A type system for dynamic instances

Version of July 20, 2019

Albert ten Napel

A type system for dynamic instances

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Albert ten Napel
born in Urk, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

A type system for dynamic instances

Author: Albert ten Napel
Student id: 4087798
Email: a.tennapel@student.tudelft.nl

Abstract

Side-effect are ubiquitous in programming. Examples include mutable state, exceptions, nondeterminism, and user input. Algebraic effects and handlers are an approach to programming that gives a structured way of programming with effects. Each effect in a system with algebraic effects is defined by a set of operations. These operations can then be called anywhere in a program. Using a handler we can give an interpretation for the operations used. Unfortunately we are unable to express dynamic effects such as the dynamic creation of mutable references using regular algebraic effects. Extending algebraic effects with effect instances enabled us to express dynamic effects. These effect instances can be dynamically created and operations called on them are distinct from the same operation called on a different instance. Without a type system these dynamic instances may result in runtime errors, because operations may Because of their dynamic nature it is hard to give a type system for these dynamic instances though. In this thesis we present a new language, Miro, which extends algebraic effects and handlers with a restricted form of effect instances. We introduce the notion of an *effect scope* which encapsulates the creation and usage of dynamically created effect instances. We give a formal description of the syntax and semantics of Miro. We also give a type system which ensures that all operation calls are handled, so that we will not have runtime errors. Because effect instances can still escape their effect scope, in computationally irrelevant parts, we encounter difficulties in proving type safety for Miro. We discuss these difficulties and give possible approaches to prove type safety in the future.

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. N. Yorke-Smith, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. R. Krebbers, Faculty EEMCS, TU Delft

Acknowledgements

I would like to thank my supervisors Robbert and Casper for their invaluable feedback and for the interesting discussions. I would like to thank my parents for their endless love and support. Finally. I give my special thanks to my girlfriend Justyna, for her neverending love, support and patience.

Albert ten Napel
Urk, the Netherlands
July 20, 2019

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 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 (Jones 2003), Eff (Bauer and Pretnar 2015), Koka (Leijen 2016), and Links (Hillerström and Lindley 2016). 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 one 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.

Using type systems we can statically rule out programs that may lead to runtime errors. Type systems can also 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 (Leijen 2016) a function which prints something to the console may be given the type:

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

Where CONSOLE shows the use of the console.

Algebraic effects and handlers (Plotkin and Pretnar 2013) 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 from the impure parts. Users can define effects and easily use them in functions, with different effects composing without any extra effort. Each effect in a system with algebraic effects 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. 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 (pseudo)code defines an effect called *State* which simulates a single mutable state cell:

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

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

handlePostInc : Int
handlePostInc =
  f <- handle (postInc) {
    get _ k -> \st -> k st st
    put newst k -> \st -> k () newst
    return x -> \st -> return x
  };
  f 42
```

The function `postInc` increments the current value in the state cell and returns the previous value. We then handle the calls to get and put of `postInc` in `handlePostInc` using a handler. The handler transforms get and put calls to a function expecting the current state. We name this function `f` and pass it 42 as the initial state.

The problem While algebraic effects and handlers have many of the desirable properties we would like, they are is unable to express multiple mutable references. Mutable references have interesting applications such as meta variables in unification algorithms and typed logic variables (Claessen and Ljunglöf 2000). In the previous example it can be seen that `postInc` does not refer to any specific reference, but instead can only manipulate one ambient reference using the get and put operations. Dynamic instances were introduced by the Eff programming language (Bauer and Pretnar 2015) to solve this problem. With dynamic instances multiple different instances of the same effect can be dynamically created. Using this multiple mutable references can be implemented by dynamically creating instances of the `State` effect (we give an example in Section 2.3). Unfortunately theses dynamic instances can escape the scope of their handler. Any operation called on one of these escaped instances will result in a runtime error, since this operation call will be unhandled. Eff also introduces *resources*, these are effect instances with a globally scoped handler associated. Because the handler is globally scoped the instance can never escape its scope and any operation call will

always be handled. Unfortunately there is no type system given for dynamic instances, so we have no static guarantees that there will be no unhandled operation calls.

In Haskell the the so-called "ST monad" (Launchbury and Peyton Jones 1994) can be used to safely manipulate multiple references in such a way that stateful computations can be encapsulated and that the references are not leaked outside of the function. Mutable references can be dynamically created and manipulated. Computations using these references can be made pure by passing them to a function called `runST`. This function will statically ensure that no references will escape their scope and that the mutation effects are encapsulated.

In this thesis we define a new language named Miro based on algebraic effects and handlers which allows for the definition of effects such as the dynamic creation of mutable references, and the opening of files and channels. Using this system we can implement a system similar to the ST monad from Haskell. We introduce a notion of effect scopes, which encapsulates the creation and usage of dynamically created effect instances. Each function is statically annotated with the effect scopes that are used in the function. Using the effect scopes we statically ensure that effects are encapsulated. We give examples of programs using these side-effects in Miro and show how to implement local mutable references. We give a formal description of the syntax, typing rules and semantics of a core calculus for Miro.

Proving type safety for Miro turns out to be more difficult than anticipated. It is common to prove type safety by first proving a type preservation lemma. Type preservation states that if a term is well-typed and if we take a step (using the semantics) then the resulting term is also well-typed (with the same type as before). Our language introduces some intermediate forms used by the small-step operational semantics. These intermediate forms are introduced by the semantics during the process of evaluation and do not appear in the source language. In order to prove the preservation lemma one also has to give typing rules for these intermediate forms. We will call these the *dynamic* typing rules. The difficulty in coming up with dynamic typing rules is that effect instances can still escape their effect scope if they are not used and not returned from a function. The dynamic typing rules have to give these escaped instance a valid type, but since they have escaped their effect scope this is tricky because the type of an instance depends on the effect scope, which it has escaped from. The escaped instances do however have no computational effect and so we conjecture that type safety still holds. Type safety might still be provable in other ways, given correct dynamic typing rules. In this thesis we will show the problems with proving type preservation for Miro and we will discuss other approaches that may allow us to prove type safety.

Contributions

- **Language.** We define a language named Miro based on algebraic effects and handlers that can handle a form of dynamic effect instances.
- **Mutable references.** We give examples in Miro that would be difficult or impossible to express with ordinary algebraic effects.
- **Operational semantics and type system.** We define a core calculus of Miro together with a small-step operational semantics and a type system.
- **Research on type safety for dynamic instances.** We show the problems we encountered when trying to prove type safety for Miro. We discuss the other approaches that may allow us to prove type safety.
- **Formalizations.** We have formalized algebraic effects and handlers with and without static instances and have proven type safety in Coq[1].

[1] https://github.com/atennapel/dynamicinstances

Thesis structure

The thesis is structured as follows. Chapter 2 gives an introduction to ordinary algebraic effects and handlers, and static and dynamic instances. Chapter 3 gives an introduction to our new language Miro. Chapter 4 gives formal definitions of systems with ordinary algebraic effects and handlers, and static instances. Chapter 5 gives a formal account of Miro, we also discuss the problem with proving type safety. 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 will not do anything else but compute, it will not 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. So we would like 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 (Section 2.1). Afterwards we will continue with static instances (Section 2.2) which allows for multiple static instances of the same effect to be used in a program. We end with dynamic instances (Section 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 similar to Koka (Leijen 2016).

2.1 Algebraic effects and handlers

We will start with 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 instead. 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#ref (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 (Milner, Tofte, and Harper 1990), 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 (Launchbury and Peyton Jones 1994) 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
```

The problem with dynamic instances 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. For example:

```
escapeRef =
  heap <- new Heap;
  escapedRef <- runRefs heap (
    r <- heap#ref 42;
    return r
  );
  escapedRef#get () // unhandled operation call!
```

Here we create a new heap and handle a computation on it using `runRefs`. The computation creates a new reference and returns it. After `runRefs` is done the reference is returned and named `escapedRef`. This reference has no handler associated with it anymore. We then call the `get` operation resuling in an unhandled operation call which is a runtime error.

Because we cannot statically know whether we are calling an operation on an escaped instance, it is 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 (Bauer and Pretnar 2015) 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.

In the next chapter (Chapter 3) we will introduce our new language Miro. By restricting how dynamic instances can be created and handled we are able to give a type system which ensures that all operation calls are handled.

Chapter 3

Introduction to Miro

In Section 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 for the instance, which we gave an example of. In this chapter we introduce our new language Miro, which combines algebraic effects and a restricted form of dynamic instances. In order to ensure all operation calls are handled 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 Section 2.3, we always have to specify a handler when creating an instance, similar to resources in Eff. Specifying a handler at the moment of creating an instance ensures that each instance always 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. The construct `runscope` creates a fresh scope and makes it available for use in a given computation. We can use the newly-created scope to dynamically create effect instances in the computation. We always have to give a handler when creating an effect instance. After `runscope` will ensure that all operation calls are handled and that the effects are encapsulated. 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 Section 2.1. We use syntax 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 Miro.

3.1.1 Effects

To start off we define a `State` effect specialized to `Int`s. The `State` effect 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 Section 2.1, in the basic algebraic effects system.

