

RADBOUD UNIVERSITY NIJMEGEN



FACULTY OF SCIENCE

A Substructural Type and Effect System

THESIS MSc SOFTWARE SCIENCE

Author:
Orpheas VAN ROOIJ

Supervisor:
dr. Robbert KREBBERS

Second reader:
prof. dr. Herman GEUVERS

March 2024

Abstract

Effect handlers are a powerful construct that can express complex programming abstractions. They are a generalisation of exception handlers that allow resumption from where the exception was raised. Crucially, allowing this resumption to happen at most once or in an unrestricted way has far-reaching consequences for a compiler that implements effect handlers. In addition to performance considerations, it breaks key rules of reasoning which as a consequence renders many optimisations no longer sound. This research addresses this problem by giving a type system that can track the number of times a resumption happens even in the presence of nested resumptions and references that store them. As a result, a compiler can efficiently implement effect handlers by knowing how continuations will be used and which optimisations are sound for each expression. The type system is proven sound by modelling types and judgements semantically via a unary logical relation using the separation logic of Iris. By virtue of using Iris, we support recursive effects and non-structurally treated references which are needed to type non-trivial programs such as a library for cooperative concurrency.

Contents

Acknowledgments	3
1 Introduction	4
2 Effect Handlers by Example	8
2.1 Stateful Computations	8
2.2 Generators from Iterators	11
2.3 Cooperative Concurrency	12
3 One-Shot Language	18
3.1 Syntax	18
3.2 Operational Semantics	22
3.3 Type Judgements and Type Safety	26
3.3.1 Type Safety	26
3.3.2 Copyable Types	27
3.3.3 Relation on Signatures, Types and Environments	27
3.4 Typing Examples	30
3.5 Typing Rules	37
4 Complete Language	42
4.1 Overview	42
4.2 Syntax and Semantics	44
4.3 Relation on Signatures and Rows	49
4.4 Typing Rules	51
5 Semantic Typing	55
5.1 Overview	55
5.2 Iris Logic	57
5.3 Program Logic	58
5.3.1 Specifications	59
5.3.2 Proof Rules	60
5.3.3 Protocols	61
5.3.4 Adequacy	63
5.4 Semantic Definitions	63
5.4.1 Semantic Types, Signatures and Rows	63
5.4.2 Semantic Typing Judgement	66
5.4.3 Semantic Relations	68
5.4.4 Semantic Constraints	69
6 Coq Formalisation	70
7 Related Work	73
7.1 Algebraic Effects and Handlers	73
7.2 Semantic Typing	73
7.3 Substructural Type and Effect Systems	74
7.4 Effect rows and polymorphism	76
7.5 Logical Relations and Effect Handlers	78
8 Conclusion and Future Work	79

Acknowledgments

I would like to firstly thank my supervisor Dr. Robbert Krebbers. Your passion in program logics, type systems and Programming Language research in general has not only introduced me to these vast and exciting fields but also inspired me to further pursue post-doctoral studies along this direction. I have learned a lot from you in this past year, and I am particularly thankful for putting the time to discuss this project on a weekly basis. I am grateful for all the helpful comments, feedback and insights you have provided and I think your attention to detail and preciseness has made me a better researcher.

To all my colleagues in the so-called MSc thesis factory, we have had our fair share of fun and distractions, but I like to think that this thesis could not have been what it is without you. You have provided me with the much needed support to finish this thesis but most importantly you helped me enjoy this journey. Through this common quest of writing our theses, I am happy that we have become more than just colleagues, we have become close friends.

I would also like to thank my family, specifically my mother Charulla and brother Lefkios for being patient with me throughout the years. Without your courage and support I would not have been able to pursue my passion as an eternal student.

Lastly, I would also like to thank the second reader Prof. Dr. Herman Geuvers as well as all the other wonderful professors at the Radboud university that have ignited my curiosity and interest in the foundations of Computer Science and PL research.

1 Introduction

Effect handlers are receiving increasing attention in the research community due to their power in expressing complex abstractions as derived constructs such as state, cooperative concurrency, and backtracking. Mainstream languages such as *OCaml 5* have already retrofitted them [Siv+21] while research oriented languages such as Eff [BP15], Effekt [BSO20], Koka [Lei14] and Frank [Con+20] take a more ubiquitous approach by centering their design around effect types and handlers.

A good way to gain an intuition into effect handlers is by comparing them with exception handlers specified with the try-in-unless syntax of Benton and Kennedy [BK01]. A try-in-unless handler has the form:

$$\text{try } x' \leftarrow N \text{ in } M \text{ unless } (\lambda x. E)$$

with N being a possibly erroneous expression, M the final expression to be executed upon non-exceptional behaviour, and $\lambda x. E$ being the exception branch with x the argument of the exception. Evaluation begins with expression N and if no exception occurs the result is bound to variable x' and execution continues with expression M . Exceptions occurring in M are not handled by the exception branch E . If an exception occurs in N , the exception value is bound to x and the exception branch E is taken.

Similarly, an effect handler has the form:

$$\text{try } N \text{ with } (\lambda x k. E) \mid (\lambda x'. M)$$

with N being an effectful expression, $\lambda x k. E$ the effect branch of the handler, and $\lambda x'. M$ being the return branch. The effect branch takes an additional argument k , a *continuation*, which can be seen as a special function that continues the execution of the expression N from the point where the effect (formerly called exception) was performed. Moreover, the argument applied to the continuation is a value that will take the place of the `do` expression that performs an effect thus providing a way for the handler to communicate with the effectful program. Just like the try-in-unless construct, evaluation begins with expression N and if no effect is performed the result is bound to x' and expression M continues. If an effect request is produced, the effect branch E is executed. The expression E has the ability to call k in its body thus resume the execution of N .

The following example defines the effectful function *decide* that returns the result of performing an effect. It is used in the effectful expression *main* to update the value of the locally declared reference x . The result of the expression is the value stored in the reference.

```
let decide = λ (). do () in
let main = λ (). let x = ref true in x := decide (); !x in
let handleDecide = λ e. try e () with
    λ () k. k false || k true
    | λ x. x
in handleDecide main
```

To interpret this effect, a handler *handleDecide* is defined that executes the associated continuation k with both boolean values and performs the logical *or* operation on the results. The (`||`) operation is short-circuited and so the handler performs a simple backtracking search. We note this language has only one kind of an effect, but multiple effects can be encoded (see section 4).

To understand how the expression is evaluated we need to identify the part of the expression that continuation k represents. The *main* expression begins evaluation by

allocating the reference x with the value `true`. The effect is then performed and the effect branch takes control. The continuation k now represents the evaluation context that `do ()` resides, delimited up to and excluding the handler `handleDecide`. Thus k is set to be $(x := _ ; !x)$, with the hole $(_)$ taking the place of the `do ()` expression. The argument that k is applied to will be placed in this hole. Evaluation then proceeds by calling $(k \text{ false})$ which returns `false` and $(k \text{ true})$ which returns `true`. The result of the whole expression is the result of applying the logical *or* operation on these two boolean values and so we get `true`.

Continuations are characterised by the number of times they are called, with one-shot continuations allowing at most one call and multi-shot continuations allowing arbitrary number of calls [HDB90]. The continuation in this example is multi-shot since it is called exactly twice.

Problem Statement

Two main problems arise from multi-shot use of continuations and they relate to performance and program reasoning in languages with imperative features.

Performance A continuation is a way to resume computation from a past point in time and to do this it stores the state of the program at that point. Supporting continuations can be done at different levels, ranging from continuation passing style (CPS) translations to lower-level approaches that adopt an unconventional stack structure [XL21; FR20]. *OCaml 5* takes the latter approach by dividing the stack into segments whereby a new segment is created for every new handler encountered [Siv+21]. In this setting, a continuation represents a list of stack segments and crucially, a one-shot continuation can be resumed by directly restoring itself to the current stack without any copying. However, multi-shot continuations require a copy before they are resumed to ensure that they can be called again. If continuations are not annotated with whether they are one- or multi-shot, it is not clear how a compiler can ensure correctness. Treating all continuations as multi-shot (so always copy before resumption) is an easy way to ensure correct usage but performance will suffer. The problem is only exacerbated when one-shot continuations are captured by multi-shot ones as that would implicitly make them multi-shot.

Unsound Optimisations In addition, certain compiler optimisations are no longer sound in the presence of multi-shot continuations because the rules of reasoning that justify them no longer hold [VP21; TB19]. The key assumption that is broken is that every code block entered is exited at most once. Consider again the above example but this time take `main` to be the following expression:

$$\lambda (). \text{let } x = \text{ref true in } x := \text{decide } () \ \&\& \ !x; \ !x$$

Instead of simply updating the reference x with the result of `decide`, we firstly apply the left-to-right, short-circuited logical *and* operation to it with the contents of the reference. Observe that we can optimise this expression in the following way:

- (1) $\lambda (). \text{let } x = \text{ref true in } x := \text{decide } () \ \&\& \ \text{true}; \ !x$
- (2) $\lambda (). \text{let } x = \text{ref true in } x := \text{decide } (); \ !x$
- (3) $\lambda (). \text{decide } ()$

At (1) we propagate the initialisation value of the reference, at (2) we optimise `(&&)`, and at (3) we remove the non-escaping reference. In *OCaml 5* that implements continuations using the stack segmentation approach, applying this optimisation would cause the outcome of `handleDecide main` to change from `false` to `true`, making it unsound. For

this reason, *OCaml 5* forbids multi-shot continuations altogether by throwing a run-time error when continuations are used more than once. Optimising programs with multi-shot continuations requires reasoning about the consequences of capturing resources such as references inside continuations [Vil22]. Solely local-level reasoning that this optimisation relies on can no longer be used.

Addressing the Problem

The issues with multi-shot continuations can be solved with a substructural type system that can detect the call usage of continuations and thus distinguish between effects that are handled in a one- or multi-shot way. With this usage information, a compiler can represent continuations in their most efficient form by allowing destructive resumptions for one-shot continuations. In addition, it can leverage the effect type information of each expression and thus avoid applying optimisations that are unsound in the presence of multi-shot effects.

In light of this observation, we propose a type and effect system named *Haffel*, that distinguishes between one- and multi-shot continuations by treating the former substructurally. The language of *Haffel* supports both shallow and deep handlers, type and effect sub-typing and polymorphism, recursive types and effects and mutable state, making it realistic enough for practical purposes. To prove type safety we take a semantic approach that interprets types, effects and type judgements in the higher-order separation logic of *Iris* [KTB17; Jun+18; Tim+22]. This semantic approach is well-suited for our language because it allows us to model all the aforementioned features in a relatively easy way. It additionally helped us in accurately (and mechanically) tracking how multi-shot effects and handlers interact with substructural resources, a non-trivial task due to the presence of non-local control flow.

Most research in the area of effects and handlers is primarily focused in devising alternative type and effect systems that are more suitable to every-day programming, presenting efficient implementations, or in defining denotational semantics for higher-order algebraic effects [VP23; XL21; BSO20; Con+20; Yan+22; BV23]. Research into combining effect systems with substructural type systems is only focused to functional languages with no mutable state [HLL20; Tan+24]. Incorporating mutable state in the language offers additional challenges and this work addresses this gap in the literature.

Hillerström, Lindley, and Longley [HLL20] devise an affine type system for a pure language with effects handlers that allows only one-shot continuations. Their main focus is comparing the asymptotic speedup gained in terms of run-time complexity when multi-shot effects instead of just one-shot effects are used. We instead focus on devising a type and effect system for a more expressive language with mutable state, multi-shot continuations, recursive types and sub-typing.

The work of Tang et al. [Tan+24] which combines linear types with multi-shot effects is more in line with our research. To ensure linearity, they devise a linear type and effect system that distinguish between one- or multi-shot effects. Their approach relies on two dual notions of linearity, value linearity which governs how variables from the environment are used and control-flow linearity that tracks how many times control can enter a code block. Their system supports effect polymorphism and enjoys principal types by incorporating qualified types that allow specification of linear dependencies between types and effects. To prove type safety they use the syntactic type approach of progress and preservation theorems [WF94].

