments Δ and Σ.
- $\Delta; \Sigma \vdash \underline{\tau}$ checks that all the scopes in the computation type $\underline{\tau}$ are valid under the environments Δ and Σ.

Lastly there are three typing judgments:

- $\Delta; \Sigma; \Gamma \vdash \nu : \tau$ checks that the value ν has the value type τ under the Δ, Σ and Γ environments.
- $\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}$ checks that the computation c has the computation type $\underline{\tau}$ under the Δ, Σ and Γ environments.
- $\Delta; \Sigma; \Gamma \vdash^{\tau} h : \underline{\tau}$ checks that the handler h transform a return value of type τ to the computation type $\underline{\tau}$.

5.3 Subtyping

Figure 5.3: Subtyping

$$\frac{}{\mathsf{Inst}\ s\ \varepsilon <: \mathsf{Inst}\ s\ \varepsilon}\ (\textsc{Sub-Inst}) \qquad \frac{\tau_2 <: \tau_1 \qquad \underline{\tau}_1 <: \underline{\tau}_2}{\tau_1 \to \underline{\tau}_1 <: \tau_2 \to \underline{\tau}_2}\ (\textsc{Sub-Arr})$$

$$\frac{\underline{\tau}_1 <: \underline{\tau}_2}{\forall s_{var}.\underline{\tau}_1 <: \forall s_{var}.\underline{\tau}_2}\ (\textsc{Sub-Forall}) \qquad \frac{\tau_1 <: \tau_2 \qquad r_1 \subseteq r_2}{\tau_1\ !\ r_1 <: \tau_2\ !\ r_2}\ (\textsc{Sub-Annot})$$

In Figure 5.3 we give the subtyping rules for both the value and the computation types. The subtyping checks that that the effects mentioned in the type on the right are the same or more general than the type on the left. An instance type $\mathsf{Inst}\ s\ \varepsilon$ is a subtype of another instance type if they are structurally equal, shown in the rule Sub-Inst. Function types $\tau \to \underline{\tau}$ are compared by subtyping the parameter types contravariantly and subtyping the return types covariantly, shown in the rule Sub-Arr. Universally quantified types $\forall s_{var}.\underline{\tau}$ are structurally recursed upon, given they the quantified variables are equal (Sub-Forall). Lastly annotated types $\tau\ !\ r$ are compared by comparing the value types and checking that the annotation on the left type is a subtype of the annotation on the right type (Sub-Annot).

Figure 5.4: Well-formedness

$$\frac{s_{var} \in \Delta}{\Delta; \Sigma \vdash s_{var}} \text{ (WF-SVar)} \qquad \frac{s_{loc} \in \Sigma}{\Delta; \Sigma \vdash s_{loc}} \text{ (WF-SLoc)}$$

$$\frac{\Delta; \Sigma \vdash s}{\Delta; \Sigma \vdash \mathsf{Inst}\ s\ \varepsilon} \text{ (WF-Inst)} \qquad \frac{\Delta; \Sigma \vdash \tau \qquad \Delta; \Sigma \vdash \underline{\tau}}{\Delta; \Sigma \vdash \tau \to \underline{\tau}} \text{ (WF-Arr)}$$

$$\frac{\Delta, s_{var}; \Sigma \vdash \underline{\tau}}{\Delta; \Sigma \vdash \forall s_{var}.\underline{\tau}} \text{ (WF-Forall)} \qquad \frac{\Delta; \Sigma \vdash \tau \qquad \forall (s \in r) \Rightarrow \Delta; \Sigma \vdash s}{\Delta; \Sigma \vdash \tau\ !\ r} \text{ (WF-Annot)}$$

5.4 Well-formedness

In Figure 5.4 we give the well-formedness rules for the value and computation types. Well-formedness checks that the effect scopes in the type are accounted for in the environments. The rules WF-SVar and WF-SLoc check that the effect scope variables s_{var} and locations s_{loc} are valid by checking that they are contained in the scope and dynamic environments respectively. For instance types we check that the mentioned effect scope is valid (WF-Inst). For function types $\tau \to \underline{\tau}$ we check that both the paramter and return type is valid. For universally quantified types $\forall s_{var}.\underline{\tau}$ we check that the computation type $\underline{\tau}$ is valid, after adding the variable s_{var} to the environment. Lastly for annotated types $\tau\ !\ r$ we first check that that the value type τ is valid. Then we check that each effect scope in the annotation r is valid.