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
```

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. 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 Section 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 intial value `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 (Launchbury and Peyton Jones 1994).

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. The type `STRef s Int` is the type of a mutable reference in the ST monad in Haskell. The type variable `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.

We can also 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.

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. The function `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`.

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. The `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 scopes. 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. The type `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.

Figure 3.3: Vector shuffling

```
1  -- random number generation effect
2  -- the operation `rand` gives back a random integer
3  -- between 0..n, where n is the argument given (exclusive)
4  effect Rng {
5    rand : Int -> Int
6  }
7
8  -- shuffles a vector given an instance of Rng
9  -- by swapping two random elements of the vector
10 -- the second argument to shuffleVector is the amount of times
11 -- to swap elements
12 shuffleVector : forall s s'. Inst s' Rng -> Int
13   -> Vector s -> ()!{s, s'}
14 shuffleVector [s] [s'] _ 0 vec = vec
15 shuffleVector [s] [s'] rng n vec =
16   let len = vlength vec;
17   i <- rng#rand(len);
18   j <- rng#rand(len);
19   a <- vget [s] i vec;
20   b <- vget [s] j vec;
21   vset [s] i b vec;
22   vset [s] j a vec;
23   shuffleVector [s] [s'] rng (n - 1) vec
```

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. The effect `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.

The function `shuffleVector` proceeds as follows. If the amount of times we want to shuf-

fle 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 3.4: 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 (vlength vec) vec;
30     return (toList 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. The function `toVector` simply recurses on the list and creates fresh mutable references for each element of the list, initialized with the value of the element. The function `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 n times (where n is the length of the vector), 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`.

Figure 3.5: Running list shuffling