Our work differs from Tang et al. [Tan+24] by considering a language with mutable state and equi-recursive effect types. By taking a semantic type soundness approach [Tim+22], we can utilise an abstract form of step-indexing as provided by the *Iris* framework to prove soundness of typing rules for recursive types and effects [AM01; Ahm06].

In addition, using *Iris*'s machinery for reasoning about concurrency we can support references that are non-substructural but store values of substructural types. These crucial ingredients are what allow us to type non-trivial programs such as an implementation of a cooperative concurrency library. The complete list of contributions is as follows:

- We introduce *Haffel*, a typed language with effect handlers and ML-style references (sections 3 and 4), that distinguishes between one- and multi-shot effects by tracking how their continuations are used.
- Our type system supports type and effect subsumption and polymorphism as well as recursive types and effects. It is expressive enough to accept non-trivial programs that create generators from iterators (section 2.2) and implement a library for cooperative concurrency (section 2.3). We additionally introduce a novel restricted effect row constructor $\text{i}\rho$ which in a certain sense is dual to the unrestricted type constructor from linear logic $!\tau$. To the best of our knowledge, we are the first to investigate and model a system that combines all the aforementioned features.
- We prove type safety (section 5) using a logical type soundness approach that interprets types and effects as semantic objects in the separation logic of *Iris*. We additionally make use of the novel *Iris*-based Separation Logic extension of Vilhena that allows reasoning about effects and handlers (section 5.3) [Vil22; VP20].
- Our work is formalised in *Coq* and is mechanically proven to be correct (section 6):

https://doi.org/10.5281/zenodo.10842591

2 Effect Handlers by Example

To show how effect handlers can be used to express different abstractions and to develop a better understanding of their workings, this section goes through three applications of effect handlers: handling Stateful Computations (section 2.1), producing Generators from Iterators (section 2.2), and implementing a library for Cooperative Concurrency (section 2.3). Our language of choice is *OCaml 5* due to its similarities with the formal language *Haffel* that we define in sections 3 and 4. Unlike *Haffel*, *OCaml 5* lacks an effect system so functions are not annotated with the possible effects they can perform and usage information of continuations is not tracked by the type system. As a result, unhandled effects and calling continuations more than once cause run-time errors. This is the main motivation of this thesis, to expand on the research for a substructural type and effect system for *OCaml 5*.

2.1 Stateful Computations

The expressivity of effect handlers can be demonstrated by the representation of state. Stateful computations as their name suggests depend on a notion of state that allows interaction by reading and writing to it. A common approach to represent stateful expressions is via two operations, `Get` that asks for the current value of the store and `Put` that replaces the contents of the store.

In a language with effect handlers we can represent `Get` and `Put` as effects that a handler must eventually provide an implementation for. In *OCaml 5* this is done by extending the GADT (Generalised Algebraic Data Type) type `a Effect.t` that represents all the possible effects that can happen in a program. The type parameter `a` of `Effect.t` represents the argument type of the continuation, so the type of values that the handler can respond to an effect call.

Assuming that our state is simply an integer value, we declare the `Get` effect as a constructor `Get` that takes the unit and returns an `int Effect.t` and the `Put` effect as a constructor `Put` that takes an integer and returns `unit Effect.t`.

```
type _ Effect.t +=  
  Get: unit -> int Effect.t  
  | Put: int -> unit Effect.t
```

Effects are called using the `perform` construct that takes an inhabitant of `Effect.t` and returns the response of the handler. From the caller side, performing an effect can be seen as a regular function call and so we provide the helper functions `get` and `put` that perform the respective effects:

```
let get () : int = perform (Get ())  
let put (s : int) : unit = perform (Put s)
```

Using our newly declared effects we can define an effectful version of the factorial function as follows. For numbers bigger than 1 we firstly update the store with the result of multiplying its current value with the argument `n`. It is then recursively called with the predecessor of the argument. The multiplication is performed before the recursive call and so the multiplications happen in descending order.

```
let rec fact (n : int) : unit =  
  if (1 < n) then  
    put (get() * n); fact (n - 1)
```

The extrinsic nature of effect calls that rely on handlers to give them an interpretation allows us to define two fundamentally different ways to handle the factorial function: an imperative implementation that relies on mutable state and a purely functional one by composing state transformers.

Handling using ML-style References The handler could be implemented by representing the state as a value stored behind a reference. To declare the handler, the `match_with` construct is used and it expects an effectful expression and its defining branches.

```

let fact_ref (n : int) : int =
  let store = ref 1 in
  match_with (fact n) {
    retc = (fun () -> !store);
    exnc = raise;
    effc = fun (type a) (eff: a Effect.t) ->
      match eff with
      | Get() ->
        Some(fun (k: (int, int) continuation) ->
          continue k !store)
      | Put(s) ->
        Some(fun (k: (unit, int) continuation) ->
          store := s; continue k ())
      | _ -> None
  }

```

The `retc` field of the record represents the return branch: the function that is applied to the resulting value that the expression `fact n` evaluates to. The `fact` function has result type `unit` and so the return branch takes the `unit` as argument. The result type of the handler is `int` since we return the value of the store.

The `exnc` field expresses how to deal with exceptions thrown during evaluation of `fact n`. In *OCaml 5*, exceptions are a separate entity in the language that are not defined using effect handlers due to their pervasiveness in applications and thus their need for low performance cost [Siv+21]. In our example, we handle exceptions by simply raising them again.

Importantly, the `effc` field represents the effect branch of the handler. The function returned expects an effect `eff` and is polymorphic in the type of effects, specified using the `type a` construct (Rank 1, Hindley-Milner style polymorphism [Hin+69; Mil78]). We pattern match on the effect to define how the `Get` and `Put` effects should be serviced. All other effects will not be caught but will instead be propagated upwards to be caught by an outer handler (last catch-all case). To understand what is happening we must first look at the type of continuations. A continuation in *OCaml 5* has type `(a, b) continuation`, with `a` being the type of values that it can be called with and `b` the resulting type. They can be called using the `continue` construct which takes the continuation and its argument and replaces the `perform` call in the continuation with the argument before resuming it. In both branches, the continuation represents the remaining computation of `fact n`. In the `Get` case, the argument type is `int` since we need to respond with the current value of the state whereas in the `Put` case the argument should be `unit` as dictated by its type `int -> unit Effect.t`¹.

¹Due to quirks in OCaml's type checker of GADTs, argument `k` in the function signature should in both cases be annotated as having a type `(a, _) continuation` to match with the argument type `a Effect.t`. The example is actually ill-typed because the type of continuations after unification is only known in the body of the function, but for the sake of clarity we do not consider this.

The return type of the continuation is inherited from the handler's return type which is `int` because it is resumed in the context of the handler. Subsequent effect calls that occur during evaluation of the continuation are again serviced by the same handler and as soon as it finishes execution, the return branch is taken and the value of the store is returned.

The Get effect is then defined as a function that takes the continuation as argument. To handle the effect, we simply resume evaluation with the current value of the store. Similarly, the Put effect is defined by firstly updating the store with the new value `s`, and then resuming the continuation with the unit value.

The overall execution of `fact n` happens by handling each effect call as either a read or a write to the store and finally returning its contents when evaluation has finished.

Handling using State Transformers Alternatively we could define a purely functional implementation as a composition of state transformers. Effectful expressions are handled by transforming them to functions that take the store as argument and return its new value as a result. The factorial function `fact_st` handles the `fact n` expression by transforming it to a function of type `int -> int`. Reverse function application (`|>`) is then used to apply it with the literal `1` which represents the initial value of the store.

The return branch of the handler takes the result of `fact n` (the unit value) and returns the identity function on stores. This represents the base case of the factorial function. If `n = 1`, the expression `fact 1` evaluates to the unit value and so the handler must map this unit value to the identity function in order to apply it with the current value of the store.

```
let fact_st n = 1 |> match_with (fact n) {
  retc = (fun () -> fun s -> s);
  exnc = raise;
  effc = fun (type a) (eff: a Effect.t) ->
    match eff with
    | Get() ->
      Some(fun (k : (int, int -> int) continuation) ->
        fun (v : int) -> continue k v v)
    | Put(s) ->
      Some(fun (k : (unit, int -> int) continuation) ->
        fun (v : int) -> continue k () s)
    | _ -> None
}
```

The return type of the continuations is that of the handler: `int -> int`. In both Get and Put the continuation is executed in the context of the handler which transforms it to a function that updates and returns a store. The Get effect branch is defined as a function that takes a store value `v`. The first argument to the continuation is the answer to the effect call and the second argument represents the current store value that is applied to the state transformer returned.

The Put effect branch is defined in a similar way. We return a function that defines how a store is updated. The difference here is that the current value of the store `v` will be discarded and instead the store value `s` will take its place. We firstly respond to the effect call by resuming the continuation with the unit value and then we pass it the store value `s` to signal that the store is updated.

2.2 Generators from Iterators

Iterators Iterators generalise sequence traversals to arbitrary containers that represent multiple elements. They can be defined as a fold-like function that takes an action to be applied on every element and returns the unit after all the applications occurred:

```
type 'elem iterator = ('elem -> unit) -> unit
```

A common example is the list iterator that applies the iterated function to every element in the list. The `list_iter` function returns an iterator from a list and is polymorphic to the type of the elements.

```
let list_iter (type a) (xs : a list) : a iterator =  
  fun (f : a -> unit) : unit ->  
    let rec go xs =  
      match xs with  
      | x :: xxs -> f x; go xxs  
      | [] -> ()  
    in go xs
```

Iterators are considered producer-centric because clients that use them have no control of the traversal. They produce the elements and decide when to apply the consumer function thus giving them complete control of the traversal.

Generators Generators on the other hand are consumer-centric: the client is in control of when an element is fetched and consumed. They can be represented as a side-effectful function that might return an element as result:

```
type 'elem generator = unit -> 'elem option
```

To create a generator from a list we can use a reference that will store the state of the traversal. At each call of the generator, the reference would be updated with the remaining elements in the list that still need to be consumed.

```
let list_gen (type a) (xs : a list) : a generator =  
  let rxs = ref xs in  
  fun () ->  
    match !rxs with  
    | x :: xxs -> rxs := xxs ; Some(x)  
    | [] -> None
```

Thus a generator can be called multiple times, with each time returning a different element from its associated container. Returning `None` signals that all elements have been produced. Note that a generator that has produced its elements can still be called and it is expected that it should keep returning `None`.

Iterators to Generators Iterators and generators are two mechanisms to traverse abstract containers, but their control differences make them applicable to different use cases. To make them more adaptable, it is useful to provide transformations between the two. Transforming a generator to an iterator is a straightforward task because they have control of the traversal. The other direction though is more challenging due to the need to suspend execution of the iterator at each step of the traversal and

resume it on demand. The non-local control flow of effect handlers are ideal candidates for this control inversion behaviour and so they can be used to transform iterators to generators. The following example shows how it can be achieved and it is largely inspired by Sivaramakrishnan [Siv15].

```

1  let generate (type elem) (iter : elem iterator) : elem generator =
2    let open struct
3      type _ Effect.t +=
4        Yield : elem -> unit Effect.t
5    end in
6    let yield (x : elem) : unit = perform (Yield x) in
7    let rec cont = ref (fun () ->
8      match_with (iter yield) {
9        retc = (fun () -> cont := (fun () -> None); None);
10       exnc = raise;
11       effc = fun (type a) (eff : a Effect.t) ->
12         match eff with
13           Yield(x) -> Some(fun (k: (unit, elem option) continuation)->
14             cont := continue k;
15             Some(x)
16           )
17         | _ -> None
18       }) in
19    fun () -> !cont ()

```

The `generate` function takes an iterator `iter` and returns its `generator` version. To do this it utilises the locally declared effect `Yield` which is passed as the argument to `iter` and thus an effect call is performed for every element produced (Line 8).