5.5 Typing rules

The typing rules for the values are given in Figure 5.5. The rules T-Var, T-Abs and T-SubVal are practically unchanged from the corresponding rules in the algebraic effects type system from Chapter 4.2. Instances $\mathsf{inst}(l)$ are assigned instance types, with the scope location and effect looked up in the dynamic environment Σ using the location l (T-Inst). Similar to abstractions, effect scope abstractions are assigned a universally quantified type $\forall s_{var}.\underline{\tau}$, by typing the body with $\underline{\tau}$ after adding s_{var} to the scope environment Δ (T-TAbs).

Figure 5.5: Value typing rules

$$\frac{\Gamma[x] = \tau}{\Delta; \Sigma; \Gamma \vdash x : \tau} \quad \text{(T-Var)} \qquad \frac{\Sigma(l) = (s_{loc}, \varepsilon)}{\Delta; \Sigma; \Gamma \vdash \mathsf{inst}(l) : \mathsf{Inst}\ s_{loc}\ \varepsilon} \quad \text{(T-Inst)}$$

$$\frac{\Delta; \Sigma; \Gamma, x : \tau \vdash c : \underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \lambda x.c : \tau \to \underline{\tau}} \quad \text{(T-Abs)} \qquad \frac{\Delta, s_{var}; \Sigma; \Gamma \vdash c : \underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \Lambda s_{var}.c : \forall s_{var}.\underline{\tau}} \quad \text{(T-TAbs)}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu : \tau_1 \qquad \Delta; \Sigma \vdash \tau_2 \qquad \tau_1 <: \tau_2}{\Delta; \Sigma; \Gamma \vdash \nu : \tau_2} \quad \text{(T-SubVal)}$$

Figure 5.6: Computation typing rules

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu : \tau}{\Delta; \Sigma; \Gamma \vdash \mathsf{return}\ \nu : \tau\ !\ \varnothing} \quad \text{(T-Return)}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu_1 : \tau \to \underline{\tau} \qquad \Delta; \Sigma; \Gamma \vdash \nu_2 : \tau}{\Delta; \Sigma; \Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}} \quad \text{(T-App)}$$

$$\frac{\Delta; \Sigma \vdash s \qquad \Delta; \Sigma; \Gamma \vdash \nu : \forall s'_{var}.\underline{\tau}}{\Delta; \Sigma; \Gamma \vdash \nu\ [s] : \underline{\tau}[s'_{var} := s]} \quad \text{(T-TApp)}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash c_1 : \tau_1\ !\ r \qquad \Delta; \Sigma; \Gamma, x : \tau_1 \vdash c_2 : \tau_2\ !\ r}{\Delta; \Sigma; \Gamma \vdash (x \leftarrow c_1;\ c_2) : \tau_2\ !\ r} \quad \text{(T-Seq)}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash \nu_1 : \mathsf{Inst}\ s\ \varepsilon \qquad op \in O^{\varepsilon} \qquad \Delta; \Sigma; \Gamma \vdash \nu_2 : \tau^1_{op}}{\Delta; \Sigma; \Gamma \vdash \nu_1 \# op(\nu_2) : \tau^2_{op}\ !\ \{s\}} \quad \text{(T-Op)}$$

$$\frac{\begin{array}{c}\Delta; \Sigma \vdash s \\ op \in O^{\varepsilon} \iff op \in h \qquad \Delta; \Sigma; \Gamma, x : \mathsf{Inst}\ s\ \varepsilon \vdash c : \tau_1\ !\ r \\ \Delta; \Sigma; \Gamma \vdash^{\tau_1} h : \tau_2\ !\ r \qquad s \in r \qquad \Delta; \Sigma; \Gamma, y : \tau_2 \vdash c' : \tau_3\ !\ r\end{array}}{\Delta; \Sigma; \Gamma \vdash \mathsf{new}\ \varepsilon @ s\ \{h; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c : \tau_3\ !\ r} \quad \text{(T-New)}$$

$$\frac{\Delta, s_{var}; \Sigma; \Gamma \vdash c : \tau\ !\ r \qquad s_{var} \notin \tau}{\Delta; \Sigma; \Gamma \vdash \mathsf{runscope}(s_{var} \to c) : \tau\ !\ (r \setminus \{s_{var}\})} \quad \text{(T-Handle)}$$