```
1  runshuffle : List -> List
2  runshuffle lst =
3    runscope(s ->
4      seedref <- ref [s] 123456789;
5      rng <- new Rng@s {
6        rand n k ->
7          currentseed <- seedref#get();
8          let newseed = (1103515245 * currentseed + 12345) % n;
9          seedref#put(newseed);
10         k newseed
11       return x -> return x
12       finally x -> return x
13     };
14     shuffle [s] rng lst)
```

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.

In this chapter we have seen how to program with effect scopes. Like the regular algebraic effects (Section 2.1) we can use and combine different effects simply by using their operations in a program. What is different is that handlers are given when creating instances. We have seen that we can abstract over and instantiate effect scopes. Lastly we saw how effect scopes enable use to implement mutable references and vectors while still being safe.

Chapter 4

Semantics and types of algebraic effects and handlers

In this chapter we will give a theoretical basics for algebraic effects and handlers as introduced in Chapter 2. We do this in order to ease the reader in to the theoretical calculus for Miro (given in Chapter 5) which is based on the calculus for algebraic effects given in this chapter. We will start with the simply-typed lambda calculus (Section 4.1) and then add algebraic effects (Section 4.2) and static instances (Section 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) (Levy, Power, and Thielecke 2003). 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 (Flanagan et al. 1993).
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.

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. 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.

Figure 4.1: Syntax of the fine-grained lambda calculus

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

Figure 4.2: Semantics of the fine-grained lambda calculus

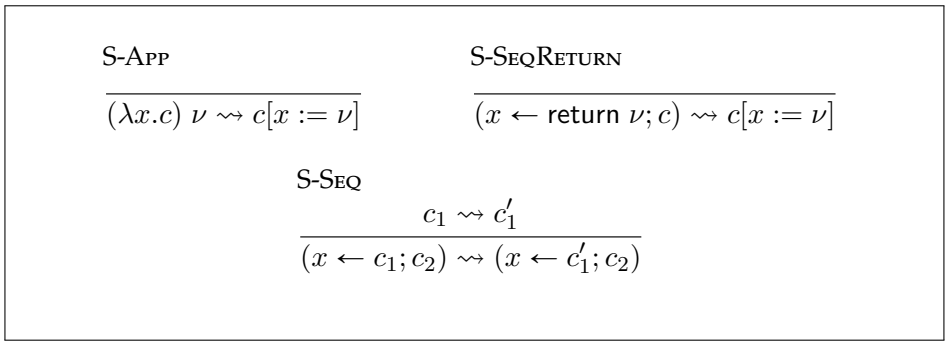
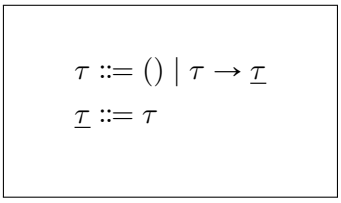


Figure 4.3: Types of the fine-grained simply-typed lambda calculus



Computations For any value ν we have return ν 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 is no choice of evaluation order. The order is made explicit by the sequencing syntax $(x \leftarrow c; c)$.

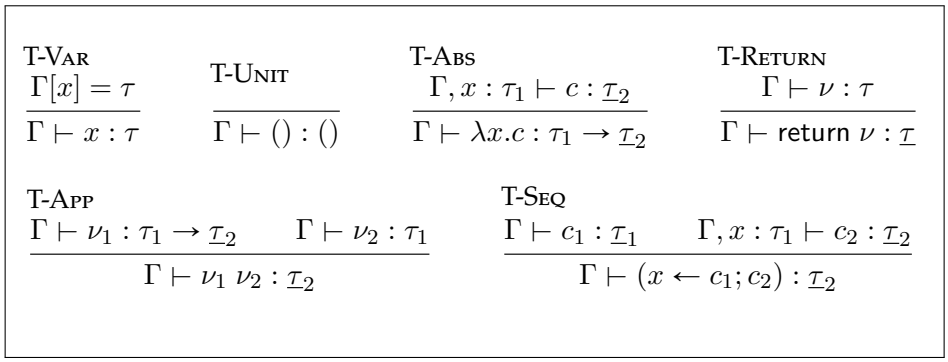
Semantics The small-step operational semantics is shown in Figure 4.2. The relation $\rightsquigarrow$ is defined on computations, where the $c \rightsquigarrow c'$ means c reduces to c' in one step. These rules are a fine-grained approach to the standard reduction rules of the simply-typed lambda calculus. 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.

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.

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



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.

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 return ν (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.

Type safety In order to prove type safety 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 return ν for some value ν.

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

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

Theorem 1 (Type safety)**.**

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

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)**.**

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

Lemma 2 (Preservation)**.**

$$\text{if } \Gamma \vdash c : \underline{\tau} \text{ and } c \rightsquigarrow c' \text{ then } \Gamma \vdash c' : \underline{\tau}$$

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.

We formalized the fine-grained simply-typed lambda calculus and have proven the type safety theorem in the Coq proof assistant. We briefly discuss the formalization in Section 4.4.

4.2 Algebraic effects

We now extend the previous calculus with algebraic effects and handlers. We assume there is a set of effect names 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 $O^{\mathsf{State}} = \{\mathsf{get}, \mathsf{put}\}$.

Figure 4.5: Syntax of algebraic effects

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

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 $op\ x\ k \rightarrow c;\ h$ ending in the return case $\mathsf{return}\ x \rightarrow c$. 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. Because rewrap the handler h in the continuation we implement what are called *deep* handlers. Another approach is to omit h from the continuation, which are called *shallow* handlers (Hillerström and Lindley 2018). We chose to define deep handlers because shallow handlers require explicit recursion to implement the example handlers we gave in Chapter 2. If the operation op is not contained in the handler then we 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

Figure 4.6: Semantics of algebraic effects

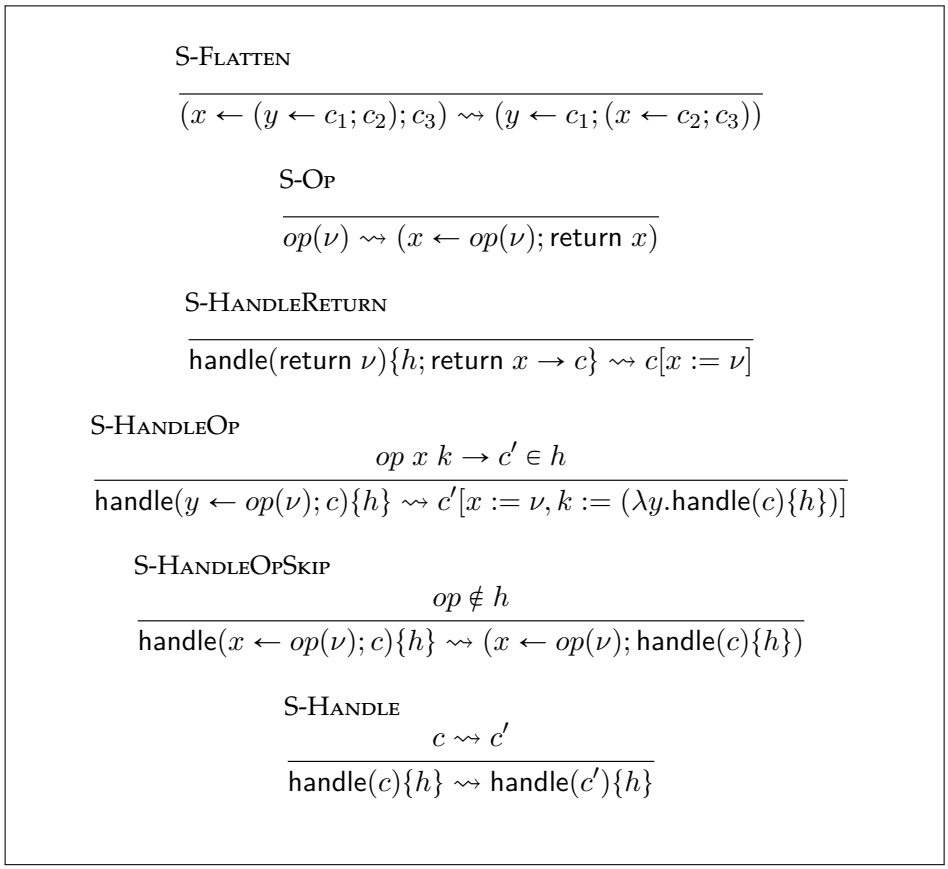


Figure 4.7: Types of algebraic effects

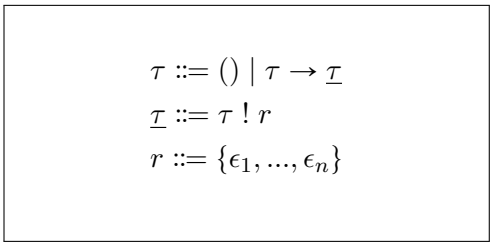


Figure 4.8: Subtyping rules of algebraic effects

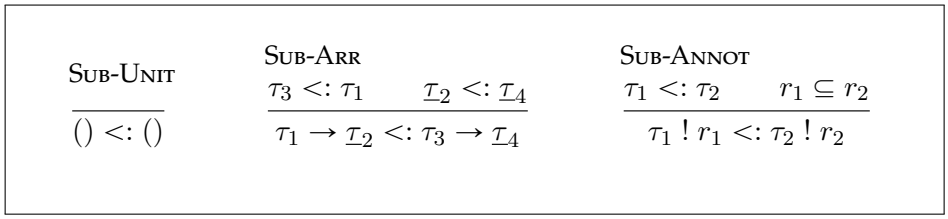
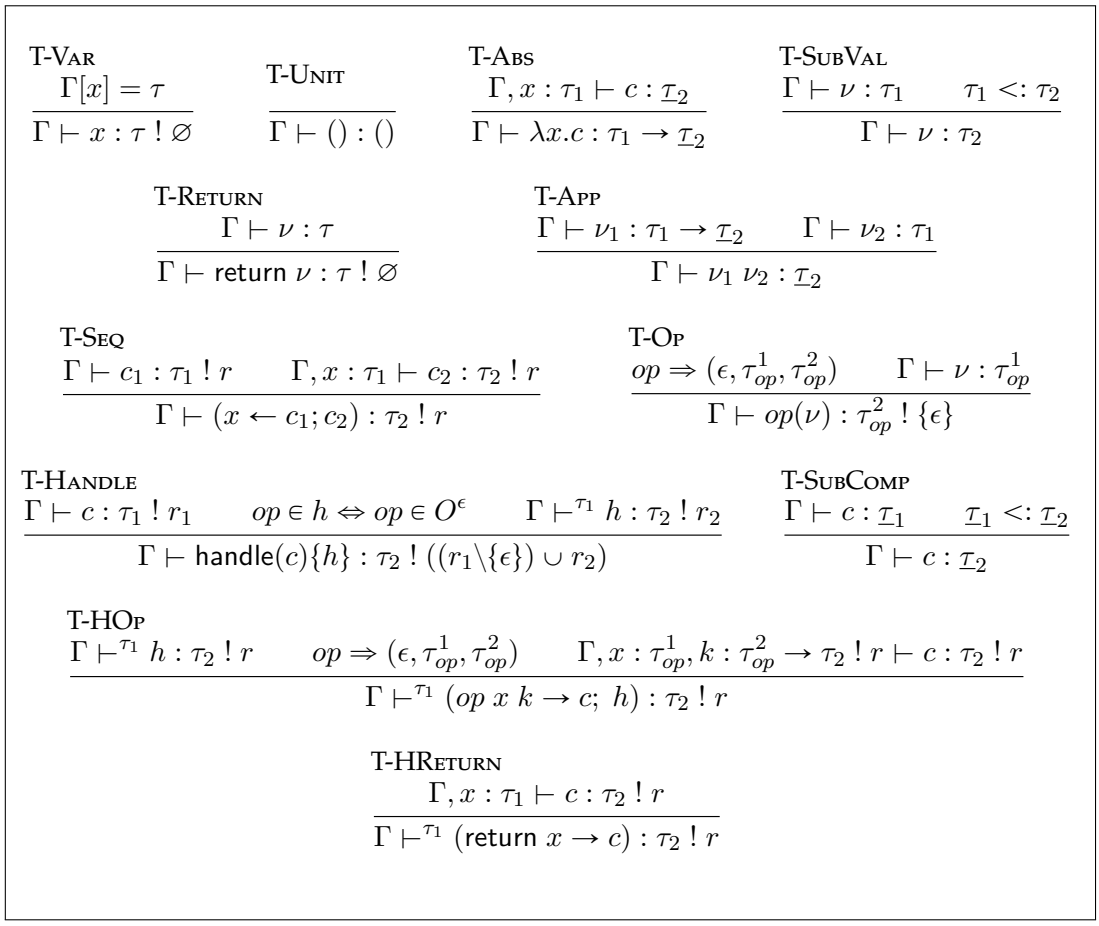


Figure 4.9: Typing rules of algebraic effects



form return ν 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 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 4.8 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 4.9. 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 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 safety We can defined a type safety theorem very similar to the one for the simply-typed lambda calculus.

Theorem 2 (Type safety)**.**

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

The theorem states that if a computation c is typed in the empty context with an empty effect annotation and we take some amount of steps, then the computation is value or we can take another step. This theorem only works with the empty effect annotation, because if an effect is in the annotation then there may be an unhandled operation call and the computation would not be able to proceed.

We can prove the type safety theorem using progress and preservation lemmas. We first give the preservation lemma, which is exactly the same as the one from the simply-typed lambda calculus.

Lemma 3 (Preservation)**.**

$$\text{if } \Gamma \vdash c : \underline{\tau} \text{ and } c \rightsquigarrow c' \text{ then } \Gamma \vdash c' : \underline{\tau}$$

The progress lemma differs only in that the computation cannot have any effects. Again because we could get stuck if any operation calls are unhandled.

Lemma 4 (Progress)**.**

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

We need a more general version of the progress lemma in order to get a strong enough induction hypothesis to be able to prove it. We define a computation c to be effectful if c has unhandled operations. Formally we define this as follows:

$$\mathsf{effectful}(c) \triangleq \left(\exists op\ \nu.c = op(\nu)\right) \text{ or } \left(\exists x\ op\ \nu\ c'.c = x \leftarrow op(\nu); c'\right)$$

We can now give the generalized version of the progress lemma:

Lemma 5 (Progress effectful)**.**

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

This lemma generalizes $\tau \,!\, \varnothing$ to $\tau \,!\, r$ in the Progress lemma above. This allows c to be typed with any effect annotation, by adding the possibility that c has unhandled operations. We need this generalization or else the induction hypothesis is too weak and we would get stuck if we used any effects.

We formalized the algebraic effect and handler system and have proven the type safety theorem in the Coq proof assistant. We briefly discuss the formalization in Section 4.4.

4.3 Static instances

Finally we extend algebraic effects with static instances. Adding static instances brings the system one step closer towards the calculus of Miro, as we will see in Chapter 5. 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.

Figure 4.10: Syntax of algebraic effects with static instances

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

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.

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

Figure 4.11: Semantics of algebraic effects with static instances

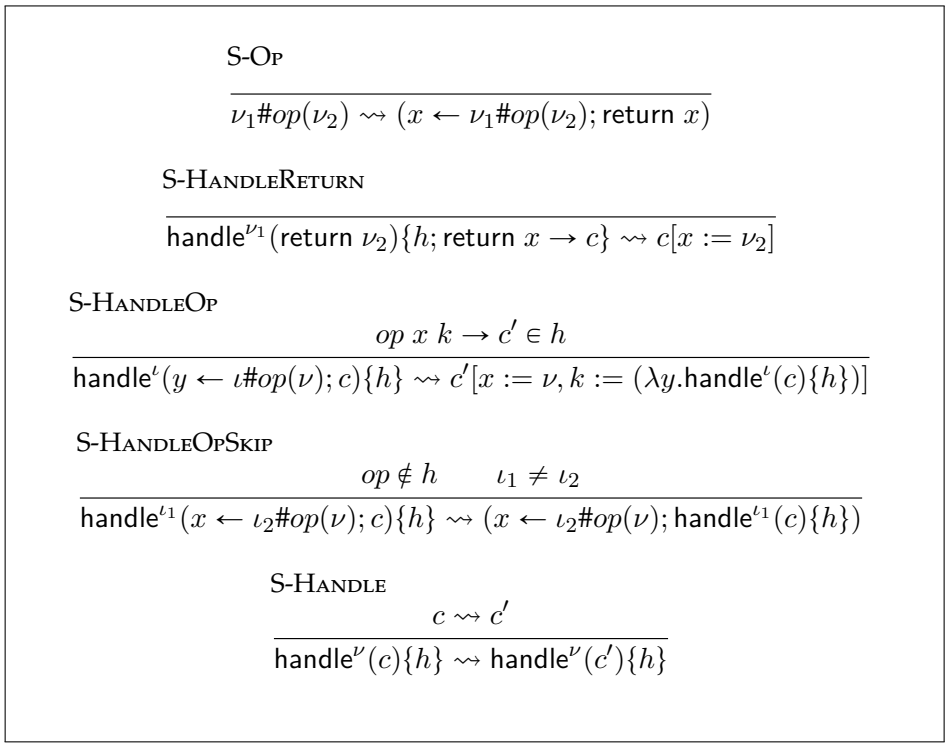


Figure 4.12: Types of algebraic effects with static instances

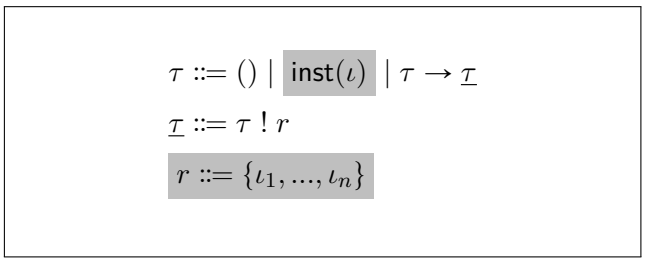


Figure 4.13: Subtyping rules of algebraic effects with static instances

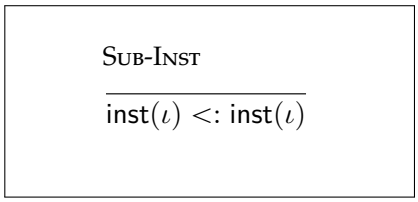
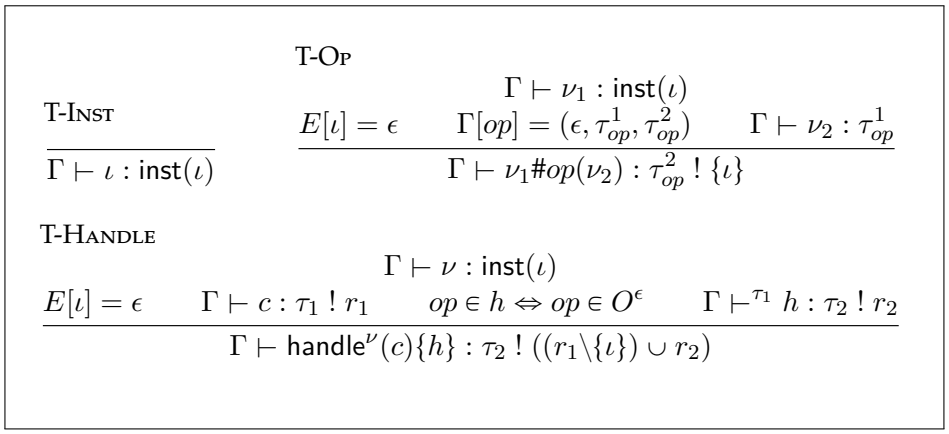


Figure 4.14: Typing rules of algebraic effects with static instances



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 safety We give a type safety theorem for the system with static instances.

Theorem 3 (Type safety)**.**

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

This theorem is exactly the same as the one for algebraic effects in Section 4.2. We can reuse the notion of a computation being a value value without any modifications.

Again we use progress and preservation lemmas to prove the type safety theorem. These lemmas are exactly the same as the ones in Section 4.2. We only need to slightly update our effectful notion, because operation calls are now called on instances.

$$\mathsf{effectful}(c) \triangleq \left(\exists \iota\ op\ \nu.c = \iota \# op(\nu)\right) \text{ or } \left(\exists \iota\ x\ op\ \nu\ c'.c = x \leftarrow \iota \# op(\nu); c'\right)$$

We formalized the system with static instances and have proven the type safety theorem in Coq. We will now discuss the formalizations of the different systems.

4.4 Formalization

We have formalized and proven type safety for all the three systems discussed in this chapter in Coq[1]. We used DeBruijn indices to deal with naming and substitution. We also restrict effects to only have one operation, modeled as a natural number. Handlers can only handle a single operation. The system formalized slightly differs in that we follow the syntax from Bauer and Pretnar (Bauer and Pretnar 2014). Operation calls are of the form $op(\nu, x.c)$, carrying around a continuation. Also handlers are first-class values of type $\underline{\tau} \Rightarrow \underline{\tau}$. These changes make the proofs easier. Having continations inside operation calls means we can move operation calls over sequencing, so the effectful predicate only has one form:

$$\mathsf{effectful}(c) \triangleq \exists op\ \nu\ c'.c = op(\nu, x.c')$$

Adding static instances did not result in much added complexity.

[1] https://github.com/atennapel/dynamicinstances

Chapter 5

Semantics and types of Miro

In this chapter we give a formal account of Miro. We give the syntax, typing rules and a small-step operation semantics. We end the chapter with a discussion on the difficulties of proving type safety for Miro. The system builds on the formal system with algebraic effects, handlers and static instances of Section 4.3. We add constructs to handle effect scope polymorphism, to create new instances, and to handle effect scopes, as informally described in Chapter 3.

In Section 5.1 we give the syntax of the terms and types of Miro. 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 Miro. Finally in Section 5.7 we discuss the problems we encountered when trying to prove type safety.

5.1 Syntax

Just like in the formal systems of algebraic effects of Section 4.2 and Section 4.3 we assume there is set of effect names $\mathsf{EffName}$ with $E \subseteq \mathsf{EffName}$. For example $E = \{\mathsf{Flip}, \mathsf{State}, \mathsf{Exc}, ...\}$. There is also a set of operation names O. Each effect $\varepsilon \in E$ has a non-empty set of operation names $O^{\varepsilon} \in O$. 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 has a parameter type τ^1_{op} and a return type τ^2_{op}.