Generators are referentially opaque because they rely on a hidden state that tracks the progress of the traversal. For this reason, the reference `cont` is used which is captured by the generator returned and thus it is shared between successive calls (Line 19). The reference is higher-order since it holds the computation that generates the remaining elements that need to be produced. Initially, it stores a function that executes the iterator inside a handler that services `Yield` effects (Line 7). At each call to `yield`, the handler will update the reference with the continuation produced which holds the remaining execution of the `iter` function. The reference is defined recursively to allow the handler to access it during initial allocation (recursive references that store functions are well-formed).

When the iterator finishes its traversal, the unit value is returned and thus the return branch of the handler will be taken (Line 9). The `None` value is returned which signals that the generator has produced all its elements. Crucially, the reference `cont` is firstly overwritten with a new function that simply returns `None`. Generators can be called again after they have returned `None` and thus this is needed to avoid the last continuation stored in the reference from being called more than once. Continuations in *OCaml 5* are one-shot and calling a continuation multiple times causes a run-time exception.

2.3 Cooperative Concurrency

Asynchronous computations as realised using the `async/await` constructs allow concurrent execution of multiple tasks. It is a form of cooperative concurrency because asynchronous tasks decide when to give control to the task scheduler and thus tasks

need to cooperate to ensure fair sharing of resources. While many languages such as F# [SPL11], JavaScript [Par15], and Python [Sel15] have been extended to support asynchronous programming, effect handlers are powerful enough to express them as derived constructs and thus avoid the need for explicit language support. In this section we demonstrate how the `async/await` can be expressed in *OCaml 5*. The encoding described is taken from Dolan et al. [Dol+18] and Vilhena and Pottier [VP21].

The `async` construct expects an argument that represents the computation that should be performed asynchronously. The result is a *promise*, a structure that identifies the computation and tracks its evaluation progress. Using the `await` construct we can perform a blocking wait on a promise to signal that we depend on the result of the asynchronous computation to continue execution.

We define `async` and `await` as effects by extending the `Effect.t` data type. The `Yield` effect indicates to the handler that a currently running task can be suspended and control can be given to another task. We could have encoded `Yield` using the `Async/Await` effects but instead we define it as a separate effect in order to make explicit to the handler that a task switch should occur.

```

type _ Effect.t +=
  Async: (unit -> 'a) -> ('a promise) Effect.t
  | Await: 'a promise -> 'a Effect.t
  | Yield: unit Effect.t

let async (type a) (t : unit -> a) : a promise = perform (Async t)
let await (type a) (p : a promise) : a = perform (Await p)
let yield () : unit = perform Yield

```

To showcase the usage of our new effects, we define the effectful function `sonnet18` shown in fig. 1 that attempts to print the first line of Shakespeare’s famous Sonnet 18 using three asynchronous tasks, the `main` task and tasks 1 and 2. To help understand how it is executed, the expression is annotated with the continuations produced. In total there are six continuations, κ_1 to κ_6 , and every continuation captures the part of the expression that is surrounded by its dotted line.

The `runner` function defines a handler for the `Async/Await` effects by implementing a LIFO (Last-In-First-Out) scheduler that shares control between the unblocked tasks and is responsible for unblocking tasks waiting on a promise. The implementation of such a handler is described in the next section.

Initially, we pass the `sonnet18` expression to `runner` which will in turn start its execution until the first `async` call at Line 3. At this point the handler takes control and receives two arguments: the continuation κ_1 running from Line 3 to Line 18 that represents the rest of `main`, and the argument of `async`, task 1. We assume that the handler chooses to resume the continuation in an `async` call instead of starting the new task immediately and so task 1 is recorded to be ready for execution and κ_1 is resumed. Similarly, in the subsequent `async` call at Line 9, the handler is made aware of task 2 and the continuation κ_2 is resumed first. Evaluation of κ_2 blocks at Line 14, waiting for the result of task 1. The continuation κ_3 generated is set to be waiting for the promise `t1` and another task is run. Since the κ_3 continuation is blocked, the scheduler will follow the LIFO order and thus run task 1. This will in turn cause the first part of the sonnet to be printed: “Shall”. The `yield` call will suspend task 1 and its continuation κ_4 at Line 6 is set to be available. Continuing the last in, first out execution order, the scheduler decides to run task 2 and the string “I” is printed before it blocks waiting for promise `t1` to be resolved. Only task 1 can be resumed now, and so the scheduler runs κ_4 which causes the string “compare thee” to be printed. It also finishes execution and so the continuations κ_3 and κ_5 are set to be ready for resumption. The scheduler runs

```

1  let sonnet18 () = (* Main Task *)
2
3  let t1 = async (fun () -> (* Task 1 *)
4      printf "Shall ";
5      yield ();
6      printf "compare thee ")  $\kappa_4$ 
7  in
8
9  let t2 = async (fun () -> (* Task 2 *)
10     printf "I ";
11     await t1;
12     printf "summer's day")  $\kappa_5$ 
13  in
14     await t1;
15     printf "to a "  $\kappa_6$ 
16     await t2;  $\kappa_3$ 
17
18
19
20 let runner (type a) (main : unit -> a) : a = (* ... *)

```

Figure 1: `sonnet18` annotated with the boundaries of the continuations produced

κ_3 instead of κ_5 since it was the first one to be blocked on the promise. The string “to a ” is printed before it gets blocked on task 2. Task 2 is not yet finished and so the continuation κ_6 is blocked. The scheduler has no other option but run κ_5 and the last part of the sentence is printed. At this point task 2 is finished and so the continuation κ_6 can be resumed which finishes the execution of `sonnet18`.

Implementation Having seen a possible execution sequence of the asynchronous program, we continue with an implementation of a handler that implements a simple LIFO scheduler. In this encoding, a promise is represented as a reference to the `status` algebraic data type. The `status` of a promise can either be `Done x` which indicates that the computation has finished with result `x`, or it can be `Wait xs` which states that it has not finished execution and the continuations `xs` wait for its result.

```

type 'a status =
  Done of 'a
  | Wait of ('a, unit) continuation list

type 'a promise = 'a status ref

```

The function `runner` in fig. 2 takes an effectful expression `main` and returns its result. The `fulfil` function fulfils a promise by running its associated computation `e`. It is called on every promise exactly once at the point where promises are created (Line 22 and Line 39). Apart from installing a handler on the associated computation, `fulfil` also updates the promise when it is completed and informs computations that are blocked on it that they can continue. As a result, a promise is completely resolved by applying `fulfil` to it.

In this context, we use the term *task* to refer to both the fulfilment of a promise and the continuations produced by the effect calls. Tasks that are ready for execution are inserted to the queue `q`. This happens when a new promise is created and must thus be fulfilled (Line 22) or when a promise finishes its execution and all the blocked tasks that are waiting for its completed value are resumed (Line 13).

```

1  let rec runner (type a) (main : unit -> a) : a =
2
3  let q : (unit -> unit) Queue.t = Queue.create () in
4  let next () = if not (Queue.is_empty q) then Queue.pop q () in
5  let resume_task v k = Queue.add (fun () -> continue k v) q in
6
7  let rec fulfil : 'a. 'a promise -> (unit -> 'a) -> unit =
8  fun p e ->
9    match_with e () {
10     retc = (fun v ->
11       let Wait ws = !p in
12       p := Done v;
13       List.iter (resume_task v) (List.rev ws);
14       next ()
15     );
16     exnc = raise;
17     effc = fun (type a) (eff : a Effect.t) ->
18       match eff with
19       | Async(comp) -> Some(
20         fun (k : (a, _) continuation) ->
21           let new_prom = ref (Wait []) in
22           Queue.add (fun () -> fulfil new_prom comp) q;
23           continue k new_prom
24       )
25       | Await(prom) -> Some(
26         fun (k: (a, _) continuation) ->
27           match !prom with
28           | Done(x) -> continue k x
29           | Wait(ws) -> prom := Wait (k :: ws); next ()
30       )
31       | Yield -> Some(
32         fun (k: (a, _) continuation) ->
33           Queue.add (continue k) q;
34           next ()
35       )
36       | _ -> None
37     } in
38  let pmain = ref (Wait []) in
39  fulfil pmain main;
40  let Done x = !pmain in x

```

Figure 2: Handling the Async and Await effects

To achieve the goals of the `fulfil` function, a handler is installed on the expression `e` to handle its effects. If the `Async` effect is requested, a new promise structure is created (Line 21) which states that no tasks are waiting on it yet. There are two options here to pass control to. The continuation can be resumed with the newly created promise or the promise can be fulfilled. We choose to resume the continuation and postpone the fulfilment of the promise by putting it to the queue. For the `Await` effect, the action performed depends on the state of the promise. If the promise has been completed (Line 28), the continuation can simply be resumed with the result of the promise. In contrast, if the promise is not yet resolved (Line 29), we update the structure to record that task `k` is also waiting. Since the task requires the promise to be completed, we cannot yet continue it and so we instead run a task from the queue. The `Yield` effect is resolved by putting the current task `k` to the queue and running instead the oldest task available.

If the computation of the promise reaches a value, the return branch is taken with the value `v` representing the result (Line 10). The promise must be in `Wait` state since `fulfil` is called exactly once for every promise (Line 11). It is marked completed and the tasks blocked on this promise are resumed by providing them the result and putting them to the queue (Line 12-13). Control is passed to one of the tasks in the queue (Line 14). At this point, the promise has been completely resolved and any future calls to it via the `Await` call will receive its completed value.

We utilise the `fulfil` function to also evaluate the `main` function. The `main` function can be viewed as an asynchronous computation that is passed to an `Async` call with no tasks waiting for its promise. Thus we associate it with a promise that has no waiting tasks (Line 38). After the call to `fulfil` returns, the promise must be in `Done` state and so we return its result.

Analysis of `sonnet18` We now have all the required information to describe the execution of `sonnet18` in detail. The table below shows the contents of the queue `q` and of the promises at each time a task switch occurs. Task switches occur whenever the handler takes control which happens at every call to `await`, `async` and `yield`. The Active Task row shows the currently running task up until the switch occurs, the Output row shows what is printed at each step, and the last three rows show the resulting state of the data structures after the handler has finished updating them at each step.

The notation $\lambda^{\text{ff}} \text{p}$ stands for the complete application of `fulfil` to the promise `p` and its associated computation. The accompanying computation is left implicit.

Active Task	$\lambda^{\text{ff}} \text{pmain}$	$\kappa_1 \text{t1}$	$\kappa_2 \text{t2}$	$\lambda^{\text{ff}} \text{t1}$	$\lambda^{\text{ff}} \text{t2}$	$\kappa_4 ()$	$\kappa_3 ()$	$\kappa_5 ()$	$\kappa_6 ()$
Output				Shall	I	compare thee	to a	summer's day	
Queue <code>q</code>	$\lambda^{\text{ff}} \text{t1}$	$\lambda^{\text{ff}} \text{t1}$ $\lambda^{\text{ff}} \text{t2}$	$\lambda^{\text{ff}} \text{t1}$ $\lambda^{\text{ff}} \text{t2}$	$\lambda^{\text{ff}} \text{t2}$ κ_4	κ_4	κ_3 κ_5	κ_5	κ_6	
Promise <code>t1</code>	Wait []	Wait []	Wait [κ_3]	Wait [κ_3]	Wait [κ_3, κ_5]	Done ()			
Promise <code>t2</code>		Wait []	Wait []	Wait []	Wait []	Wait []	Wait [κ_6]	Done ()	

Evaluation of `runner sonnet18` begins by fulfilling the promise `pmain`. The handler is installed in the whole expression `sonnet18` and thus when the first `async` call happens, the continuation κ_1 produced is delimited at Line 18. The handler services the `Async` effect by adding the task `fulfil t1 (fun () -> (* Task 1 *) ...)` to the queue and continues κ_1 with the promise `t1`. Importantly, the `fulfil` function will install the handler on the computation of task 1 and thus any effect calls emitted by task 1 will be handled and continuations will be delimited at end of the task, so Line 6. This process is repeated for the next `async` call at Line 9.