$$\frac{s_{loc} \in \Sigma \qquad \Delta; \Sigma; \Gamma \vdash c : \tau\ !\ r \qquad s_{loc} \notin \tau}{\Delta; \Sigma; \Gamma \vdash \mathsf{runscope}^{s_{loc}}(c) : \tau\ !\ (r \setminus \{s_{loc}\})} \quad \text{(T-HandleScope)}$$

$$\frac{\begin{array}{c}\Sigma(l) = (s_{loc}, \varepsilon) \qquad op \in O^{\varepsilon} \iff op \in h \\ \Delta; \Sigma; \Gamma \vdash c : \tau_1\ !\ r \qquad \Delta; \Sigma; \Gamma \vdash^{\tau_1} h : \tau_2\ !\ r\end{array}}{\Delta; \Sigma; \Gamma \vdash \mathsf{runinst}^{l}\{h\}(c) : \tau_2\ !\ r} \quad \text{(T-HandleInst)}$$

$$\frac{\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}_1 \qquad \Delta; \Sigma \vdash \underline{\tau}_2 \qquad \underline{\tau}_1 <: \underline{\tau}_2}{\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}_2} \quad \text{(T-SubComp)}$$

The typing rule for the computations are given in Figure 5.6. The rules T-Return, T-App, T-Seq and T-SubComp are practically unchanged from the corresponding rules in the algebraic effects type system from Chapter 4.2.

The rule for effect scope application (T-TApp) checks that, in the application $\nu\ [s]$, the scope s is well-formed. Then we check that ν is a universally quantified type $\forall s'.\underline{\tau}$. Finally we substitute the given scope s for the quantified variable s' in $\underline{\tau}$.

For operation calls (T-Op) $\nu_1 \# op(\nu_2)$ we first check that the type of ν_1 is an instance type $\mathsf{Inst}\ s\ \varepsilon$. We then check that the operation op is an operation of the effect of the instance ε ($op \in O^{\varepsilon}$). Lastly we check that the given value ν_2 matches the parameter type of the operation op (τ^1_{op}). The type given to the operation call is the return type of the operation τ^2_{op} with the scope of the instance, s, in the annotation.

The rule T-New types the creation of new instance: $\mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c$. We are creating a new instance x of effect ε in the effect scope s. First we check that the given effect scope s is valid. Then we check that the operations in the handler h are exactly the operations of the effect of the new instance ε. This means it is not valid to either omit operations or to have operations of other effects in the handler. This way we can ensure that every operation is accounted for. We then typecheck the computation c with the new instance added the to the environment as x, with the type $\tau_1\ !\ r$. Then we typecheck the handler h, passing the type τ_1 of c to the handler typing judgement. The handler can transform the return type τ_1 to another type τ_2, but note that the effects annotation r has to be the same. We check that the scope s is contained in r. The reason for this is because the creation of an instance in s is also an effect and so we have to account for this effect in the annotation by adding s to r. Lastly we check the $\mathsf{finally}$ case. We add τ_2 to the environment as y and typecheck the computation c' as $\tau_3\ !\ r$. The $\mathsf{finally}$ case is allowed to transform the return type of the handler τ_2 to another type τ_3. Finally the type of the whole instance creation is $\tau_3\ !\ r$. Where τ_3 is the final type of the $\mathsf{finally}$ case and r is the effects from c, the handler h and the $\mathsf{finally}$ case.

The rule T-Handle types handling computations: $\mathsf{runscope}(s_{var} \to c)$. We typecheck the body c with the effect scope variable s_{var} added to the scope

environment. Then we check that the effects do not escape their scope by checking that that s_{var} is not contained in the return type τ. We then type the whole computation as $\tau \,!\, (r \setminus \{s_{var}\})$. Knowing that s_{var} does not escape we can safely remove it from the effect annotation.

The rule T-HandleScope deals with the handling of a specific scope s_{loc} and is very similar to the previous rule T-Handle. Instead of the effect scope *variable* s_{var} we now deal with an effect scope *location* s_{loc}. We have to check that the location actually exists, by checking that it is contained in the dynamic environment Σ. We can then proceed like in T-Handle, checking that s_{loc} does not escape.

T-HandleInst typechecks the handling of an instance at location l using a handler h. First we check that the *location* is contained in the dynamic environment Σ We retrieve the scope location s_{loc} and effect ε of the instance from Σ. We check that the operations in the handler match the operations of the effect. We then typecheck the computation c and the handler h like in T-New.