In Figure 5.1 we show the syntax of the types and terms of Miro. We introduce some intermediate forms which are introduced by the semantics but do not appear in the source language. We color these forms with a gray background.

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 parameter type is a value type and the return type is a computation type; or a universally quantified computation type $\forall s_{var}.\underline{\tau}$, where the

Figure 5.1: Syntax

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

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 instance location l, which are modeled by some countable infinite set. Instances would not appear in the surface language, but are introduced by the semantics. Values can also be 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.

As usual, 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 intermediate 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 $\mathsf{runinst}^{l}_{s_{loc},\varepsilon}\{h\}(c)$ handle the operations of a single instance of the location l in the computation c, we also keep track of the associated scope location s_{loc} and effect ε. We will discuss these in Section 5.6.

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

We will now give the environment and judgments used in the typing rules and semantics.

Environments The syntax for the environments is shown in Figure 5.2.

Figure 5.2: Environments

$$
\begin{aligned}
\Gamma &::= \cdot \mid \Gamma, x : \tau \\
\Delta &::= \cdot \mid \Delta, s_{var} \mid \Delta, s_{loc} \mid \Delta, l := (s_{loc}, \varepsilon) \\
\sigma &::= \cdot \mid \sigma, s_{loc} \mid \sigma, l
\end{aligned}
$$

- Γ is the *typing enviroment*, which assigns variables x to value types τ.
- Δ is the *static environment*, which keeps track of the scope variables s_{var}, scope locations s_{loc}, and instance locations l that are in use. For instance locations we also keep track of the associated scope location s_{loc} and effect ε. Because instance locations depend on scope locations we introduce a well-formedness judgment for Δ in Section 5.4;
- σ is the *dynamic environment*, which keeps track of scope location s_{loc} and instance locations l. The dynamic environment is used to generate fresh scope and instance locations. This environment is only used by the semantics.

Judgments There are four kinds of judgments: subtyping (Section 5.3), well-formedness (Section 5.4) and typing judgments (Section 5.5).

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 $\underline{\tau}'$.

We have a well-formedness judgments for the static environment Δ, scopes s, and value and computation types τ and $\underline{\tau}$. The well-formedness judgments have the following forms:

- $\vdash \Delta$ asserts that all instance locations in Δ refer to valid scope locations and that all scope variables and scope and instance locations are unique.
- $\Delta \vdash s$ asserts that the scope s exists in Δ.
- $\Delta \vdash \tau$ asserts that all the scopes in the value type τ are valid under Δ.
- $\Delta \vdash \underline{\tau}$ asserts that all the scopes in the computation type $\underline{\tau}$ are valid under Δ.

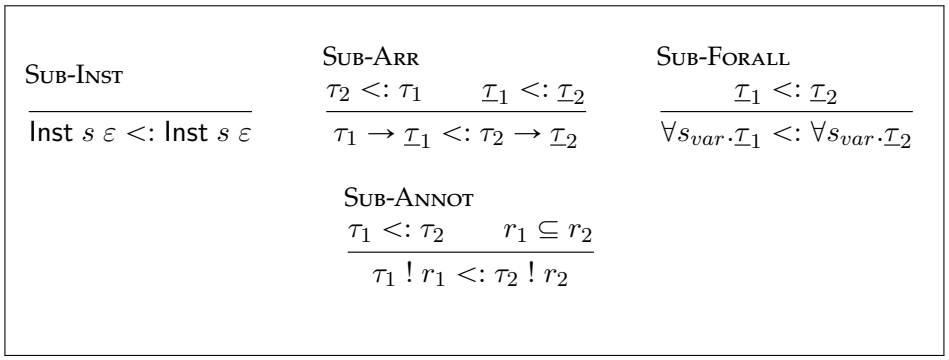
Lastly, there are three typing judgments:

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

5.3 Subtyping

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

Figure 5.3: Subtyping



general than the type on the left. An instance type 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).

5.4 Well-formedness

Figure 5.4: Well-formedness for Δ

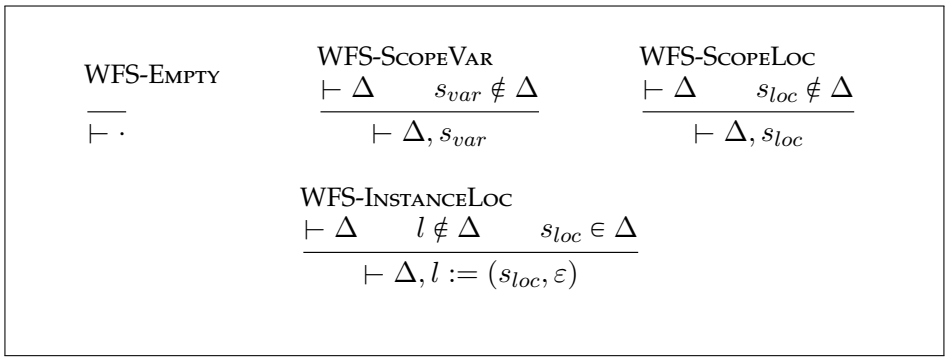
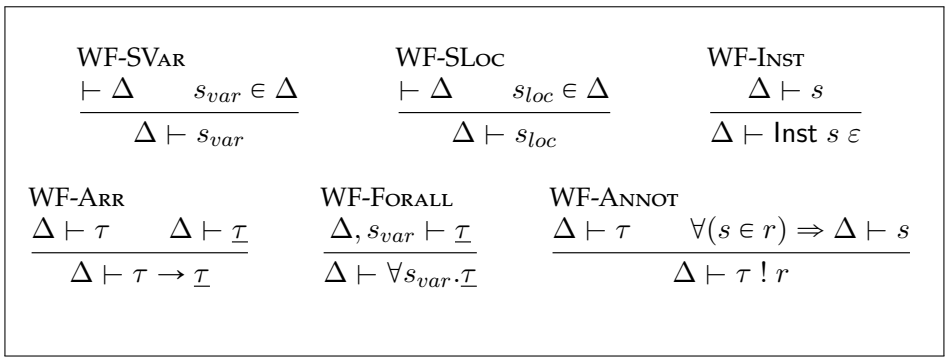


Figure 5.5: Well-formedness for scopes and types



In Figure 5.4 we give the well-formedness rules for the static environment Δ. If Δ is empty then it is well-formed (WFS-Empty). For scope locations s_{loc} we check that the rest of Δ is well-formed and that s_{loc} does not occur in it, meaning that s_{loc} is unique (WFS-ScopeLoc).

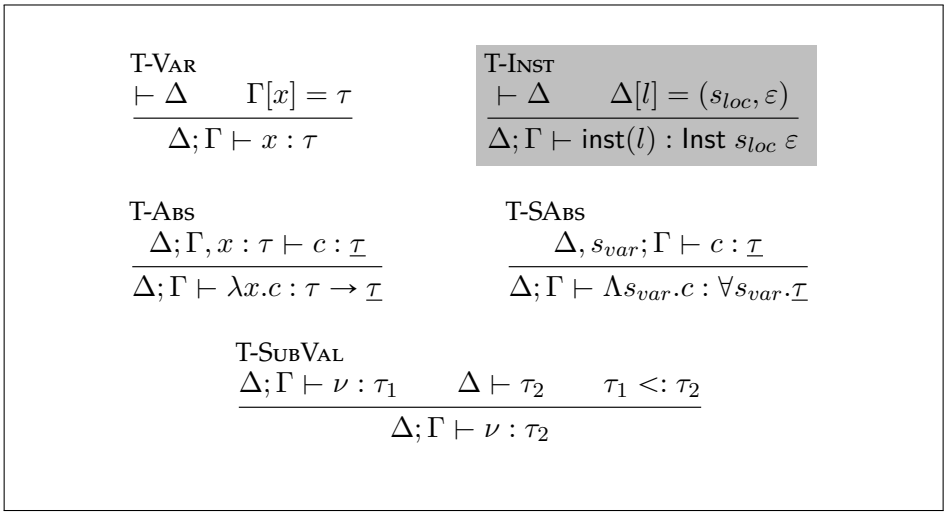
For instance locations l we also check that the rest of Δ is well-formed and that l is unique. Lastly, we check that the scope location s_{loc} used by l is defined in Δ.

In Figure 5.5 we give the well-formedness rules for the value and computation types. Well-formedness asserts that the effect scopes in the type are accounted for in Δ. The rules WF-SVar and WF-SLoc assert that the effect scope variables s_{var} and locations s_{loc} are valid by checking that they are contained in the static environment Δ. 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 parameter 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 values, computations and handlers are given in Figure 5.6, Figure 5.7 and Figure 5.8, respectively. We call the typing rules for the intermediate forms (as given in Section 5.1) the *dynamic typing rules*. As we will discuss in Section 5.7 these typing rules are likely incomplete or incorrect. We color the dynamic typing rules with a gray background.

Figure 5.6: Value typing rules



The typing rules for the values are given in Figure 5.6. The rules T-Var, T-Abs and T-SubVal are practically unchanged from the corresponding rules in the algebraic effects type system from Section 4.2. Instances $\mathsf{inst}(l)$ are assigned instance types, with the scope location and effect looked up in the static environment Δ 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 Δ (T-TAbs).

The typing rules for the computations are given in Figure 5.7. 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 Section 4.2.

The rule for effect scope application (T-TApp) asserts 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.

Figure 5.7: Computation typing rules

T-Return
$$\frac{\Delta;\Gamma \vdash \nu : \tau}{\Delta;\Gamma \vdash \mathsf{return}\ \nu : \tau\ !\ \varnothing}$$

T-App
$$\frac{\Delta;\Gamma \vdash \nu_1 : \tau \to \underline{\tau} \qquad \Delta;\Gamma \vdash \nu_2 : \tau}{\Delta;\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}}$$

T-SApp
$$\frac{\Delta \vdash s \qquad \Delta;\Gamma \vdash \nu : \forall s'_{var}.\underline{\tau}}{\Delta;\Gamma \vdash \nu\ [s] : \underline{\tau}[s'_{var} := s]}$$

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

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

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

T-Runscope
$$\frac{\Delta, s_{var};\Gamma \vdash c : \tau\ !\ r \qquad s_{var} \notin \tau}{\Delta;\Gamma \vdash \mathsf{runscope}(s_{var} \to c) : \tau\ !\ (r \backslash \{s_{var}\})}$$

T-RunscopeLoc
$$\frac{s_{loc} \notin \Delta \qquad \Delta, s_{loc};\Gamma \vdash c : \tau\ !\ r \qquad s_{loc} \notin \tau}{\Delta;\Gamma \vdash \mathsf{runscope}^{s_{loc}}(c) : \tau\ !\ (r \backslash \{s_{loc}\})}$$

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

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

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

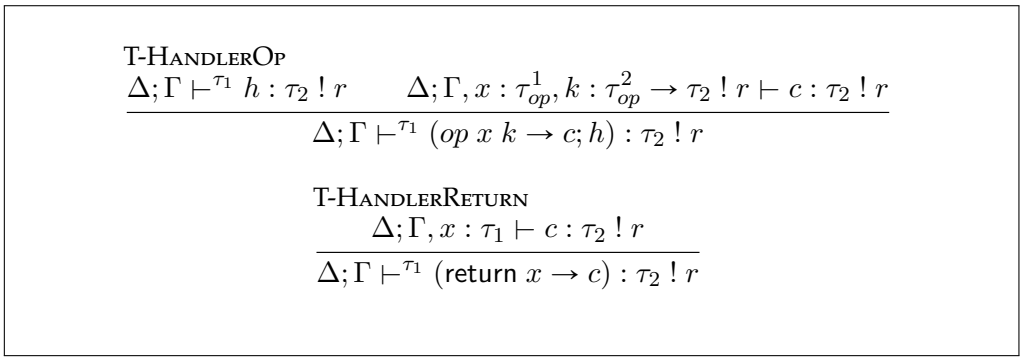
operations of the effect of the new instance ε. This means it is not valid to either omit operations or to have operations of other effects in the handler. This way we can ensure that every operation is accounted for. We then typecheck the computation c with the new instance added the to the environment Γ 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 finally case. We add τ_2 to the environment as y and typecheck the computation c' as $\tau_1 \,!\, r$. Note that the finally case is not allowed to return a different type than the computation c. Finally the type of the whole instance creation is $\tau_1 \,!\, r$.

The rule T-Handle types handling computations: $\mathsf{runscope}(s_{var} \rightarrow c)$. 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 $\tau \,!\, (r \backslash \{s_{var}\})$. 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 first check that s_{loc} is not contained in Δ, if this is the case then we would have duplicate scope locations. We then proceed like in T-Handle, checking that s_{loc} does not escape.

The rule T-HandleInst typechecks the handling of an instance at location l using a handler h. First we check that l is unique by checking it is not contained in Δ. Then we check that the scope location s_{loc} is well-formed under Δ. 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, after adding l to Δ with the associated scope location s_{loc} and effect ε.

Figure 5.8: Handler typing rules



Finally we discuss the typing rules for the handlers, given in Figure 5.8. 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_{op}^1 of the operation to the environment. We also add the continuation k to the environment. This is a function from the return type τ_{op}^2 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 Miro

The judgment $c_1 \underset{\sigma}{\rightsquigarrow} c_2$ takes a step from the computation c_1 to c_2 in the dynamic environment σ. The environment σ is used as a stack of in-scope scope locations and instance locations, and is also used to generate fresh locations. σ will always be monotically extended by $\rightsquigarrow$.

Figure 5.9: Semantics

S-App

$$\frac{}{(\lambda x.c)\ \nu \underset{\sigma}{\rightsquigarrow} c[x := \nu]}$$

S-SApp

$$\frac{}{(\Lambda s_{var}.c)\ [s'] \underset{\sigma}{\rightsquigarrow} c[s_{var} := s']}$$

S-Seq

$$\frac{c_1 \underset{\sigma}{\rightsquigarrow} c_1'}{(x \leftarrow c_1;\ c_2) \underset{\sigma}{\rightsquigarrow} (x \leftarrow c_1';\ c_2)}$$

S-SeqReturn

$$\frac{}{(x \leftarrow (\mathsf{return}\ \nu);\ c) \underset{\sigma}{\rightsquigarrow} c[x := \nu]}$$

S-Flatten

$$\frac{}{(y \leftarrow (x \leftarrow c_1;\ c_2);\ c_3 \underset{\sigma}{\rightsquigarrow} (x \leftarrow c_1;\ y \leftarrow c_2;\ c_3)}$$

S-LiftNew

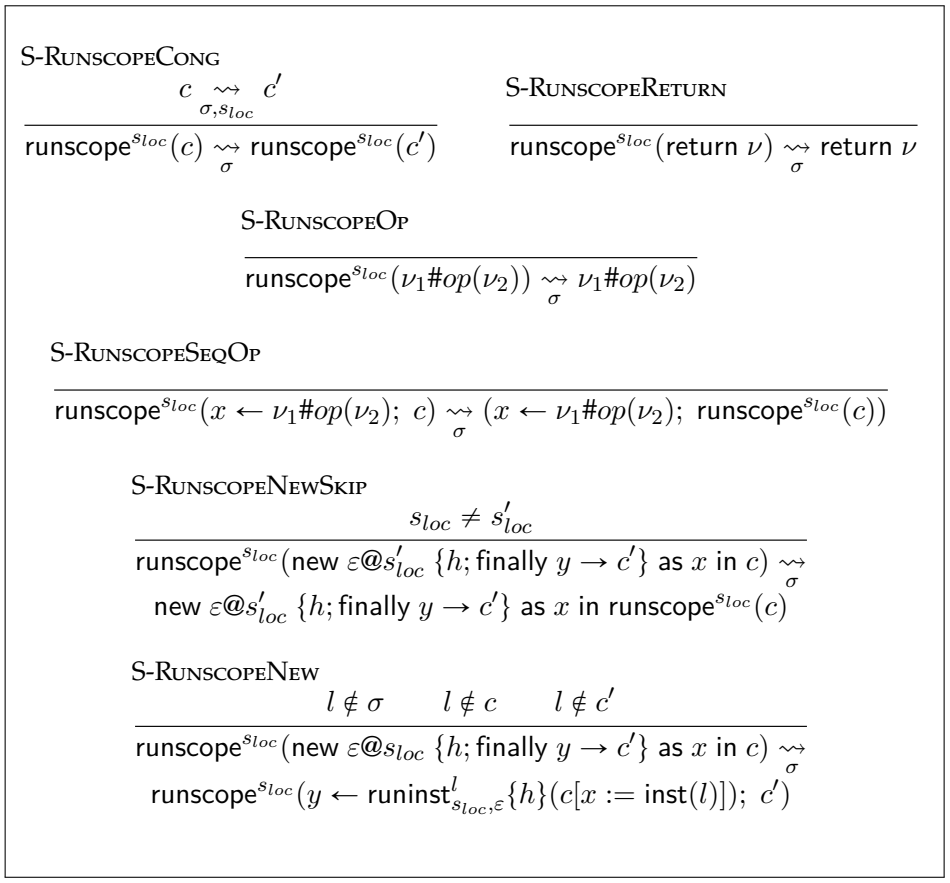
$$\frac{}{(x \leftarrow (\mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ z \to c_3\}\ \mathsf{as}\ y\ \mathsf{in}\ c_1);\ c_2) \underset{\sigma}{\rightsquigarrow} \mathsf{new}\ \varepsilon@s\ \{h; \mathsf{finally}\ z \to c_3\}\ \mathsf{as}\ y\ \mathsf{in}\ (x \leftarrow c_1;\ c_2)}$$

S-Runscope

$$\frac{s_{loc} \notin \sigma \qquad s_{loc} \notin c}{\mathsf{runscope}(s_{var} \to c) \underset{\sigma}{\rightsquigarrow} \mathsf{runscope}^{s_{loc}}(c[s_{var} := s_{loc}])}$$

In Figure 5.9 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 Section 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-Runscope creates a fresh scope location with which to handle instances. A fresh scope location s_{loc} is created, we aseert that s_{loc} is fresh by checking it is not contained in σ or c. This new scope location is substituted in the body c for s_{var}. We then transform $\mathsf{runscope}(s_{var} \to ...)$ to the intermediate form $\mathsf{runscope}^{s_{loc}}(...)$, which is tagged with the scope location s_{loc} which it will handle.

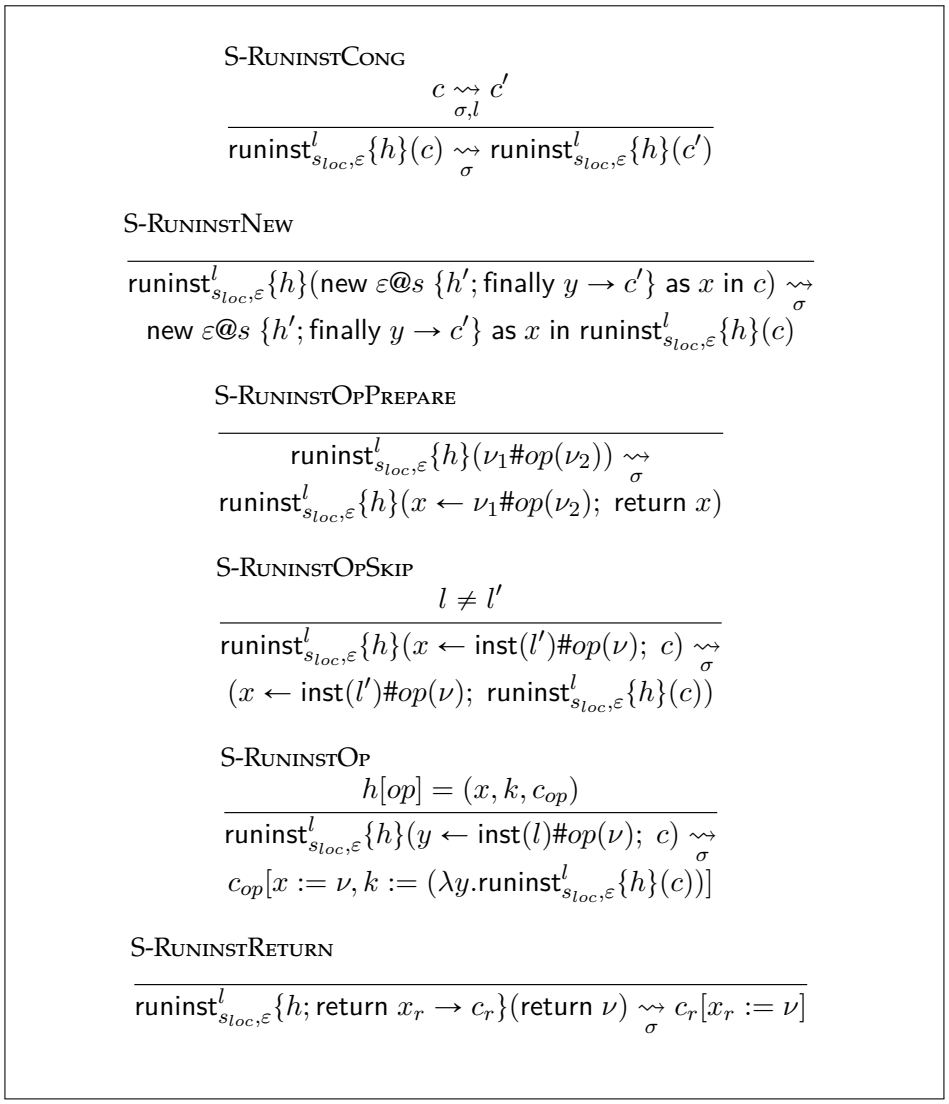
Figure 5.10: Semantics of effect scope handlers



In Figure 5.10 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-RunscopeNew). 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 asserted to be fresh by checking that it is not contained in σ, c and c'. If a new is encountered with a different effect scope we skip it and nest the scope handler inside (T-RunscopeNewSkip). Using the rule S-RunscopeCong we can reduce a computation inside a scope handler. We add the scope location s_{loc} to σ to ensure newly created scope location are different from s_{loc}. We can remove a scope handler if we encounter either a return or operation call (S-RunscopeReturn and S-RunscopeOp). Effect scope handlers can be pushed inside sequencing, lifting an operation call over it (S-RunscopeSeqOp).

Lastly, in Figure 5.11 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-RuninstOpPrepare). When an operation call is encountered on an instance with the same location as the instance handler, the operation is handled (S-RuninstOp). 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-RuninstOpSkip). Similarly new calls are also skipped, again nesting the instance handler inside (S-RuninstNew). Lastly, computations inside instance handlers can be reduced further (S-RuninstCong). Similarly to scope locations, we add the instance location l to σ in order to ensure newly created locations are different from l.