The continuation κ_2 will then run and the **Await** effect will be emitted with the promise **t1** as argument. The handler reads that the promise **t1** is in **Wait []** state, and so updates it to record that κ_3 is waiting for it. The oldest task from the queue is run which is the fulfilment of promise **t1**. The string “**Shall** ” is printed followed by a call to the **Yield** effect. This will in turn suspend the execution of task 1 and the continuation κ_4 is inserted at the end of the queue.

The fulfilment of promise **t2** begins next and the string “**I** ” is printed before getting blocked on promise **t1**. The **t1** promise is updated to **Wait [κ_3, κ_5]** to record this fact. Resumption of continuation κ_4 follows which will print the string “**compare thee** ” and its promise will be completed which in turn adds κ_3 and κ_5 to the queue. The process continues until all tasks from the queue are executed.

3 One-Shot Language

This section introduces the typed language $Haffel_{os}$ an OCaml-like, call-by-value language that supports unnamed one-shot effects and mutable state. The language is based on the untyped language HH [VP21] and the main feature $Haffel_{os}$ introduces to HH is a substructural type and effect system that enforces one-shot use of continuations.

Section 3.1 introduces the syntax of $Haffel_{os}$, section 3.2 describes its semantics, and the substructural type and effect system is described in sections 3.3 to 3.5. In section 4 we show how to extend $Haffel_{os}$ to the complete $Haffel$ language which also allows multi-shot use of continuations and further supports named effects. In section 5 we will show how to prove type safety for this fully-fledged language using the logical type soundness approach [Tim+22]. For the complete proofs of the type system we refer the reader to the Coq formalisation discussed in section 6.

3.1 Syntax

Expressions

The syntax of $Haffel_{os}$ is shown in fig. 3. It is a λ -calculus that supports impredicative polymorphism, products, sums, mutable state, iso-recursive types and effect handlers. We note that throughout the sections that follow we assume the Barendregt variable convention [Bar84]. Thus expressions and types are assumed equal up to alpha-equivalence and substitutions are assumed to be capture-avoiding.

$\odot ::= + \mid - \mid == \mid <$	
$e ::= x \mid \mathbf{rec} f x. e \mid e_1 e_2 \mid \Lambda. e \mid e \langle \rangle$	(Polymorphic λ -calculus)
$\mid \mathbf{true} \mid \mathbf{false} \mid (i \in \mathbb{Z})$	(Literals)
$\mid \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3 \mid e_1 \odot e_2$	(Operations)
$\mid () \mid (e_1, e_2) \mid \mathbf{let} (x_1, x_2) = e_1 \mathbf{in} e_2$	(Products)
$\mid \mathbf{injL} e \mid \mathbf{injR} e \mid (\mathbf{match} e_1 \mathbf{with} e_2 \mid e_3)$	(Sums)
$\mid \mathbf{fold} e \mid \mathbf{unfold} e$	(Iso-recursive Types)
$\mid (\ell \in \mathit{Loc}) \mid \mathbf{ref} e \mid !e \mid e_1 := e_2$	(Mutable State)
$\mid \mathbf{do} e \mid \mathbf{eff} v N$	(Effects)
$\mid \mathbf{cont} \ell N$	(Continuations)
$\mid (\mathbf{try} e_1 \mathbf{with} e_2 \mid e_3) \mid (\mathbf{deep-try} e_1 \mathbf{with} e_2 \mid e_3)$	(Handlers)

Figure 3: Syntax of expressions in $Haffel_{os}$. Expressions highlighted in blue can only be dynamically created.

Expressions are variables (x), recursive abstractions ($\mathbf{rec} f x. e$), applications ($e_1 e_2$), type abstraction ($\Lambda. e$) and type instantiation ($e \langle \rangle$), literals, if-else boolean eliminators ($\mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3$), binary operations, the unit ($()$), pairs ((e_1, e_2)), pair eliminators ($\mathbf{let} (x_1, x_2) = e_1 \mathbf{in} e_2$), left and right sum injections, sum eliminators ($\mathbf{match} e_1 \mathbf{with} e_2 \mid e_3$), recursive type folding and unfolding, locations on the heap (ℓ), reference allocations ($\mathbf{ref} e$), reference reads ($!e$), reference writes ($e_1 := e_2$), effect performing ($\mathbf{do} e$), active effect expressions ($\mathbf{eff} v N$), one-shot continuations ($\mathbf{cont} \ell N$), shallow handlers ($\mathbf{try} e_1 \mathbf{with} e_2 \mid e_3$) and deep handlers. We assume that x and f range

over a countably infinite set of variables. Non-recursive abstractions, Let expressions, sequencing and Option type constructors are derived constructs encoded as:

$$\begin{aligned}
& \lambda x. e := \mathbf{rec} \ f \ x. \ e && \text{where } f \notin \mathit{FreeVar}(e) \\
\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 &:= (\lambda x. e_2) \ e_1 \\
e_1; e_2 &:= (\lambda x. e_2) \ e_1 && \text{where } x \notin \mathit{FreeVar}(e_2) \\
\mathbf{none} &:= \mathbf{injL} \ () \\
\mathbf{some} \ e &:= \mathbf{injR} \ e
\end{aligned}$$

Expressions highlighted in blue indicate run-time expressions that are only created during evaluation. A user can only use static expressions to define programs and this is enforced by the type system because dynamic expressions are not well-typed terms in the language. The neutral evaluation context N represents an evaluation context that does not span through a handler and is used to describe a continuation. Evaluation contexts are presented in section 3.2 as part of the operational semantics of the language.

Shallow versus Deep Handling: A shallow handler $\mathbf{try} \ e_1 \ \mathbf{with} \ \lambda x \ k. \ e_2 \mid \lambda x. \ e_3$ consists of an effect branch $\lambda x \ k. \ e_2$ and a return branch $\lambda x. \ e_3$. The return branch is executed when the handled expression e_1 finishes its execution and it is used to change the type of the evaluated expression e_1 . The effect branch is used when an effect call happens. It is a function that takes two arguments, the effect value x and a continuation k . Deep handlers $\mathbf{deep-try} \ e_1 \ \mathbf{with} \ \lambda x \ k. \ e_2 \mid \lambda x. \ e_3$ are defined in the same way as shallow handlers but they differ operationally by being persistent. A deep handler will intercept all effect calls from expression e_1 until it finishes execution whereas a shallow handler services only the first effect and disappears.

Types

For an expression e to be well-typed with effect signature σ and type τ , there must exist a derivable judgement of the form:

$$\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2$$

The two-environment judgement can be read as follows: for an initial variable environment Γ_1 that associates all free variables in e with a given type, expression e can perform effects that adhere to the signature σ and after evaluation the resulting value has type τ and the environment Γ_2 will hold. The initial variable environment Γ_1 differs from the final Γ_2 in that affine usage of resources is enforced thus variables of a substructural type in the initial environment that are used during evaluation of e do not appear in the final environment. The rules to derive such judgements are presented in section 3.3.

The syntax of types and effect signatures is shown in fig. 4. Let α and θ range over a countably infinite set of type variables.

Base Types *Haffel_{os}* supports four base types, `void`, `()`, `bool` and `int`. The `void` type has no inhabitants. The `()` type is a nullary product constructed using `()`. The `bool` type represents booleans constructed using `false` and `true` and eliminated using `if e_1 then e_2 else e_3` . The `int` type is for integers constructed using the syntactical integers i and are used in arithmetic operations \odot .

Function Types Function types exist in two kinds, the affine function type denoted using the $\overset{\sigma}{\rightarrow}$ arrow, and the unrestricted function type denoted using the regular arrow $\overset{\sigma}{\rightarrow}$. Constructors for the affine function type are abstractions $\lambda x. e$ that capture variables from the environment that are substructural such as one-shot continuations.

$\tau, \kappa, \iota ::= \alpha$	(Type Variables)
$\mathbf{void} \mid () \mid \mathbf{bool} \mid \mathbf{int}$	(Base Types)
$\mid \tau \xrightarrow{\sigma} \kappa \mid \tau \xrightarrow{\sigma} \kappa$	(Function Types)
$\mid \tau * \kappa \mid \tau + \kappa$	(Product and Sum Types)
$\mid \mathbf{ref} \tau \mid !\tau$	(Reference and Unrestricted Type)
$\mid \forall \alpha. \tau \mid \forall \theta. \tau$	(Universal Types)
$\mid \mu \alpha. \tau$	(Recursive Types)
$\sigma, \hat{\sigma}, \sigma' ::= \theta$	(Signature Variables)
$\mid \perp$	(Nil Signature)
$\mid \mu \theta. \forall \alpha. \tau \Rightarrow \kappa$	(Effectful Signature)

Figure 4: Syntax of types in $Haffel_{os}$.

The affineness property of these variables propagates to the whole function and so the function itself must be called at most once. Constructors for the unrestricted function type are (possibly) recursive functions $\mathbf{rec} f x. e$ that can be called multiple times. They are treated non-substructurally and thus can only capture non-substructural variables. Since recursive functions can be called again inside their body, by definition they do not adhere to the at most one call restriction and must be given the unrestricted function type. Eliminators for both function types are application expressions $e_1 e_2$.

Additionally, function types are annotated with an effect signature σ which indicates whether it may perform an effect. A non-effectful function² is annotated with the empty signature \perp . In such cases we avoid writing the signature and instead use the notations $\xrightarrow{\perp}$ and $\xrightarrow{\perp}$ for the function types $\xrightarrow{\perp}$ and $\xrightarrow{\perp}$.

Product and Sum Types Product and sum types have the usual pair (e_1, e_2) and injections $\mathbf{injL} e, \mathbf{injR} e$ as constructors. The pair eliminator $\mathbf{let} (x_1, x_2) = e_1 \mathbf{in} e_2$ is used instead of projection constructs because it can access both components simultaneously. To extract both components of a pair using projections, the pair would have to be used twice which is not possible if the pair is substructural. The sum eliminator $\mathbf{match} e_1 \mathbf{with} \lambda x. e_2 \mid \lambda x. e_3$ expects an abstraction in its two branches with the binder x representing the inner inhabitant of the sum type.

We note that the option type is defined using a sum with its constructors being **none** and **some** e :

$$\mathbf{Option} \tau := () + \tau$$

Reference Types Reference types $\mathbf{ref} \tau$ are constructed using the $\mathbf{ref} e$ expression which allocates a new reference and returns a location ℓ that identifies it. References can be read using the $!e$ expression and updated using $e_1 := e_2$ which updates the reference in e_1 with the value of e_2 . Loading from a reference $!e$ is only allowed if the reference points to a value of a copyable type to ensure that it will not be used more than once.

²Non-effectful function in that no user-defined effects (performed via the \mathbf{do} construct) can occur. General computational effects such as side-effects can still happen. $Haffel_{os}$ has imperative features such as ML-style references that are not tracked by the effect system.

Universal Types The universally quantified type $\forall\alpha. \tau$ is constructed from expressions $\Lambda. e$ and eliminated using $e \langle \rangle$. To ensure type soundness in the presence of mutable state, generalisation does happen to arbitrary expressions but only to expressions $\Lambda. e$. This delays the evaluation of e until it is instantiated using $e \langle \rangle$. The approach is similar to the value restriction which only allows generalisation of values [Wri95].

Similarly, universally quantified signatures $\forall\theta. \tau$ allow polymorphism of effect signatures and are constructed from expressions $\Lambda. e$ and eliminated by $e \langle \rangle$. Higher-order functions that take as argument arbitrary effectful functions can be given an effect polymorphic signature to indicate that its effects come from its arguments. A prime example is the application function which can be given the type:

$$\Lambda. \Lambda. \Lambda. \lambda f x. f x : \forall\theta. \forall\alpha. \forall\beta. (\alpha \xrightarrow{\theta} \beta) \rightarrow \alpha \xrightarrow{\theta} \beta$$

For any effect signature θ and types α, β , given a function with type $(\alpha \xrightarrow{\theta} \beta)$ and an inhabitant of α , we receive an inhabitant of β . The fact that function application takes an affine function as argument indicates that it will call its argument at most once. The sub-typing relation defined in section 3.3.3 will allow us to pass an unrestricted function as well. Function application itself is an unrestricted function as it does not capture variables from the environment. When it is partially applied it should return an affine function to ensure that its function argument is used at most once. This is evident from the result type $\alpha \xrightarrow{\theta} \beta$.