Figure 5.7: Handler typing rules

$$\frac{\Delta; \Sigma; \Gamma \vdash^{\tau_1} h : \tau_2 \,!\, r \qquad \Delta; \Sigma; \Gamma, x : \tau^1_{op}, k : \tau^2_{op} \to \tau_2 \,!\, r \vdash c : \tau_2 \,!\, r}{\Delta; \Sigma; \Gamma \vdash^{\tau_1} (op\ x\ k \to c; h) : \tau_2 \,!\, r} \quad (\text{T-HandlerOp})$$

$$\frac{\Delta; \Sigma; \Gamma, x : \tau_1 \vdash c : \tau_2 \,!\, r}{\Delta; \Sigma; \Gamma \vdash^{\tau_1} (\mathsf{return}\ x \to c) : \tau_2 \,!\, r} \quad (\text{T-HandlerReturn})$$

Finally we discuss the typing rules for the handlers, given in Figure 5.7. T-HandlerReturn types the return case of a handler. We typecheck the body c after adding the variable x with type τ_1 to the environment. The type τ_1 is passed with the handler typing judgement in the typing rules T-New and T-HandleInst. It is the return type of the computation we are handling. The computation c can transform this type to another type τ_2 with some effect annotation r, which is the return type of the return case.

The rule T-HandlerOp shows the typing of an operation case. We first

typecheck the rest of handler h, passing along the return type τ_1 of the computation we are handling. We typecheck the rest of the handler as $\tau_2 \,!\, r$, this is the return type of the whole handler. Then we typecheck the body of the operation case c. We add the operation call argument x with the parameter type τ^1_{op} of the operation to the environment. We also add the continuation k to the environment. This is a function from the return type τ^2_{op} of the operation, to the return type of the whole handler $\tau_2 \,!\, r$. We check that the body of the case c returns a computation of the same type.

5.6 Semantics

Finally we give a small-step operational semantics for X.

The rule $c_1 \mid \Sigma_1 \rightsquigarrow c_2 \mid \Sigma_2$ takes a step from the computation c_1 to c_2 in the dynamic environment Σ_1. Σ_1 might be updated with new effect scope locations or instance locations resulting in Σ_2.

Figure 5.8: Semantics

$$\frac{}{(\lambda x.c)\ \nu \mid \Sigma \rightsquigarrow c[x := \nu] \mid \Sigma} \quad \text{(S-App)}$$

$$\frac{}{(\Lambda s_{var}.c)\ [s'] \mid \Sigma \rightsquigarrow c[s_{var} := s'] \mid \Sigma} \quad \text{(S-TApp)}$$

$$\frac{c_1; \Sigma \rightsquigarrow c_1'; \Sigma'}{(x \leftarrow c_1;\ c_2) \mid \Sigma \rightsquigarrow (x \leftarrow c_1';\ c_2) \mid \Sigma'} \quad \text{(S-Seq)}$$

$$\frac{}{(x \leftarrow (\mathsf{return}\ \nu);\ c) \mid \Sigma \rightsquigarrow c[x := \nu] \mid \Sigma} \quad \text{(S-SeqReturn)}$$

$$\frac{}{(y \leftarrow (x \leftarrow c_1;\ c_2);\ c_3 \mid \Sigma \rightsquigarrow (x \leftarrow c_1;\ y \leftarrow c_2;\ c_3) \mid \Sigma} \quad \text{(S-Flatten)}$$

$$\frac{}{\begin{array}{c}(x \leftarrow (\mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ z \to c_3\}\ \mathsf{as}\ y\ \mathsf{in}\ c_1);\ c_2) \mid \Sigma \rightsquigarrow \\ \mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ z \to c_3\}\ \mathsf{as}\ y\ \mathsf{in}\ (x \leftarrow c_1;\ c_2) \mid \Sigma\end{array}} \quad \text{(S-LiftNew)}$$

$$\frac{s_{loc} \notin \Sigma}{\mathsf{runscope}(s_{var} \to c) \mid \Sigma \rightsquigarrow \mathsf{runscope}^{s_{loc}}(c[s_{var} := s_{loc}]) \mid \Sigma, s_{loc}} \quad \text{(S-FreshScope)}$$

In Figure 5.8 we give the semantics for every construct except the effect scope and instance handlers. The rules S-App, S-Seq, S-SeqReturn and S-Flatten are the same as the corresponding rules in the algebraic effects system of Chapter 4.2. The rule S-TApp handles an effect scope application similarly to a normal application, by substituting the effect scope s' for the scope variable s_{var} of the scope abstraction. S-LiftNew lifts the creation of an instance out and over sequencing. By repeatedly applying this rule we can bubble up the instance creation until we hit an effect scope handler The rule S-FreshScope creates a fresh scope location with which to handle instances. A fresh scope location s_{loc} is created and added to the environment Σ. This new scope location is substituted in the body c. We wrap the computation with an effect scope handler for new scope location s_{loc}.