Figure 5.11: Semantics of instance handlers



Example semantics derivation We now give an example derivation of a small program creating and manipulating a single State instance. The program creates a new State on the scope s called r, retrieves the current value, adds 1 and returns the new value. The instance is handled using the standard State handler, which we call h. We give the used semantics rules on the right for every step. When multiple rules are given we mean a nesting of rules. For example (S-RunscopeCong, S-SeqReturn) means we use S-SeqReturn on the computation inside of $\mathsf{runscope}^{s_{loc}}(c)$.

$$
\begin{aligned}
h = \\
&\quad get\ x\ k \to \mathsf{return}\ (\lambda st.f \leftarrow k\ st;\ f\ st);\\
&\quad put\ x\ k \to \mathsf{return}\ (\lambda st.f \leftarrow k\ ();\ f\ x);\\
&\quad \mathsf{return}\ x \to \mathsf{return}\ (\lambda st.\mathsf{return}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{runscope}(s \to \mathsf{new\ State@}s\ \{h;\mathsf{finally}\ f \to f\ 0\}\ \mathsf{as}\ r\ \mathsf{in}\\
&\quad x \leftarrow r\#\mathsf{get}();\ \_ \leftarrow r\#\mathsf{put}(x+1);\ r\#\mathsf{get}())
\end{aligned}
$$

$$\mathsf{runscope}(s \to \mathsf{new\ State@}s\ \{h; \mathsf{finally}\ f \to f\ 0\}\ \mathsf{as}\ r\ \mathsf{in}$$
$$\quad x \leftarrow r\#\mathsf{get}();\ \_ \leftarrow r\#\mathsf{put}(x+1);\ r\#\mathsf{get}())$$
$$\rightsquigarrow \qquad \text{(S-RunScope)}$$

$$\mathsf{runscope}^s(\mathsf{new\ State@}s\ \{h; \mathsf{finally}\ f \to f\ 0\}\ \mathsf{as}\ r\ \mathsf{in}$$
$$\quad x \leftarrow r\#\mathsf{get}();\ \_ \leftarrow r\#\mathsf{put}(x+1);\ r\#\mathsf{get}())$$
$$\rightsquigarrow \qquad \text{(S-RunscopeNew)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{runinst}^l_{s,\mathsf{State}}\{h\}($$
$$\quad x \leftarrow \mathsf{inst}(l)\#\mathsf{get}();\ \_ \leftarrow \mathsf{inst}(l)\#\mathsf{put}(x+1);\ \mathsf{inst}(l)\#\mathsf{get}());\ f\ 0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-RuninstOp)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{return}\ (\lambda st.f \leftarrow (\lambda x.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}($$
$$\quad \_ \leftarrow \mathsf{inst}(l)\#\mathsf{put}(x+1);\ \mathsf{inst}(l)\#\mathsf{get}()))\ st;\ f\ st);\ f0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-SeqReturn)}$$

$$\mathsf{runscope}^s((\lambda st.f \leftarrow (\lambda x.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}($$
$$\quad \_ \leftarrow \mathsf{inst}(l)\#\mathsf{put}(x+1);\ \mathsf{inst}(l)\#\mathsf{get}()))\ st;\ f\ st)\ 0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-App)}$$

$$\mathsf{runscope}^s(f \leftarrow (\lambda x.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}($$
$$\quad \_ \leftarrow \mathsf{inst}(l)\#\mathsf{put}(x+1);\ \mathsf{inst}(l)\#\mathsf{get}()))\ 0;\ f\ 0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-App)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{runinst}^l_{s,\mathsf{State}}\{h\}($$
$$\quad \_ \leftarrow \mathsf{inst}(l)\#\mathsf{put}(0+1);\ \mathsf{inst}(l)\#\mathsf{get}());\ f\ 0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-RuninstOp)}$$

$$\mathsf{runscope}^s(f \leftarrow (\mathsf{return}\ (\lambda st.f \leftarrow (\lambda\_.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}($$
$$\quad \mathsf{inst}(l)\#\mathsf{get}()))\ ();\ f\ (0+1)));\ f\ 0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-SeqReturn)}$$

$$\mathsf{runscope}^s((\lambda st.f \leftarrow (\lambda\_.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{inst}(l)\#\mathsf{get}()))\ ();\ f\ (0+1))\ 0)$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-App)}$$

$$\mathsf{runscope}^s(f \leftarrow (\lambda\_.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{inst}(l)\#\mathsf{get}()))\ ();\ f\ (0+1))$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-App)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{inst}(l)\#\mathsf{get}());\ f\ (0+1))$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-RuninstOpPrepare)}$$

$$\mathsf{runscope}^s(f \leftarrow \mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(x \leftarrow \mathsf{inst}(l)\#\mathsf{get}();\ \mathsf{return}\ x);\ f\ (0+1))$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-Seq, S-RuninstOp)}$$

$$\mathsf{runscope}^s(f \leftarrow (\mathsf{return}\ (\lambda st.f \leftarrow (\lambda x.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{return}\ x))\ st;\ f\ st));\ f\ (0+1))$$
$$\rightsquigarrow \qquad \text{(S-RunscopeCong, S-SeqReturn)}$$

$$\mathsf{runscope}^s((\lambda st.f \leftarrow (\lambda x.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{return}\ x))\ st;\ f\ st)\ (0+1))$$

$\rightsquigarrow$ (S-RunscopeCong, S-App)

$\mathsf{runscope}^s(f \leftarrow (\lambda x.\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{return}\ x))\ (0+1);\ f\ (0+1))$

$\rightsquigarrow$ (S-RunscopeCong, S-Seq, S-App)

$\mathsf{runscope}^s(f \leftarrow \mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(\mathsf{return}\ (0+1));\ f\ (0+1))$

$\rightsquigarrow$ (S-RunscopeCong, S-Seq, S-Runinst)

$\mathsf{runscope}^s(f \leftarrow \mathsf{return}\ (\lambda st.\mathsf{return}\ (0+1));\ f\ (0+1))$

$\rightsquigarrow$ (S-RunscopeCong, S-SeqReturn)

$\mathsf{runscope}^s((\lambda st.\mathsf{return}\ (0+1))\ (0+1))$

$\rightsquigarrow$ (S-RunscopeCong, S-App)

$\mathsf{runscope}^s(\mathsf{return}\ (0+1))$

$\rightsquigarrow$ (S-RunscopeCong, S-RunscopeReturn)

$\mathsf{return}\ (0+1)$

One thing to notice from this derivation is that $\mathsf{runscope}^{s_{loc}}(...)$ remains wrapped around at the top until all operation calls on its scope are handled. Similarly $\mathsf{runinst}^l_{s,\mathsf{State}}\{h\}(...)$ always remains wrapped around the part of the program that has operation calls on l.

5.7 The problem with type safety

Because our initial goal was to find a safe type system for dynamic instances, it is important to establish that our system is actually type safe. We will now give a precise main type soundness theorem. We will also give lemmas that are necessary in order to prove this theorem.

Theorem 4 (Type soundness)**.**

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

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 6 (Progress)**.**

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

Lemma 7 (Preservation)**.**

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

Just like the algebraic effects systems of Chapter 4 we can prove the type soundness theorem using progress and preservation lemmas. The progress lemma states that given a well-typed computation with no effects then either the computation is a value or we can take a step. The preservation lemma states that when taking a step on a well-typed computation, the result will still be well-typed.

Again we need to generalize the progress lemma in order to be able to proof it. We slightly modify the notion of effectful of Section 4.2 for our calculus, changing the operation calls to now have an instance location.

$$\mathsf{effectful}(c) \text{ if } (\exists l\ op\ \nu.c = \mathsf{inst}(l)\#op(\nu)) \text{ or } (\exists x\ l\ op\ \nu\ c'.c = x \leftarrow \mathsf{inst}(l)\#op(\nu);\ c')$$

We can now state the generalized version of the progress lemma, for the case where c does have effects.

Lemma 8 (Progress effectful)**.**

$$\text{if } \cdot; \Sigma; \cdot \vdash c : \tau \mathbin{!} r \text{ then } \mathsf{value}(c) \text{ or } \mathsf{effectful}(c) \text{ or } (\exists c'\ \Sigma'.\ c \mid \Sigma \rightsquigarrow c' \mid \Sigma')$$

Finally we give two lemmas which will be useful when proving the type soundness theorem. The first allows the dynamic environment Σ to be weakened. Adding fresh scope and instance locations to Σ should not disallow a previously well-typed computation from being well-typed.

Lemma 9 (Σ weakening)**.**

$$\text{if } (\Delta; \Sigma; \Gamma \vdash c : \underline{\tau}) \text{ and } \vdash \Sigma' \text{ and } \Sigma \subseteq \Sigma' \text{ then } \Delta; \Sigma'; \Gamma \vdash c : \underline{\tau}$$

The second lemma states that the operational semantics will always extend Σ monotonically. No scope or instance location occuring in Σ will be removed.

Lemma 10 (Semantics does valid monotonic extension)**.**

$$\text{if } c \mid \Sigma \rightsquigarrow c' \mid \Sigma' \text{ then } \Sigma \subseteq \Sigma'$$

Unfortunately due to time constraints we have not been able to formalize and prove the type soundness theorem in Coq at the time of writing.

Chapter 6

Related work

Algebraic effects and handlers is a wide field with many different areas and features to consider, such as concurrency and asynchrony (Dolan et al. 2017; Leijen 2017a), effect subtyping (Saleh et al. 2018), shallow handlers (Hillerström and Lindley 2018), event correlation (Bracevac et al. 2018), and effects in dependent types (Ahman 2018). We will focus only on languages which have support for dynamic effects, namely Eff (Bauer and Pretnar 2015), an OCaml embedding of Eff (Kiselyov and Sivaramakrishnan 2016), and Koka (Leijen 2017b; Leijen 2016).

Eff The Eff programming language (Bauer and Pretnar 2015) was the first language with support for algebraic effects and handlers. It featured fully dynamic instances (Section 2.3) allowing for dynamic effects. Eff did not feature an effect typing system initially. Having no effect typing system means there are fewer restrictions on what one can do, but there are also fewer static guarantees. For example it is not statically guaranteed that all operation call will be handled, which could result in runtime errors (as we have shown in Section 2.3). An effect typing system was proposed (Bauer and Pretnar 2014) but it did not feature dynamic instances, only static instances (Section 2.2 and Section 4.3). The current version of Eff does not have dynamic instances since they were considered too difficult in the theory. Dynamic instances in Eff can be introduced without an associated handler, different than Miro. In most language mutable references are globally scoped. Eff supports these kinds of globally scoped dynamic effects using *resources*, taking an example from "Programming with algebraic effects and handlers" (Bauer and Pretnar 2015):

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

Here the `new E @ x with h` syntax creates a new instance of effect `E` with an associated global handler `h`. The `x` is a piece of state that the handler is allowed to manipulate. In the example the `ref` function creates a new instance of the `ref` effect with initial value `x`. Resources always have a handler associated with them. These special handlers are globally scoped (they are wrapped around the whole program) and are more restricted than regular handlers and have different semantics. Because the handler is globally scoped any operation call on an instance with a resource will always be handled. Such handlers are also sometimes called *default handlers*. In the future work we discuss the possibility of adding default handlers to Miro in Section 7.2.

OCaml embedding A variant of Eff was implemented as an embedding in OCaml (Kiselyov and Sivaramakrishnan 2016). This embedding features algebraic effects and handlers and dynamic instances, again without an effect typing system. The implementation relies on multi-prompt delimited control in OCaml. The embedding does not implement the resources or default handlers from Eff. Instead the observation is made that dynamic effects can be seen as just another effect, which are called *higher-order effects*. A effect called `New` is defined with a `new` operation. We give the definition of `New` here but refer to the paper for a full explanation.