Unrestricted Type The unrestricted type $!\tau$ is similar to the one from linear logic and specifies that type τ is copyable. It is primarily used with polymorphic functions that work with arbitrary copyable types. For instance, the type given to the following function that duplicates its argument utilises unrestricted types to state that it is polymorphic only over copyable types:

$$\Lambda. \lambda x. (x, x) : \forall\alpha. !\alpha \rightarrow !\alpha * !\alpha$$

Using the sub-typing relation defined in section 3.3.3 we can convert a type τ to an unrestricted type $!\tau$ and vice-versa. We note that contrary to other presentations of linear type systems, we do not explicitly feature the Dereliction rule since it can be derived by the effect sub-typing rule and Promotion is *not* sound in our system [Abr93; Wad91]. We expand more on this in section 7.

Recursive Types $Haffel_{os}$ supports iso-recursive types which allows us to take fix-points of arbitrary type constructors [Pie02]. They are constructed from `fold` e expressions and eliminated using `unfold` e . As is the case with iso-recursive types, the folded $\mu\alpha. \tau$ and unfolded version $\tau[\alpha/\mu\alpha. \tau]$ of a recursive type are not equivalent but are isomorphic. This isomorphism is witnessed from the unfolding reduction rule shown in fig. 5.

The List type is defined using recursive types as:

$$\mathbf{list} \tau := \mu\alpha. () + \tau * \alpha$$

Effect Signatures An effect signature can either be the nil signature \perp or the effectful signature $\mu\theta. \forall\alpha. \tau \Rightarrow \kappa$. The nil signature indicates the absence of effects and thus expressions annotated with nil do not call any effects. The effectful signature gives a recursively polymorphic signature where types τ and κ can depend on type variable α and signature variable θ . If an effect signature is not recursive we omit writing the recursive binder and similarly we omit the universal binder for monomorphic signatures.

An expression that performs an effect of signature $\mu\theta. \forall\alpha. \tau \Rightarrow \kappa$ can view an effect call as a form of a function call, where τ is the type of the argument passed to the `do` construct and κ is the resulting type of `perform`. From a handler’s perspective, the effect signature states that an effect is a value of type τ and a continuation that takes an argument of type κ representing the response to the effect. In this sense, the effect signature constraints the communication between handler and effectful expression by requiring that it adheres to the type specification.

Effect signatures can be polymorphic and so an expression that performs an effect will instantiate α with an arbitrary type. This means that the argument of `do` will be of type $\tau [\iota/\alpha]$ for some type ι . For instance, we can specify the polymorphic identity signature that allows an expression to call the effect with an argument of any type:

$$\forall\alpha. \alpha \Rightarrow \alpha$$

A handler for this signature must work for all types and so can only call the continuation once with the effect value it received. Effect signatures can also be recursive which allows types τ and κ to reference the same signature they are part of. For instance, we can define the following signature which allows an effectful expression to pass a function that takes the unit and returns a value of some type:

$$\mu\theta. \forall\alpha. (\ () \xrightarrow{\theta} \alpha) \Rightarrow \alpha$$

Importantly, the function passed can be effectful over the same signature. A handler must install itself recursively in the application of the function if it wants to resolve the effects produced by the function. For instance, in the Cooperative Concurrency example from section 2.3, the handler represented using the `fulfil` function registered itself in the computation returned from the `Async` effect.

We note that effect signatures support type polymorphism over a single type variable only. Extending effect signatures to support quantification over a list of type variables introduces some complexity in the *Coq* formalisation and so it is left for future work.

3.2 Operational Semantics

Effects and their handlers add some complexity to the operational semantics of a language because part of the evaluation context must be captured during execution. Before delving into the small-step semantics of *Haffel_{os}*, a simple example is presented that illustrates how an effectful program is executed.

Reader Example

The following program written in *Haffel_{os}*, defines the effect `read` and its associated handler `withRead`. The notation $\lambda (). e$ is used to define an abstraction that takes the unit value as argument. The handler has two branches, the effect branch ($\lambda x k. k n$) and the return branch ($\lambda x. x$). It is parameterised with an argument n that is the constant value passed to the continuation when the `read` effect is performed.

```

let read = λ (). do () in
let withRead = λ n e. try e () with
    λ x k. k n      -- effect branch
  | λ x. x          -- return branch
in 1 + withRead 2 (λ (). read () + 3)

```

After expanding the let-bindings and applying the first three applications to arguments n , e and the unit value, evaluation continues with the expression $read () + 3$ which triggers a call to $do ()$. At this point, the perform expression is transformed to the active effect expression $\mathbf{eff} () \bullet$ with $()$ being the value passed to perform and \bullet a hole to be filled. The purpose of the active effect expression is to swallow its surrounding context up to the enclosing handler and thus it becomes $\mathbf{eff} () (\bullet + 3)$. The context stops at the **try-with** handler and so the addition of 1 does not appear.

At this point the expression has been partially evaluated to:

$$1 + (\mathbf{try} (\mathbf{eff} () (\bullet + 3)) \mathbf{with} \\ \lambda x k. k 2 \\ | \lambda x. x)$$

The next reduction step is to reify the active effect expression to a continuation. It becomes the value $\mathbf{cont} \ell (\bullet + 3)$ with ℓ being a fresh memory location pointing to false. The reference ℓ stores whether the continuation has been called. The effect branch of the handler is executed with the unit and the continuation substituted, $(k 2)[\mathbf{cont} \ell (\bullet + 3)/k][()/x]$, which correspondingly resumes the continuation with the constant 2. Calling the continuation simply involves substituting the argument 2 to the hole \bullet . To record that the continuation has been called the content of the memory location ℓ is also changed to true. The resulting expression becomes $1 + (2 + 3)$.

This example shows that effectful programs capture evaluation contexts up to their nearest enclosing handler. The dynamic nature of evaluation contexts suggests that continuation building must be part of the small-step semantics. Additionally, by associating with each continuation a boolean field that tracks its called status, we can operationally forbid calling a one-shot continuation multiple times³. By incorporating this check in the semantics of the language we get that type safety implies that one-shot continuations are called at most once.

Small-Step Semantics

The small-step semantics of $Haffel_{os}$, are inspired by Kammar, Lindley, and Oury [KLO13] semantics and from its simplified reformulation for unnamed effects by Vilhena and Pottier [VP21]. Figure 5 defines the values, stores, evaluation contexts and the head step reduction relation of $Haffel_{os}$. Values in the language are recursive abstractions $\mathbf{rec} f x. e$, type abstractions $\Lambda. e$, booleans and integer literals, the unit $()$, pairs (v, w) , left sum type injections $\mathbf{injL} v$, right sum type injections $\mathbf{injR} v$, folded values $\mathbf{fold} v$, locations on the heap ℓ and continuations $\mathbf{cont} \ell N$. Stores are finite maps from locations to values and are updated using the operation $\sigma[\ell \mapsto v]$ which returns an identical store to σ apart from location ℓ where it points to v . Evaluation contexts are either neutral, N , or are general evaluation contexts K that include neutral contexts and can span through handlers as well. They are defined according to the evaluation order of $Haffel_{os}$, which is right-to-left. The complete small-step relation is given using the following inference rule.

$$\frac{e / \sigma \rightsquigarrow^h e' / \sigma'}{K[e] / \sigma \rightsquigarrow K[e'] / \sigma'}$$

The rules for application, operators, boolean, pair and sum eliminators, recursive type unfolding and references are fairly standard and show their usual evaluation behaviour. We focus on the rules concerning the effects which show how they are performed and serviced by the handlers.

³This operational trick is credited to Vilhena and Pottier which they used in their *HH* language to show that their program logic ensures one-shot usage of continuations [VP21].

The first step in creating a continuation is by evaluating a `do v` expression. The run-time expression `eff v •` is introduced, that has a hole `•` signifying that no context has been captured so far. The step rule $N[\mathbf{eff} v N'] / \sigma \rightsquigarrow^h \mathbf{eff} v (N[N']) / \sigma$ subsequently causes it to swallow its surrounding context up to the enclosing handler. The operation $N[\mathbf{eff} v N']$ fills the hole in N with the expression `eff v N'` which results in a valid expression. The operation $N[N']$ on the other hand yields a valid neutral context as neutral contexts can be composed. Some concrete examples of this rule are:

$$\begin{aligned} \text{injL}(\mathbf{eff} v N) / \sigma &\rightsquigarrow^h \mathbf{eff} v (\text{injL} N) / \sigma \\ (\mathbf{eff} v_1 N) v_2 / \sigma &\rightsquigarrow^h \mathbf{eff} v_1 (N v_2) / \sigma \end{aligned}$$

When the expression `eff v N` reaches a handler, evaluation proceeds by executing the effect branch with the values v and the reified version of N .

To model a run-time error when a continuation is called more than once, a fresh ℓ location is created that is stored in the `cont ℓ N` structure⁴. Application of a continuation is only valid when ℓ points to a value `false` and upon correct application, this value is updated to `true` signalling that no further applications can occur. Thus calling a continuation twice will result in a stuck expression as no reduction rule applies in the case where ℓ points to `true`.

The difference between deep and shallow handlers is that deep ones reinstate themselves in the continuations. This means that all subsequent effects produced by the resumption of the continuation are serviced by the same handler. To do this applications to the continuation are performed in the context of the handler by wrapping the continuation with the abstraction:

$$\lambda x. \text{deep-try}(\text{cont } \ell N) x \text{ with } \lambda x k. h \mid \lambda x. r$$

⁴Locations ℓ serve two purposes, one for representing allocated memory cells on the heap and another for tracking the call usage of continuations. Thus we model both the heap and the run-time error using the notion of a store, but this is just a convenience to avoid introducing two separate objects. These two purposes should be seen as distinct, meaning locations in `cont ℓ N` should not be used in reference operations and vice versa.