Figure 5.9: Semantics of effect scope handlers

$$\frac{c \mid \Sigma \rightsquigarrow c' \mid \Sigma'}{\mathsf{runscope}^{s_{loc}}(c) \mid \Sigma \rightsquigarrow \mathsf{runscope}^{s_{loc}}(c') \mid \Sigma'} \quad \text{(S-HandleScopeCong)}$$

$$\frac{}{\mathsf{runscope}^{s_{loc}}(\mathsf{return}\ \nu) \mid \Sigma \rightsquigarrow \mathsf{return}\ \nu \mid \Sigma} \quad \text{(S-HandleScopeReturn)}$$

$$\frac{}{\mathsf{runscope}^{s_{loc}}(\nu_1 \# op(\nu_2)) \mid \Sigma \rightsquigarrow \nu_1 \# op(\nu_2) \mid \Sigma} \quad \text{(S-HandleScopeOp)}$$

$$\frac{}{\mathsf{runscope}^{s_{loc}}(x \leftarrow \nu_1 \# op(\nu_2);\ c) \mid \Sigma \rightsquigarrow (x \leftarrow \nu_1 \# op(\nu_2);\ \mathsf{runscope}^{s_{loc}}(c)) \mid \Sigma} \quad \text{(S-HandleScopeSeqOp)}$$

$$\frac{s_{loc} \neq s'_{loc}}{\begin{array}{c}\mathsf{runscope}^{s_{loc}}(\mathsf{new}\ \varepsilon@s'_{loc}\ \{h; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c) \mid \Sigma \rightsquigarrow \\ \mathsf{new}\ \varepsilon@s'_{loc}\ \{h; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{runscope}^{s_{loc}}(c) \mid \Sigma\end{array}} \quad \text{(S-HandleScopeNewSkip)}$$

$$\frac{l \notin \mathsf{Dom}(\Sigma)}{\begin{array}{c}\mathsf{runscope}^{s_{loc}}(\mathsf{new}\ \varepsilon@s_{loc}\ \{h; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c) \mid \Sigma \rightsquigarrow \\ \mathsf{runscope}^{s_{loc}}(y \leftarrow \mathsf{runinst}^{l}\{h\}(c[x := \mathsf{inst}(l)]);\ c') \mid \Sigma, l := (s_{loc}, \varepsilon)\end{array}} \quad \text{(S-HandleScopeNew)}$$

In Figure 5.9 we give the semantics for the effect scope handlers. An effect scope handler for a specific effect scope location will create fresh instances when a new construct is encountered (S-HandleScopeNew). An instance handler with the handler of the new construct is wrapped around the computation and the finally case is wrapped around that. The newly created location l is added to the dynamic environment together with the scope lo-

cation and the effect. If a new is encountered with a different effect scope we skip it and nest the scope handler inside (T-HandleScopeNewSkip). Using the rule S-HandleScopeCong to reduce a computation inside a scope handler. We can remove a scope handler if we encounter either a return or operation call (S-HandleScopeReturn and S-HandleScopeOp). Effect scope handlers can be pushed inside sequencing, lifting an operation call over it (S-HandleScopeSeqOp).

Figure 5.10: Semantics of instance handlers

$$\frac{c \mid \Sigma \rightsquigarrow c' \mid \Sigma'}{\mathsf{runinst}^l\{h\}(c) \mid \Sigma \rightsquigarrow \mathsf{runinst}^l\{h\}(c') \mid \Sigma'} \quad \text{(S-HandleInstCong)}$$

$$\frac{}{\mathsf{runinst}^l\{h\}(\mathsf{new}\ \varepsilon@s\ \{h'; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ c) \mid \Sigma \rightsquigarrow \mathsf{new}\ \varepsilon@s\ \{h'; \mathsf{finally}\ y \to c'\}\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{runinst}^l\{h\}(c) \mid \Sigma} \quad \text{(S-HandleInstNew)}$$

$$\frac{}{\mathsf{runinst}^l\{h\}(\nu_1 \# op(\nu_2)) \mid \Sigma \rightsquigarrow \mathsf{runinst}^l\{h\}(x \leftarrow \nu_1 \# op(\nu_2);\ \mathsf{return}\ x) \mid \Sigma} \quad \text{(S-HandleInstOpPrepare)}$$