```
type eff handler t = {h: forall w. eff result prompt -> (unit -> w) -> w}
type dyn instance =
  New : eff handler t * (eff result prompt -> dyn instance result)
    -> dyn instance
let new instance p arg = shift0 p (fun k -> Eff (New (arg,k)))
let new handler p thunk =
  handle it p thunk
    (fun v -> v)
    (fun loop -> function New ({h=h},k) ->
      let new instance p = new prompt () in
        h new instance p (fun () -> loop @@ k new instance p))
let pnew = new prompt ()
let newref s0 = new instance pnew {h = handle ref s0}
```

The `new` operation takes as arguments an effect and a handler. Then a handler for `New` is defined which creates instances for each `new` operation and wraps the continuation in the handler that was given as argument. We use a very similar technique to implement creation of instances. Our `runscope` construct is similar to the handler of `New` in the Eff embedding. In essence we ban the normal creation of dynamic instances and force users to always use the equivalent of the `New` effect. This restriction allows use to give a type system which make sure no instance escape their handler. In the OCaml embedding no such restrictions are made and just like Eff one has no static guarantees that all operations will be handled.

Koka Koka is a programming language with effect inference using row polymorphism (Leijen 2017b). Later algebraic effects and handlers were also added (Leijen 2016). Some notion of mutable references can be implemented (Biernacki et al. 2018) by extending the language with a `lift` (also called `inject`) operation to inject effects, skipping a handler. References implemented using this technique are very limited though, being unable to escape even single functions. Leijen proposed an extension for Koka with dynamic effect handlers (Leijen 2018). This extension introduces *umbrella effects*, which are effects that can contain dynamic effects. For example an umbrella effect `heap` can be defined which contains dynamic effects of type `ref` (mutable references).