Values	Store
$v, w \in Val ::= \text{rec } f x. e \mid \Lambda. e$ $\mid \text{true} \mid \text{false} \mid (i \in \mathbb{Z})$ $\mid () \mid (v, w) \mid \text{injL } v \mid \text{injR } v$ $\mid \text{fold } v \mid (\ell \in Loc) \mid \text{cont } \ell N$	$Store \triangleq Loc \rightarrow_{\text{fin}} Val$ $\sigma, \sigma' \in Store$
Evaluation Contexts	
$N ::= \bullet \mid e N \mid N v \mid N \langle \rangle$ $\mid \text{if } N \text{ then } e_1 \text{ else } e_2 \mid e \odot N \mid N \odot v$ $\mid (e, N) \mid (N, v) \mid \text{let } (x_1, x_2) = N \text{ in } e$ $\mid \text{injL } N \mid \text{injR } N \mid (\text{match } N \text{ with } e_1 \mid e_2)$ $\mid \text{fold } N \mid \text{unfold } N$ $\mid \text{ref } N \mid !N \mid e := N \mid N := v$ $\mid \text{do } N$ $K ::= N \mid (\text{try } K \text{ with } e_1 \mid e_2) \mid (\text{deep-try } K \text{ with } e_1 \mid e_2)$	
Head Reduction Relation	
<i>Standard Lambda Calculus</i>	
$(\text{rec } f x. e) v / \sigma \rightsquigarrow^h e[v/x][\text{rec } f x. e/f] / \sigma$ $(\Lambda. e) \langle \rangle / \sigma \rightsquigarrow^h e / \sigma$ $\text{if true then } e_1 \text{ else } e_2 / \sigma \rightsquigarrow^h e_1 / \sigma$ $\text{if false then } e_1 \text{ else } e_2 / \sigma \rightsquigarrow^h e_2 / \sigma$ $v_1 \odot v_2 / \sigma \rightsquigarrow^h v / \sigma \text{ if } \llbracket v_1 \rrbracket \llbracket \odot \rrbracket \llbracket v_2 \rrbracket = \llbracket v \rrbracket$ $\text{let } (x_1, x_2) = (v_1, v_2) \text{ in } e_3 / \sigma \rightsquigarrow^h e_3[v_1/x_1][v_2/x_2] / \sigma$ $\text{match (injL } v) \text{ with } e_1 \mid e_2 / \sigma \rightsquigarrow^h e_1 v / \sigma$ $\text{match (injR } v) \text{ with } e_1 \mid e_2 / \sigma \rightsquigarrow^h e_2 v / \sigma$ $\text{unfold (fold } v) / \sigma \rightsquigarrow^h v / \sigma$	
<i>References</i>	
$\text{ref } v / \sigma \rightsquigarrow^h \ell / \sigma[\ell \mapsto v] \text{ if } \ell \notin \text{Dom } \sigma$ $! \ell / \sigma \rightsquigarrow^h v / \sigma \text{ if } \sigma(\ell) = v$ $\ell := v / \sigma \rightsquigarrow^h w / \sigma[\ell \mapsto v] \text{ if } \sigma(\ell) = w$	
<i>Effect Handling</i>	
$\text{do } v / \sigma \rightsquigarrow^h \text{eff } v \bullet / \sigma$ $N[\text{eff } v N'] / \sigma \rightsquigarrow^h \text{eff } v (N[N']) / \sigma$ $(\text{cont } \ell N) v / \sigma[\ell \mapsto \text{false}] \rightsquigarrow^h N[v] / \sigma[\ell \mapsto \text{true}]$ $\text{try } v \text{ with } \lambda x k. h \mid \lambda x. r / \sigma \rightsquigarrow^h r[v/x] / \sigma$ $\text{deep-try } v \text{ with } \lambda x k. h \mid \lambda x. r / \sigma \rightsquigarrow^h r[v/x] / \sigma$ $\text{try (eff } v N) \text{ with } \lambda x k. h \mid \lambda x. r / \sigma \rightsquigarrow^h h[v/x][\text{cont } \ell N/k] / \sigma[\ell \mapsto \text{false}]$ $\text{if } \ell \notin \text{Dom } \sigma$ $\text{deep-try (eff } v N) \text{ with } \lambda x k. h \mid \lambda x. r / \sigma \rightsquigarrow^h h[v/x][k'/k] / \sigma[\ell \mapsto \text{false}]$ $\text{if } \ell \notin \text{Dom } \sigma$ where $k' := \lambda x. \text{deep-try (cont } \ell N) x \text{ with}$ $\lambda x k. h$ $\mid \lambda x. r$	

Figure 5: Values, Evaluation Contexts and the head reduction relation

3.3 Type Judgements and Type Safety

The type system of $H_{\text{aff}}\text{el}_{\text{os}}$ is substructural, supports sub-typing and tracks user effects. More specifically, it enforces affine usage of one-shot continuations, supports subsumption of environments, signatures and types and further tracks the effects produced by each expression at the type level. The effect system that tracks the signatures of effects is partly inspired by *TES* [VP23], a non-substructural type and effect system for a *HH*-like language that allows multi-shot use of continuations. To make the system substructural we restrict the contraction rule to hold only for variables with copyable types, i.e., variables that can be used more than once. This is achieved by utilising two-environment judgements $\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2$ that allow threading the *unused* part of the environment between sub-derivations according to the evaluation order. The weakening rule holds for all types because affine (instead of linear) usage of resources is enforced. We note that contraction and weakening can be performed on the final environments Γ_2 as well by utilising the subsumption relation between environments. This is elaborated in section 3.3.3.

$\frac{\text{CONTRACTION} \quad x : \kappa, x : \kappa, \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2 \quad \text{Copy } \kappa}{x : \kappa, \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}$	$\frac{\text{WEAKENING} \quad \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}{x : \kappa, \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}$
--	--

By restricting the contraction rule, the typing rules ensure that inhabitants of substructural types such as one-shot continuations are used at most once. This guarantee is essential to show type safety since calling a continuation more than once leads to an unsafe expression.

3.3.1 Type Safety

Well-typed expressions obey the desired safety guarantees such as calling every one-shot continuation at most once, calling effects only of the signature mentioned and evaluating an expression either makes it diverge or eventually reaches a value. In section 5 we prove type safety by giving the typing judgement a semantic interpretation that makes the notions of initial and final environments, performing an effect and having a type concrete.

The desired guarantee of well-typed expressions is safety which is formally defined as follows:

$$\text{safe } e := \forall e' \sigma'. e / \emptyset \rightsquigarrow^* e' / \sigma' \implies (e' \in \text{Val} \vee \exists e'' \sigma''. e' / \sigma' \rightsquigarrow e'' / \sigma'')$$

It states that any evaluation sequence of the expression starting from an empty heap, will either reach a value signalling the end of computation, or it will diverge. Most essentially, it will not get *stuck*, i.e., no reduction rule can be applied while itself not being a value. This notion of safety implies that safe expressions do not call a continuation more than once, every effect call is handled, and no type mismatch occurs. The reason for this lies in the operational semantics of the language, since there is no head reduction rule that starts from $\text{cont } \ell \ N$ when ℓ points to true. Additionally, an unhandled effect will eventually cause the top-level expression to be of the form $\text{eff } v \ N$ which is not a value and no reduction rule applies.

Using this notion of safety, we can define type safety to mean that expressions typed in empty initial and final environments and with the nil effect signature are safe:

$$\vdash e : \perp : \tau \dashv \implies \text{safe } e$$

To prove type safety we use a logical approach to *semantic type soundness* [Tim+22]. Types, signatures, copyable constraints and sub-typing relations will be given a semantic

meaning and the typing rules and judgements will be given a semantic interpretation that will imply safety (section 5).

3.3.2 Copyable Types

A Copyable type is a synonym for a non-substructural type meaning that its inhabitants can be used an arbitrary number of times. $Haffel_{os}$ only tracks one-shot continuations substructurally and so only the affine function type and types constructed from it are treated substructurally.

Figure 6 shows a simple derivation system to determine whether a type is copyable. A copy constraint $\text{Copy}_X \tau$ is parameterised with a set X of type variables that are assumed to be copyable. This set X is used in the rule for recursive types to specify the assumption that the recursive variable is copyable in the sub-derivation. We use the notation $\text{Copy} \tau$ to denote the copy constraint with no assumptions $\text{Copy}_\emptyset \tau$.

$\frac{\alpha \in X}{\text{Copy}_X \alpha}$	$\text{Copy}_X \text{void}$	$\text{Copy}_X ()$	$\text{Copy}_X \text{bool}$	$\text{Copy}_X \text{int}$
$\text{Copy}_X (\tau \xrightarrow{\sigma} \kappa)$	$\text{Copy}_X (!\tau)$	$\text{Copy}_X (\text{ref } \tau)$	$\text{Copy}_X (\forall \alpha. \tau)$	$\text{Copy}_X (\forall \theta. \tau)$
$\frac{\text{Copy}_X \tau \quad \text{Copy}_X \kappa}{\text{Copy}_X (\tau * \kappa)}$	$\frac{\text{Copy}_X \tau \quad \text{Copy}_X \kappa}{\text{Copy}_X (\tau + \kappa)}$	$\frac{\text{Copy}_{X \cup \{\alpha\}} \tau}{\text{Copy}_X (\mu \alpha. \tau)}$		

Figure 6: Copyable types

One-shot continuations are typed with the affine function type making it non-copyable. Base types and the unrestricted function type $(\tau \xrightarrow{\sigma} \kappa)$ are copyable because they do not capture one-shot continuations from the environment. The unrestricted type $(!\tau)$ is also copyable because it can only be formed from non-substructural types τ . Interestingly, a reference type $(\text{ref } \tau)$ is copyable regardless of its inner type τ which means that we can treat references non-substructurally even if they store one-shot continuations. To make this sound, loading from a reference is only allowed if the inner type is non-substructural. To retrieve the value of a reference that stores a substructural type we need to use the store operation that replaces its contexts with a new value first and returns the previous value. We choose to make polymorphic types $(\forall \alpha. \tau)$ and signatures $(\forall \theta. \tau)$ also copyable to allow multiple instantiations. Product $(\tau * \kappa)$ and sum $(\tau + \kappa)$ types are copyable according to their components. Lastly, recursive types $(\mu \alpha. \tau)$ are copyable if we can show that the inner type τ is copyable under the assumption that the recursive variable α is copyable.

The copyable constraint is extended to contexts by requiring that all variables are of a copyable type:

$\text{Copy } \emptyset$	$\frac{\text{Copy } \Gamma \quad \text{Copy } \tau}{\text{Copy } (x : \tau, \Gamma)}$
--------------------------	---

3.3.3 Relation on Signatures, Types and Environments

Figure 7 shows the subsumption ordering between types $\Sigma \vdash \tau \triangleleft \kappa$, between signatures $\Sigma \vdash \sigma \triangleleft \hat{\sigma}$ and environments $\Gamma \triangleleft \Gamma'$. Two types are related, $\tau \triangleleft \kappa$, if there exists a

judgement of the form $\emptyset \vdash \tau < \kappa$. Similarly, signatures are related, $\sigma < \hat{\sigma}$, if the signature judgement under the empty context Σ is derivable. The context Σ in the judgements represents a list of assumptions, where an assumption is either that two types, $\tau < \kappa$, or two signatures, $\sigma < \hat{\sigma}$, are related. The relation on environments is given directly from the inference rules.

There are six key rules that allow us to express the following requirements. The remainder of the rules simply extend the relations to apply to any type, signature and environment.

- Translate between a non-structural type and an unrestricted type (URES-INTRO and URES-ELIM).
- Use an unrestricted function in place where an affine function is expected (ARROW).
- Assign any effect signature to a non-effectful expression (EMPTY).
- Contract and weaken an environment (CONTRACTION and WEAKEN).

All three relations are pre-orders that do not satisfy anti-symmetry due to the presence of rules URES-INTRO and URES-ELIM. The judgement on types $\Sigma \vdash \kappa < \tau$ indicates that κ is subsumed by type τ under the context Σ . Similarly for effect signatures, $\Sigma \vdash \sigma < \hat{\sigma}$ states that σ is subsumed by signature $\hat{\sigma}$ under the context Σ . The context is used in REC and EFF to track the assumptions needed to show that the relation holds. The rules T-ASSUM and S-ASSUM allow discharging of assumptions. The relation on environments also defines the subsumption order but does not depend on a context.

The relation on types $\tau < \kappa$ allows an expression of type τ to be given the type κ . The `void` type serves as the minimal element in this relation (VOID). The rules AFF-ARROW and URES-ARROW define how to relate two function types of the same kind and rules PROD and SUM define the relation on products and sums to be component-wise. The relation for polymorphic types and signatures is defined in a similar way. To show that two recursive types are related, we need to show that their inner types τ and κ are related under the assumption that their recursion variables α and β are related. The rule is identical to Cardelli et al.'s subtyping rule for iso-recursive types in the Amber language [Car+86; Pie02].

The relation on signatures $\sigma < \hat{\sigma}$ states that an expression with signature σ also performs effects of signature $\hat{\sigma}$. The `nil` signature is a minimal element in this relation and thus allows a non-effectful expression to be given any signature. The EFF rule incorporates the sub-typing relation of types to that on effect signatures. Interestingly, it is covariant on the argument type and contravariant on the result type as opposed to function types. An expression with signature $\tau \Rightarrow \kappa$ performs effects where the effect value is of type τ and the response of the handler is of type κ . For it to be given the signature $\tau' \Rightarrow \kappa'$, the effect values must be of type τ' and handler responses will be of type κ' . It is thus covariant in the argument to be able to transform the effect value from type τ to type τ' and contravariant in the result to transform the handler's response from κ' to κ . As was the case with recursive types, the recursive variables are assumed to be in relation when showing that the components of the signature are related.

The rule CONS on environments utilises the relation on types to relate environments.