$$\frac{l \neq l'}{\mathsf{runinst}^l\{h\}(x \leftarrow \mathsf{inst}(l') \# op(\nu);\ c) \mid \Sigma \rightsquigarrow (x \leftarrow \mathsf{inst}(l') \# op(\nu);\ \mathsf{runinst}^l\{h\}(c)) \mid \Sigma} \quad \text{(S-HandleInstOpSkip)}$$

$$\frac{h[op] = (x, k, c_{op})}{\mathsf{runinst}^l\{h\}(y \leftarrow \mathsf{inst}(l) \# op(\nu);\ c) \mid \Sigma \rightsquigarrow c_{op}[x := \nu, k := (\lambda y.\mathsf{runinst}^l\{h\}(c))] \mid \Sigma} \quad \text{(S-HandleInstOp)}$$

$$\frac{}{\mathsf{runinst}^l\{h; \mathsf{return}\ x_r \to c_r\}(\mathsf{return}\ \nu) \mid \Sigma \rightsquigarrow c_r[x_r := \nu] \mid \Sigma} \quad \text{(S-HandleInstReturn)}$$

Lastly in Figure 5.10 we give the semantics for the instance handlers. Instance handlers handle operation calls on instances with the same location as the handler. To be able to handle operation calls with one rule we first have to transform operation calls that are not being sequenced to the sequencing form (S-HandleInstOpPrepare). When an operation call is encountered on an instance with the same location as the instance handler, the operation is handled (S-HandleInstOp). The operation is looked up in the handler h and the computation c_{op} in the operation case is performed. If a return is encountered the computation c_r in the return case

is performed (S-HandleInstReturn). Operation calls on instances with a different location l' are skipped, nesting the instance handler inside (S-HandleInstOpSkip). Similarly new calls are also skipped, again nesting the instance handler inside (S-HandleInstNew). Lastly, computations inside instance handlers can be reduced further (S-HandleInstCong).

TODO: add example derivation

5.7 Type soundness

TODO
We define a computation c to be a value if c is of the form return ν for some value ν.

$$\mathsf{value}(c) \text{ if } \exists \nu. c = \mathsf{return}\ \nu$$

Theorem 4 (Type soundness)**.**

$$\text{if } (\cdot; \Sigma; \cdot \vdash c : \tau \,!\, \varnothing) \wedge (c \mid \Sigma \rightsquigarrow^* c' \mid \Sigma') \text{ then } \mathsf{value}(c') \vee (\exists c''\ \Sigma''.\ c' \mid \Sigma' \rightsquigarrow c'' \mid \Sigma'')$$

The type soundness theorem states that if a computation typechecks with no effects in the annotation and we take some amount of steps, then either the computation is a value or we can take another step. This means that if a computation typechecks with no effects then we cannot get stuck on an operation call.

Lemma 7 (Progress)**.**

$$\text{if } \cdot; \Sigma; \cdot \vdash c : \tau \,!\, \varnothing \text{ then } \mathsf{value}(c) \vee (\exists c'\ \Sigma'.\ c \mid \Sigma \rightsquigarrow c' \mid \Sigma')$$

Lemma 8 (Preservation)**.**

$$\text{if } (\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}) \ \wedge\ (c \mid \Sigma \rightsquigarrow c' \mid \Sigma') \text{ then } \Delta; \Sigma'; \Gamma \vdash c' : \underline{\tau}$$

Chapter 6

Related work

- Abstracting Algebraic Effects
- Abstraction-Safe Effect Handlers via Tunneling
- First Class Dynamic Effect Handlers: or, Polymorphic Heaps with Dynamic Effect Handlers
- Algebraic Effect Handlers with Resources and Deep Finalization
- Eff Directly in OCaml
- Programming with Algebraic Effects and Handlers
- An Effect System for Algebraic Effects and Handlers
- Koka: A Language with Row-Polymorphic Effect Inference

Chapter 7

Conclusion and future work

7.1 Future work

- More detailed effect annotations
- Formalization
- Incomplete handlers
- Shallow handlers
- Multi-handlers
- Combine with normal algebraic effects
- Allow escaping effects with exception effect
- Full parametric polymorphism
- Polymorphic effects
- Polymorphic operations