```
effect heap {
  fun new-ref(init : a) : ref<a>
}
effect dynamic ref<a> in heap {
  fun get() : a
  fun set(x : a) : ()
}
```

Values of type `ref` can then created using by defining a dynamic handler.

```
fun with-ref(init:a, action:ref<a> -> e b ) : e b {
  handle dynamic (action) (local=init) {
    get() -> resume(local,local)
    set(x) -> resume((),x)
} }
```

Similar to how we have give a handler when creating a dynamic instance. Using these dynamic handlers we can implement polymorphic heaps. In order to let references escape the function in which they are created, a `new-ref` operation is defined on the umbrella effect `heap`.

```
fun heap(action : () -> <heap|e> a ) : e a {
  handle(action) {
    new-ref(init) -> with-ref(init,resume)
} }
```

The `heap` handler then creates the dynamic `ref` handlers for each time `new-ref` is called, installing these handlers under the `heap` handler. This way the references can escape functions that define them as long as these functions are called under the `heap` handler. This is very similar to the `New` effect of the OCaml Eff embedding and to our `runscope` construct. Koka does not statically check that the dynamic effects do not escape their handler. Instead an exception effect is added to the effect annotation each time a dynamic handler is created. This means that one is always forced to handle the exceptions even if you know that none will be thrown. Similar to Miro safe references are proposed using higher-ranked types, like in the ST monad in Haskell, in order to ensure that no unhandled operations will happen. These definitions still do not remove the exception effect in the effect annotation though. In Miro instead we statically guarantee that instances do not escape their scope, we do not require and exception effect in the effect annotation.

Chapter 7

Conclusion and future work

We conclude with a brief discussion of what we presented in this thesis and discuss possible future work.

7.1 Conclusion

In Chapter 2 we have seen that algebraic effects and handlers are a composable approach to programming with side-effects. Using algebraic effects effects we can keep functions pure until we handle the effects within them. We have also seen that we are unable to express mutable references within algebraic effects and handlers. Dynamic instances, as introduced by Bauer and Pretnar in Eff (Bauer and Pretnar 2015), allow the programmer to dynamically generate instances of effects. Using dynamic instances one can implement dynamic effects such as mutable references and the dynamic opening of channels. Unfortunately the type system of Eff under-approximates the uses of effects, which can lead to runtime errors.

In Chapter 3 we presented a new language Miro with algebraic effects and handlers, and dynamic instances. Using a notion of effect scopes we are able to safely use dynamically created instances, ensuring that all operation calls are handled. We have shown how we can implement mutable references and vectors in Miro.

In Chapter 4 we presented formal accounts of algebraic effects and handler, with and without static instances, giving a type system and operational semantics. We have formalized these systems and proven type safety in Coq.

Finally, in Chapter 5 we have presented a type system and operational semantics for the core language of Miro. We ended the chapter with a discussion on the difficulties of proving type safety for Miro and gave possible approaches that might allow us to make progress on these proofs.

In this thesis we have shown that we can safely combine algebraic effects with a restricted form of dynamic instances by giving an explicit scope for the use of an instance. Using the notion of an effect scope and effect scope variables we can ensure that no operation call will be left unhandled and we can avoid runtime errors. We have also discussed the difficulties in proving type safety for such a system. By enabling the definition of dynamic effects, such as mutable references, algebraic effects and handlers can be useful in more situations.

7.2 Future work

Mechanization We currently have formalized the system with static instances of Section 2.1 and Section 2.2 and have proven type safety in Coq[1]. This formalization is briefly discussed in Section 4.4. Due to time constraints we were unable to also provide a formalization for Miro. It would be beneficial to also formalize the syntax, semantics and typing rules Miro and to prove type safety, in order to gain more certainty the system is safe.

We have implemented a prototype of Miro in Haskell[2]. We implemented the typing rules in a bidirectional (Pierce and Turner 2000) style. We also implemented the small-step operation semantics. Using the prototype we can typecheck and run Miro programs and verify our ideas.

There are many kinds of features possible which increase the expressiveness and guarantees of Miro. We will now discuss possible extensions to Miro.

Parametric polymorphism over any type. In order to keep the system simple and to only focus on the novel elements, Miro only supports parametric polymorphism over effect scopes. It would be very useful in practice to allow quantification over any type. We do not think that adding this will interfere with the other elements of the system.

Polymorphic effects. Having added polymorphism over any type it makes sense to also allow for polymorphic effects. In our examples we have defined a `State` effect with `Int` values. In order to avoid having to define a seperate effect for each type we would like to keep in our state, we could allow for effects to have type parameters. For example, we could define a polymorphic `State` effect like:

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

Using this effect we can have fully polymorphic mutable references. For example, the type of a reference holding an integer value would be `Inst s (State Int)`. Given that each reference carries the type of the value in the reference, `get` and `put` can be type safe.

Improved effect annotations. Currently the effect annotation of a computation type is a set of effect scopes. We could make these annotations more precise by also noting which effects are used on each scope. For example, from `Int!{s1, s2}` to `Int!{{State, Flip}@s1, {Rng}@s2}`. We could also allow users to restrict which effects occur on which effect scope in this way, giving more static guarantees. We do not see any difficulty in extending the annotations in this way.

Combine with regular algebraic effects and handlers. In Miro handlers are given when creating instances. These handlers are necessary in order to make sure that every instance has a handler, which completely handles the effects of that specific instance. In regular algebraic effects (Section 2.1) operation calls can be called anywhere and can also be handled anywhere higher up.
These are two different ways of using algebraic effects. We could combine Miro with the regular algebraic effects and handlers. For example:

[1] https://github.com/atennapel/dynamicinstances
[2] https://github.com/atennapel/dynamicinstances/tree/master/hs