$\Sigma \vdash \tau < \kappa$		
$\frac{\text{REFL}}{\Sigma \vdash \tau < \tau}$	$\frac{\text{TRANS} \quad \Sigma \vdash \tau < \kappa \quad \Sigma \vdash \kappa < \iota}{\Sigma \vdash \tau < \iota}$	$\frac{\text{T-ASSUM} \quad \tau < \kappa \in \Sigma}{\Sigma \vdash \tau < \kappa}$
$\frac{\text{VOID}}{\Sigma \vdash \text{void} < \tau}$	$\frac{\text{URES-INTRO} \quad \text{Copy } \tau}{\Sigma \vdash \tau < !\tau}$	$\frac{\text{URES-ELIM}}{\Sigma \vdash !\tau < \tau}$
$\frac{\text{ARROW}}{\Sigma \vdash \tau \xrightarrow{\sigma} \kappa < \tau \xrightarrow{\sigma} \kappa}$	$\frac{\text{AFF-ARROW} \quad \Sigma \vdash \tau < \tau' \quad \Sigma \vdash \hat{\sigma} < \sigma \quad \Sigma \vdash \kappa' < \kappa}{\Sigma \vdash \tau' \xrightarrow{\hat{\sigma}} \kappa' < \tau \xrightarrow{\sigma} \kappa}$	
$\frac{\text{URES-ARROW} \quad \Sigma \vdash \tau < \tau' \quad \Sigma \vdash \hat{\sigma} < \sigma \quad \Sigma \vdash \kappa' < \kappa}{\Sigma \vdash \tau' \xrightarrow{\hat{\sigma}} \kappa' < \tau \xrightarrow{\sigma} \kappa}$		
$\frac{\text{PROD} \quad \Sigma \vdash \tau' < \tau \quad \Sigma \vdash \kappa' < \kappa}{\Sigma \vdash \tau' * \kappa' < \tau * \kappa}$	$\frac{\text{SUM} \quad \Sigma \vdash \tau' < \tau \quad \Sigma \vdash \kappa' < \kappa}{\Sigma \vdash \tau' + \kappa' < \tau + \kappa}$	
$\frac{\text{FORALL-TYPE} \quad \Sigma \vdash \tau < \kappa}{\Sigma \vdash \forall \alpha. \tau < \forall \alpha. \kappa}$	$\frac{\text{FORALL-SIG} \quad \Sigma \vdash \tau < \kappa}{\Sigma \vdash \forall \theta. \tau < \forall \theta. \kappa}$	$\frac{\text{REC} \quad \alpha < \beta, \Sigma \vdash \tau < \kappa}{\Sigma \vdash \mu \alpha. \tau < \mu \beta. \kappa}$
$\Sigma \vdash \sigma < \hat{\sigma}$		
$\frac{\text{S-ASSUM} \quad \sigma < \hat{\sigma} \in \Sigma}{\Sigma \vdash \sigma < \hat{\sigma}}$	$\frac{\text{TRANS} \quad \Sigma \vdash \sigma < \hat{\sigma} \quad \Sigma \vdash \hat{\sigma} < \sigma'}{\Sigma \vdash \sigma < \sigma'}$	
$\frac{\text{EMPTY}}{\Sigma \vdash \perp < \sigma}$	$\frac{\text{EFF} \quad \theta < \theta', \Sigma \vdash \tau < \tau' \quad \theta < \theta', \Sigma \vdash \kappa' < \kappa}{\Sigma \vdash \mu \theta. \forall \alpha. \tau \Rightarrow \kappa < \mu \theta'. \forall \alpha. \tau' \Rightarrow \kappa'}$	
$\Gamma' < \Gamma$		
$\frac{\text{REFL}}{\Gamma < \Gamma}$	$\frac{\text{TRANS} \quad \Gamma_1 < \Gamma_2 \quad \Gamma_2 < \Gamma_3}{\Gamma_1 < \Gamma_3}$	$\frac{\text{CONS} \quad \Gamma' < \Gamma \quad \emptyset \vdash \tau < \kappa}{x : \tau, \Gamma' < x : \kappa, \Gamma}$
$\frac{\text{WEAKEN} \quad x : \tau, \Gamma < \Gamma}{x : \tau, \Gamma < \Gamma}$	$\frac{\text{CONTRACTION} \quad \text{Copy } \tau}{x : \tau, \Gamma < x : \tau, x : \tau, \Gamma}$	$\frac{\text{SWAP}}{x : \tau, y : \kappa, \Gamma < y : \kappa, x : \tau, \Gamma}$

Figure 7: Subsumption Relations

3.4 Typing Examples

Before giving the formal definitions of the typing rules we firstly explain the intuition behind them by reviewing the three examples from section 2 but this time in $Haffel_{os}$. Throughout this section, $Haffel_{os}$ expressions will be additionally annotated with their types and the typing judgement they reside in. For illustration purposes, typing judgements specify only the substructural variables in their environments and copyable variables are assumed to exist in a separate persistent environment that is not shown. This contrasts with the formal presentation of the typing rules in section 3.5, where all free variables referenced in an expression exist in the environment regardless if they were substructural or not.

State Effect Figure 8 shows the factorial function $fact$ from section 2.1 but written in $Haffel_{os}$. The non-recursive, monomorphic state effect signature st is defined to be:

$$st := (() + \text{int}) \Rightarrow \text{int}$$

A sum of type $() + \text{int}$ is needed to call this effect which has the purpose of distinguishing between the two kinds of effects, get and put . $Haffel_{os}$ does not support proper named effects and as a partial solution we encode them using sums. The limitation of this approach is that the two effects must share the same result type. Thus we adapt the semantics of put by expecting its result to be the previous value of the state instead of just the unit value. The full-fledged $Haffel$ language does support named effect and is presented section 4.

```

let get : ()  $\xrightarrow{st}$  int =
   $\lambda () . \vdash \text{do } (\text{InjL } ()) : ( () + \text{int} ) \Rightarrow \text{int} : \text{int } \dashv$ 

let put : int  $\xrightarrow{st}$  int =
   $\lambda (s : \text{int}) . \vdash \text{do } (\text{InjR } s) : ( () + \text{int} ) \Rightarrow \text{int} : \text{int } \dashv$ 

let fact : int  $\xrightarrow{st}$  () =
  rec (f : int  $\xrightarrow{st}$  ()) (n : int) .  $\vdash$ 
    if  $\vdash 1 < n : st : \text{bool } \dashv$  then
       $\vdash \text{put } (\text{get } () * n) : st : \text{int } \dashv;$ 
       $\vdash f(n - 1) : st : () \dashv$ 
    else
       $\vdash () : st : () \dashv$ 

```

Figure 8: Typing of the stateful factorial function in $Haffel_{os}$

All three functions are typed with the unrestricted function type since they do not capture substructural variables from the environment. This can be seen from the typing judgements that contain no variables before and after their turnstiles. The helper functions get and put in fig. 8 perform the corresponding effects using the do construct.

$$\begin{array}{c}
\text{Sub} \frac{\vdash () : \perp : () \dashv \quad \perp <: (() + \text{int}) \Rightarrow \text{int}}{\vdash () : (() + \text{int}) \Rightarrow \text{int} : () \dashv} \\
\text{Left-Inj} \frac{\vdash \text{injL} () : (() + \text{int}) \Rightarrow \text{int} : () + \text{int} \dashv}{\vdash \text{do} (\text{injL} ()) : (() + \text{int}) \Rightarrow \text{int} : \text{int} \dashv} \\
\text{Do} \frac{\vdash \text{do} (\text{injL} ()) : (() + \text{int}) \Rightarrow \text{int} : \text{int} \dashv}{\vdash \text{do} (\text{injL} ()) : st : \text{int} \dashv} \\
\text{Unfold } st \\
\text{URes-Fun} \frac{\vdash \text{do} (\text{injL} ()) : st : \text{int} \dashv}{\vdash \lambda (). \text{do} (\text{injL} ()) : \perp : () \xrightarrow{st} \text{int} \dashv}
\end{array}$$

The complete typing derivation for the *get* function is shown above. An unrestricted function is typed with the nil signature to indicate that we do not evaluate under a lambda. Its body must be typed with the effect signature *st* that is present in its type. To perform an effect, the type of the argument passed to the perform call must match the argument type of the effect signature. Similarly, the result type of this perform call is the result type of the effect signature. In addition, values such as the unit are typed with the non-effectful signature. We thus utilise the subsumption rule and the relation on signatures to type the unit value. This is sound because a non-effectful expression can be typed with any effect signature.

The recursive function construct is used to type the *fact* expression. The if-else statement inside its body is typed in the usual way with the additional restriction that the effect signature of all three sub-expressions must be the same.

$$\begin{array}{c}
\vdots \\
\hline
\vdash \text{put} : st : \text{int} \xrightarrow{st} \text{int} \dashv \quad \text{int} \xrightarrow{st} \text{int} <: \text{int} \xrightarrow{st} \text{int} \quad \vdots \\
\text{App} \frac{\vdash \text{put} : st : \text{int} \xrightarrow{st} \text{int} \dashv \quad \vdash \text{get} () * n : st : \text{int} \dashv}{\vdash \text{put} (\text{get} () * n) : st : \text{int} \dashv}
\end{array}$$

The above partial derivation shows how the application of *put* is typed. As was the case with the if-else expression, the effect signatures of the two sub-expressions must match. In addition, the function produced by the evaluation of the *put* expression must be an affine and effectful function with the same signature. Using the subsumption rule, it is enough to show that *put* is typed with the unrestricted function type. The sub-typing relation between types gives us that unrestricted function types can be used in a context that expects an affine function type.

The two handlers for the effectful factorial expression are defined in fig. 9. Their derivations have the following form for some environments Γ_1, Γ_2 , expressions r and h , type τ and signature σ . The type and signature of the return branch r and effect branch h must match and the handler inherits this type τ and signature σ . The effect branch is defined to take two variables, a variable x of type $() + \text{int}$ that matches the argument type of the *st* signature and a one-shot continuation k of type $\text{int} \xrightarrow{\sigma} \tau$. The argument type of the continuation is the result type of the signature *st* and represents the response to the effect call. Importantly, the result type and signature of the continuation are also of type τ and σ because it is executed in the context of the handler. Lastly, the type derivation for expression *fact* n can depend on an environment Γ_1 and the resulting environment Γ_2 will be available in the return branch of the handler. Notice that environment Γ_2 can contain substructural variables because the return branch of a deep handler is executed at most once when only one-shot continuations are allowed. This does not hold for the effect branch because it is executed every time an effect is performed.

$$\frac{\begin{array}{c} \vdots \\ \hline \Gamma_1 \vdash \text{fact } n : st : () \dashv \Gamma_2 \end{array} \quad \begin{array}{c} \vdots \\ \hline \Gamma_2 \vdash r : \sigma : \tau \dashv \end{array} \quad \begin{array}{c} \vdots \\ \hline x : () + \text{int}, k : \text{int} \xrightarrow{\sigma} \tau \vdash h : \sigma : \tau \dashv \end{array}}{\Gamma_1 \vdash \text{deep-try fact } n \text{ with } \lambda x k. h \mid \lambda (). r : \sigma : \tau \dashv}$$

The *fact_ref* function defines an unrestricted function that takes an integer as parameter. It allocates a reference *r* to represent the state and installs a deep handler on the expression *fact n*. The observable signature σ of the handler is the nil signature and the resulting type τ is **int**. Since the continuation in the effect branch is substructural, it needs to be threaded explicitly through the initial and final environments of the typing judgements. The following derivation for the first branch of the match construct shows how this is done.

$$\text{App} \frac{\text{Var} \frac{}{k : \text{int} \multimap () \vdash k : \perp : \text{int} \multimap () \vdash} \quad \text{Frame} \frac{\text{Var} \frac{}{\vdash !r : \perp : \text{int} \vdash} \quad \text{Copy int} \frac{}{\vdash !r : \perp : \text{int} \vdash} \quad \text{Read} \frac{}{k : \text{int} \multimap () \vdash !r : \perp : \text{int} \vdash k : \text{int} \multimap ()}}{k : \text{int} \multimap () \vdash k (!r) : \perp : \text{int} \vdash}}$$