	Eff[1][2]	Links [3]	Koka[4]	Frank[6]	Idris (effects library)[7]
Shallow handlers	No	Yes	Yes	Yes	No
Deep handlers	Yes	Yes	Yes	With recursion	Yes
Effect subtyping	Yes	No	No	No	No
Row polymorphism	No	Yes	Only for effects	No	No
Effect instances	Yes	?	Duplicated labels	No	Using labels
Dynamic effects	Yes	No	Using heaps	No	No
Indexed effects	No	No	No	No	Yes

7.2 Shallow and deep handlers

Handlers can be either shallow or deep. Let us take as an example a handler that handles a *state* effect with *get* and *set* operations. If the handler is shallow then only the first operation in the program will be handled and the result might still contain *get* and *set* operations. If the handler is deep then all the *get* and *set* operations will be handled and the result will not contain any of those operations. Shallow handlers can express deep handlers using recursion and deep handlers can encode shallow handlers with an increase in complexity. Deep handlers are easier to reason about *I think expressing deep handlers using shallow handlers with recursion might require polymorphic recursion.*
Frank has shallow handlers by default, while all the other languages have deep handlers. Links and Koka have support for shallow handlers with a *shallowhandler* construct.

In Frank recursion is needed to define the handler for the state effect, since the handlers in Frank are shallow.

```
state : S -> <State S>X -> X
state _ x = x
state s <get -> k> = state s (k s)
state _ <put s -> k> = state s (k unit)
```

Koka has deep handlers and so the handler will call itself recursively, handling all state operations.

```
val state = handler(s) {
  return x -> (x, s)
  get() -> resume(s, s)
  put(s') -> resume(s', ())
}
```

7.3 Effect subtyping and row polymorphism

A handler that only handles the *State* effect must be able to be applied to a program that has additional effects to *State*. Two ways to solve this problem are effect subtyping and row polymorphism. With effect subtyping

we say that the set of effects set_1 is a subtype of set_2 if set_2 is a subset of set_1.

$$\frac{s_2 \subseteq s_1}{s_1 \leq s_2}$$

With row polymorphism instead of having a set of effects there is a row of effects which is allowed to have a polymorphic variable that can unify with effects that are not in the row. We would like narrow a type as much as we can such that pure functions will not have any effects. With row polymorphic types this means having a closed or empty row. These rows cannot be unified with rows that have more effects so one needs to take care to add the polymorphic variable again when unifying, like Koka does.
Eff uses effect subtyping while Links and Koka employ row polymorphism *Not sure yet about Frank and Idris.*

7.4 Effect instances

One might want to use multiple instances of the same effect in a program, for example multiple *state* effects. Eff achieves this by the *new* operator, which creates a new instance of a specific effect. Operations are always called on an instance and handlers also reference the instance of the operations they are handling. In the type annotation of a program the specific instances are named allowing multiple instances of the same effect.
Idris solves this by allowing effects and operations to be labeled. These labels are then also seen in the type annotations.

In Idris labels can be used to have multiple instances of the same effect, for example in the following tree tagging function.

```
-- without labels
treeTagAux : BTree a -> { [STATE (Int, Int)] } Eff (BTree (Int, a))
-- with labels
treeTagAux : BTree a -> {['Tag ::: STATE Int, 'Leaves ::: STATE Int]} Eff (B
```

Operations can then be tagged with a label.

```
treeTagAux Leaf = do
        'Leaves :- update (+1)
        pure Leaf
treeTagAux (Node l x r) = do
        l' <- treeTagAux l
        i <- 'Tag :- get
        'Tag :- put (i + 1)
        r' <- treeTagAux r
        pure (Node l' (i, x) r')
```

In Eff one has to instantiate an effect with the *new*, operations are called on this instance and they can also be arguments to an handler.

```
type 'a state = effect
  operation get: unit -> 'a
  operation set: 'a -> unit
end

let r = new state

let monad_state r = handler
  | val y -> (fun _ -> y)
  | r#get () k -> (fun s -> k s s)
  | r#set s' k -> (fun _ -> k () s')

let f = with monad_state r handle
    let x = r#get () in
    r#set (2 * x);
    r#get ()
in (f 30)
```

7.5 Dynamic effects

One effect often used in imperative programming languages is dynamic allocation of ML-style references. Eff solves this problem using a special type of effect instance that holds a *resource*. This amounts to a piece of state that can be dynamically altered as soon as a operation is called. Note that this is impure. Haskell is able to emulate ML-style references using the ST-monad where the reference are made sure not to escape the thread where they are

used by a rank-2 type. Koka annotates references and read/write operations with the heap they are allowed to use.