```
combination : forall s. (Inst s State) -> ()!{s, State}
combination [s] r =
  x <- #get();
  r#put(x)
```

The function `combination` takes as argument a **`State`** instance on some effect scope `s`. We then call the `get` operation, but not on an instance and call this value `x`. We then store `x` in the reference argument `r`, by calling the `r#put` . From the effect annotation on the type of `combination` we can see we are both using the effect scope `s` and **`State`** without a scope. While `r` already has a handler associated with it, because one has to be given when creating it, the `#get` operation does not. We still have to give a handler for `#get` higher up, like one would do with regular algebraic effects and handlers. Effect interfaces can be used for both systems as we do not change these from the regular algebraic effects system in Miro. It is not clear how difficult it is to combine these two systems as they can interact, regular operations can be called within an effect scope.

Global scope Usually mutable references are globally scoped, meaning they are valid for the entire program. In Miro we have to explicitly scope instances using `runscope`. In order to fully emulate global mutable references we could add a special scope location `global`. Instances created on the `global` scope are globally scoped and can be used anywhere. We can modify the `ref` function from Section 3.1 to create globally scoped mutable references.

```
globalref : Int -> (Inst global State)!{global}
globalref v =
  new State@global {
    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
```

Notice that we no longer need `forall s.` in the type. By combining globally scoped instances and polymorphic effects we would be able to emulate fully polymorphic mutable references, as seen in the ML programming language. We would need a special runglobalscope construct in the semantics which always surrounds the entire program and handles any global instances. Care would have to be taken to ensure global instances cannot escape, we have to still make sure their operations are always handled. Questions remain on what limitations we need on the global handlers. Should we be allowed to call other effects in the handlers? Should we be allowed to invoke the continuation zero or multiple times? This idea is similar to the concept of resources in the Eff programming language, as discussed in the related work in Chapter 6. Handlers on the global scope are similar to default handlers.

Bibliography

Ahman, Danel (2018). "Handling fibred algebraic effects". In: *PACMPL* 2.POPL, 7:1–7:29. DOI: 10.1145/3158095. URL: https://doi.org/10.1145/3158095.

Bauer, Andrej and Matija Pretnar (2014). "An Effect System for Algebraic Effects and Handlers". In: *Logical Methods in Computer Science* 10.4. DOI: 10.2168/LMCS-10(4:9)2014. URL: https://doi.org/10.2168/LMCS-10(4:9)2014.

— (2015). "Programming with algebraic effects and handlers". In: *J. Log. Algebr. Meth. Program.* 84.1, pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001. URL: https://doi.org/10.1016/j.jlamp.2014.02.001.

Biernacki, Dariusz et al. (2018). "Handle with care: relational interpretation of algebraic effects and handlers". In: *PACMPL* 2.POPL, 8:1–8:30. DOI: 10.1145/3158096. URL: https://doi.org/10.1145/3158096.

Bracevac, Oliver et al. (2018). "Versatile event correlation with algebraic effects". In: *PACMPL* 2.ICFP, 67:1–67:31. DOI: 10.1145/3236762. URL: https://doi.org/10.1145/3236762.

Claessen, Koen and Peter Ljunglöf (2000). "Typed Logical Variables in Haskell". In: *Electr. Notes Theor. Comput. Sci.* 41.1, p. 37. DOI: 10.1016/S1571-0661(05)80544-4. URL: https://doi.org/10.1016/S1571-0661(05)80544-4.

Dolan, Stephen et al. (2017). "Concurrent System Programming with Effect Handlers". In: *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. Ed. by Meng Wang and Scott Owens. Vol. 10788. Lecture Notes in Computer Science. Springer, pp. 98–117. ISBN: 978-3-319-89718-9. DOI: 10.1007/978-3-319-89719-6_6. URL: https://doi.org/10.1007/978-3-319-89719-6%5C_6.

Flanagan, Cormac et al. (1993). "The Essence of Compiling with Continuations". In: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. Ed. by Robert Cartwright. ACM, pp. 237–247. ISBN: 0-89791-598-4. DOI: 10.1145/155090.155113. URL: https://doi.org/10.1145/155090.155113.

Hillerström, Daniel and Sam Lindley (2016). "Liberating effects with rows and handlers". In: *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*. Ed. by James Chapman and Wouter Swierstra. ACM, pp. 15–27. ISBN: 978-1-4503-4435-7. DOI: 10.1145/2976022.2976033. URL: https://doi.org/10.1145/2976022.2976033.

— (2018). "Shallow Effect Handlers". In: *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*. Ed. by Sukyoung Ryu. Vol. 11275. Lecture Notes in Computer Science. Springer, pp. 415–435. ISBN: 978-3-030-02767-4. DOI: 10.1007/978-3-030-02768-1_22. URL: https://doi.org/10.1007/978-3-030-02768-1%5C_22.

Jones, Simon Peyton (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Kiselyov, Oleg and K. C. Sivaramakrishnan (2016). "Eff Directly in OCaml". In: *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016*. Ed. by Kenichi Asai and Mark R. Shinwell. Vol. 285. EPTCS, pp. 23–58. DOI: 10.4204/EPTCS.285.2. URL: https://doi.org/10.4204/EPTCS.285.2.

Launchbury, John and Simon L. Peyton Jones (1994). "Lazy Functional State Threads". In: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. Ed. by Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa. ACM, pp. 24–35. ISBN: 0-89791-662-X. DOI: 10.1145/178243.178246. URL: https://doi.org/10.1145/178243.178246.

Leijen, Daan (2016). *Algebraic Effects for Functional Programming*. Tech. rep. Technical Report. 15 pages. https://www. microsoft. com/en-us/research …

— (2017a). "Structured asynchrony with algebraic effects". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*. Ed. by Sam Lindley and Brent A. Yorgey. ACM, pp. 16–29. ISBN: 978-1-4503-5183-6. DOI: 10.1145/3122975.3122977. URL: https://doi.org/10.1145/3122975.3122977.

— (2017b). "Type directed compilation of row-typed algebraic effects". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, pp. 486–499. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837. URL: http://dl.acm.org/citation.cfm?id=3009872.

— (2018). "First class dynamic effect handlers: or, polymorphic heaps with dynamic effect handlers". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018*. Ed. by Richard A. Eisenberg and Niki Vazou. ACM, pp. 51–64. DOI: 10.1145/3240719.3241789. URL: https://doi.org/10.1145/3240719.3241789.

Levy, Paul Blain, John Power, and Hayo Thielecke (2003). "Modelling environments in call-by-value programming languages". In: *Inf. Comput.* 185.2, pp. 182–210. DOI: 10.1016/S0890-5401(03)00088-9. URL: https://doi.org/10.1016/S0890-5401(03)00088-9.

Milner, Robin, Mads Tofte, and Robert Harper (1990). *Definition of standard ML*. MIT Press. ISBN: 978-0-262-63132-7.

Pierce, Benjamin C. and David N. Turner (2000). "Local type inference". In: *ACM Trans. Program. Lang. Syst.* 22.1, pp. 1–44. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100.

Plotkin, Gordon D. and Matija Pretnar (2013). "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4. DOI: 10.2168/LMCS-9(4:23)2013. URL: https://doi.org/10.2168/LMCS-9(4:23)2013.

Saleh, Amr Hany et al. (2018). "Explicit Effect Subtyping". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, pp. 327–354. ISBN: 978-3-319-89883-4. DOI: 10.1007/978-3-319-89884-1_12. URL: https://doi.org/10.1007/978-3-319-89884-1%5C_12.