In Eff resources can be used to emulate ML-style references.

```
let ref x =
  new ref @ x with
    operation lookup () @ s -> (s, s)
    operation update s' @ _ -> ((), s')
  end

let (!) r = r#lookup ()
let (:=) r v = r#update v
```

In Koka references are annotated with a heap parameter.

```
fun f() { var x := ref(10); x }
f : forall<h> () -> ref<h, int>
```

Note that values cannot have an effect, so we cannot create a global reference. So Koka cannot emulate ML-style references entirely.

```
> val x = ref(1)
        ^
((4), 5): error: effects do not match
  context         : val x = ref(1)
  term            :     x
  inferred effect: <alloc<_h>|_e>
  expected effect: total
  because         : Values cannot have an effect
```

7.6 Indexed effects

Similar to indexed monad one might like to have indexed effects. For example it can be perfectly safe to change the type in the *state* effect with the *set* operation, every *get* operation after the *operation* will then return a value of this new type. This gives a more general *state* effect. Furthermore we would like a version of typestate, where operations can only be called with a certain state and operations can also change the state. For example closing a file handle can only be done if the file handle is in the *open* state, after which this

state is changed to the *closed* state. This allows for encoding state machines on the type-level, which can be checked statically reducing runtime errors.

Only the effects library Idris supports this feature.

```
data State : Effect where
  Get : { a } State a
  Put : b -> { a ==> b } State ()

STATE : Type -> EFFECT
STATE t = MkEff t State

instance Handler State m where
  handle st Get k = k st st
  handle st (Put n) k = k () n

get : { [STATE x] } Eff x
get = call Get

put : y -> { [STATE x] ==> [STATE y] } Eff ()
put val = call (Put val)
```

Note that the *Put* operation changes the type from a to b. The *put* helper function also shows this in the type signature (going from $STATE\ x$ to $STATE\ y$).

Bibliography

[1] Bauer, Andrej, and Matija Pretnar. "Programming with algebraic effects and handlers." Journal of Logical and Algebraic Methods in Programming 84.1 (2015): 108-123.

[2] Bauer, Andrej, and Matija Pretnar. "An effect system for algebraic effects and handlers." International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.

[3] Hillerström, Daniel, and Sam Lindley. "Liberating effects with rows and handlers." Proceedings of the 1st International Workshop on Type-Driven Development. ACM, 2016.

[4] Leijen, Daan. "Type directed compilation of row-typed algebraic effects." POPL. 2017.

[5] Leijen, Daan. Algebraic Effects for Functional Programming. Technical Report. 15 pages. https://www. microsoft. com/en-us/research/publication/algebraic-effects-for-functional-programming, 2016.

[6] Lindley, Sam, Conor McBride, and Craig McLaughlin. "Do Be Do. In: POPL'2017. ACM, New York, pp. 500-514. ISBN 9781450346603, http://dx. doi. org/10.1145/3009837.3009897."

[7] Brady, Edwin. "Programming and Reasoning with Side-Effects in IDRIS." (2014).

[8] Levy, PaulBlain, John Power, and Hayo Thielecke. "Modelling environments in call-by-value programming languages." Information and computation 185.2 (2003): 182-210.

[9] Flanagan, Cormac, et al. "The essence of compiling with continuations." ACM Sigplan Notices. Vol. 28. No. 6. ACM, 1993.

[10] Biernacki, Dariusz, et al. "Handle with care: relational interpretation of algebraic effects and handlers." Proceedings of the ACM on Programming Languages 2.POPL (2017): 8.

[11] Robin Milner, Mads Tofte, and David Macqueen. 1997. The Definition of Standard ML. MIT Press, Cambridge, MA, USA.ndar

[12] Jones, Simon Peyton, ed. Haskell 98 language and libraries: the revised report. Cambridge University Press, 2003.

[13] Plotkin, Gordon D., and Matija Pretnar. "Handling algebraic effects." arXiv preprint arXiv:1312.1399 (2013).

[14] Launchbury, John, and Simon L. Peyton Jones. "Lazy functional state threads." ACM SIGPLAN Notices 29.6 (1994): 24-35.

[15] Claessen, Koen, and Peter Ljunglöf. "Typed Logical Variables in Haskell." Electr. Notes Theor. Comput. Sci. 41.1 (2000): 37.

[16] Pretnar, Matija. "An introduction to algebraic effects and handlers. invited tutorial paper." Electronic Notes in Theoretical Computer Science 319 (2015): 19-35.