The right-to-left evaluation order of *Haffel_{os}* is evident from the typing derivation as the continuation is firstly threaded through the reference load before being used by the variable rule. To forward the continuation the Frame rule is used which *frames* the continuation by requiring a typing derivation for $!r$ which does not make use *k*. Importantly, we are allowed to read from the reference because **int** is a copyable type. For the second branch of the match construct we firstly update the value of the store and continue *k* with its previous value. Notice that a reference store returns the previous value of the reference as result instead of the unit value.

```

let fact_ref : int → int = λ (n : int).
  ⊢ let r = ref 1 : ⊥ : ref int ⊢ in
  ⊢ deep-try ⊢ (fact n) : st : () ⊢ with
    λ (x : () + int) (k : int → ())
      match x with
        λ (). k : int → () ⊢ k (! r) : ⊥ : int ⊢
        | λ (s : int). k : int → () ⊢ k (r := s) : ⊥ : int ⊢
        | λ (). ⊢ ! r : ⊥ : int ⊢

let fact_st : int → int = λ (n : int).
  (deep-try ⊢ (fact n) : st : () ⊢ with
    λ (x : () + int) (k : int → int → int).
      match x with
        λ (). k : int → int → int ⊢ λ (s : int). k s s : ⊥ : int → int ⊢
        | λ (s : int). k : int → int → int ⊢ λ (s' : int). k s' s') : ⊥ : int → int ⊢
    ) 1

```

Figure 9: Typing of the handlers for the factorial function in *Haffel_{os}*

The handler defined in the *fact_st* function uses the nil signature for the observable signature σ and its result type τ is **int** \multimap **int**. Its result is applied to the constant 1 which represents the initial state. Crucially, the result in both branches of the match construct is an affine function because both functions use the substructural continuation *k* in their body. The Aff-Fun typing rule as seen below allows an abstraction to use substructural variables in its body as long as it is typed with the affine function type.

$$\frac{\vdots \quad s : \text{int}, k : \text{int} \multimap \text{int} \multimap \text{int} \vdash k s s : \perp : \text{int} \vdash}{k : \text{int} \multimap \text{int} \multimap \text{int} \vdash \lambda s. k s s : \perp : \text{int} \multimap \text{int} \vdash} \text{Aff-Fun}$$

Generators from Iterators We introduce the typing rules for shallow handlers, type and effect polymorphism and non-substructural references through the Generators from Iterators example from section 2.2. Recall that an iterator is a higher-order function that takes a function that specifies how to consume each element. In *Haffel_{os}*, we give iterators the following type which also takes into consideration the effect signature of the consumer function:

$$\text{iterator } \tau := \forall \theta. (\tau \xrightarrow{\theta} ()) \xrightarrow{\theta} ()$$

Importantly, an iterator is an effect polymorphic function and its effect signature is inherited from the signature of its argument. This means that we can call an iterator with any effectful function as long as the argument and result types match. In this example, to transform an iterator to a generator we will use the yield effect *yd*:

$$\text{yd } \tau := \tau \circ ()$$

Generators on the other hand will be given the following unrestricted function type:

$$\text{generator } \tau := () \rightarrow \text{Option } \tau$$

Internally a generator will use a reference to track the status of the iteration, but *Haffel_{os}* does not treat reference operations as effectful and so it is given the nil signature. Generators can be used non-substructurally since the unrestricted function type is copyable.

Figure 10 shows the *generate* function from section 2.2 written in *Haffel_{os}*. Just like the previous example, the source code is annotated with typing judgements and extra type information. In contrast to the OCaml version, we define *generate* using a shallow handler written with the try-with syntax to illustrate how they are typed. To make *generate* type polymorphic, we wrap it with the type abstraction construct $\Lambda.e$ which has the following typing rule:

$$\text{Forall-Type-Intro} \frac{\vdash e : \perp : \tau \dashv}{\vdash \Lambda.e : \perp : \forall \alpha. \tau \dashv}$$

The type $\forall \alpha. \tau$ states that its inhabitants are thunks that do not produce user effects when evaluated. In addition, the result of every thunk can be given the type τ for any instantiation of α . A polymorphic expression is usually instantiated multiple times and for this reason it is typed with empty substructural environments. This has the result of making the universal type copyable.

We define *yield* as a function that simply calls an effect with its argument as the effect value. In addition, we allocate a reference r that will be captured by the resulting generator and will be used to track the state of the iteration. It is initialised with a thunk that begins the iteration by calling *iter* with the yield function. To do this, we firstly instantiate *iter* with the yield signature using the $e \langle \rangle$ construct. The following derivation shows how this signature instantiation is typed.

$$\begin{array}{c} \text{Var} \frac{}{\vdash \text{iter} : \perp : \forall \theta. (\alpha \xrightarrow{\theta} ()) \xrightarrow{\theta} () \dashv} \quad \perp < \text{yd } \alpha \\ \text{Sub} \frac{}{\vdash \text{iter} : \perp : \forall \theta. (\alpha \xrightarrow{\theta} ()) \xrightarrow{\theta} () \dashv} \\ \text{Forall-Sig-Elim} \frac{\vdash \text{iter} : \text{yd } \alpha : \forall \theta. (\alpha \xrightarrow{\theta} ()) \xrightarrow{\theta} () \dashv}{\vdash \text{iter } \langle \rangle : \text{yd } \alpha : (\alpha \xrightarrow{\text{yd } \alpha} ()) \xrightarrow{\text{yd } \alpha} () \dashv} \end{array}$$

To make references non-substructural regardless of the type of values they store, reading from a reference of type $\text{ref } \tau$ requires the inner type τ to be copyable. This ensures that we do not use a value of a substructural type multiple times by simply

```

let generate :  $\forall \alpha. \text{iterator } \alpha \rightarrow \text{generator } \alpha =$ 
   $\Lambda : \forall \alpha. \lambda (\text{iter} : \text{iterator } \alpha).$ 
    let yield :  $\alpha \xrightarrow{yd} () = \lambda (x : \alpha). x : \alpha \vdash \text{do } x : yd \alpha : () \dashv \text{in}$ 
    let r :  $\text{ref } ( () \xrightarrow{yd} () ) = \text{ref } (\lambda (). \vdash \text{iter } () \text{ yield} : yd \alpha : () \dashv) \text{ in}$ 
       $\lambda (). \text{let comp} : () \xrightarrow{yd} () = (r := \vdash \lambda (). ()) : \perp : () \xrightarrow{yd} () \dashv)$ 
        try comp :  $() \xrightarrow{yd} () \vdash \text{comp } () : yd \alpha : () \dashv \text{with}$ 
           $(\lambda (x : \alpha) (k : () \xrightarrow{yd} ())).$ 
             $x : \alpha, k : () \xrightarrow{yd} () \vdash r := k : \perp : () \xrightarrow{yd} () \dashv x : \alpha;$ 
             $x : \alpha \vdash \text{some } x : \perp : \text{Option } \alpha \dashv$ 
            |  $(\lambda (). \vdash \text{none} : \perp : \text{Option } \alpha \dashv)$ 

```

Figure 10: Typing of the generate function in $Haffel_{os}$

storing it behind a reference. Instead, the only operation we can perform on such references is to swap its contents with another substructural value. The store operation allows us to perform this swapping because at every store the previous value is returned as result. By requiring that we replace the contents of the reference at every read, we ensure that we can never get the same value twice.

Indeed in the *generate* function, the reference r stores an affine function which is substructural and so we are not allowed to directly read from it. To access r , we update its contents with the affine function $\lambda (). ()$.

The format of type derivations for shallow handlers that service the yield effect is shown below. There are two key differences of these derivations from that of deep handlers. Firstly, an additional substructural environment Γ can be shared between the return and effect branch of the handler. A shallow handler will not reinstate itself in the continuation and thus only one of the two branches will be evaluated. Thus the variables in the environment Γ will be used at most once. Additionally, to reflect the fact that the continuation is not executed in the context of the same handler, its effect signature and result type is yield and unit. These are inherited from the *comp* $()$ expression.

$$\frac{\frac{\vdots}{\Gamma_1 \vdash \text{comp } () : \alpha \Rightarrow () : () \dashv \Gamma_2} \quad \frac{\vdots}{\Gamma_2; \Gamma \vdash r : \sigma : \tau \dashv \Gamma_3} \quad \frac{\vdots}{\Gamma'; \Gamma \vdash h : \sigma : \tau \dashv \Gamma_3}}{\Gamma_1; \Gamma \vdash \text{try comp}() \text{ with } \lambda x k. h \mid \lambda (). r : \sigma : \tau \dashv \Gamma_3}$$

The shallow handler used for the *generate* function uses the nil signature for σ and $\text{Option } \alpha$ for the result type τ . When a yield effect is produced, the reference is updated with the new state of the iteration tracked by the continuation k and the element x is returned.

Cooperative Concurrency To motivate the need for recursive and polymorphic effect signatures we briefly describe how to type the library for Cooperative Concurrency from section 2.3. The example relied on promises to represent asynchronous computations and so in $Haffel_{os}$ they are given the following type. It is a direct translation from the *OCaml 5* version with the only difference that one-shot continuations are given the affine function type in our language.

$\text{promise } \tau := \text{ref } (\tau + \text{list } (\tau \multimap ()))$

The signature for the **Async** and **Await** effects is as follows. Since $Haffel_{os}$ does not have named effects we take the same approach as with the State Effect example where we use a sum type to distinguish between the two effects. The **Async** effect is selected using the left injection and the right injection is used for the **Await** effect.

$$coop := \mu\theta. \forall\beta. (\ () \xrightarrow{\theta} !\beta) + promise (!\beta) \Rightarrow (promise !\beta) + !\beta$$

Crucially, the computation that should be performed asynchronously may itself perform **Async** and **Await** calls, so it should be given the effect signature $coop$. The recursive effect signature allows us to express exactly this, by allowing the type of the effect argument to reference the same signature that we are defining. In addition, a client of this library should be able to call the **Async** effect with any computation without requiring its result type to be of a certain type. The polymorphic signature gives us this possibility by allowing a client to instantiate β with a concrete type and thus use an instance of $coop$. The type variable β is wrapped with the unrestricted type constructor to indicate that type variable β must only be instantiated with a copyable type. This is because there might be multiple **Await** calls to the same promise and thus the value of type β behind a promise might be used multiple times.

The result type of the signature is again a sum type to allow the handler to respond with two different values according to the effect requested. Unfortunately, using a sum type has its limitations, specifically that it does not rule out the possibility of immediately getting the result of a computation when calling **Async** or getting a promise back when calling **Await**. We can recover the expected behaviour of the *async* and *await* functions as in the OCaml version, by defining them as shown in fig. 11. In section 4 we introduce proper named effects to $Haffel_{os}$ where each effect is given its own signature and as a result this workaround is not needed.

```

let async :  $\forall\alpha. (\ () \xrightarrow{coop} !\alpha) \xrightarrow{coop} promise !\alpha =$ 
   $\Lambda : \forall\alpha. \lambda (c : (\ () \xrightarrow{coop} !\alpha)).$ 
    match  $c : (\ () \xrightarrow{coop} !\alpha) \vdash (\mathbf{do} (\mathbf{injL} c)) : coop : (promise !\alpha) + !\alpha \dashv$  with
       $\lambda (p : promise !\alpha). \vdash p : coop : promise !\alpha \dashv$ 
    |  $\lambda (a : !\alpha). \vdash \mathbf{ref} (\mathbf{injL} a) : coop : promise !\alpha \dashv$ 

let await :  $\forall\alpha. promise !\alpha \xrightarrow{coop} !\alpha =$ 
   $\Lambda : \forall\alpha. \mathbf{rec} (f : promise !\alpha \xrightarrow{coop} !\alpha) (p : promise !\alpha).$ 
    match  $\vdash (\mathbf{do} (\mathbf{injR} p)) : coop : (promise !\alpha) + !\alpha \dashv$  with
       $\lambda (p : promise !\alpha). \vdash f p : coop : !\alpha \dashv$ 
    |  $\lambda (a : !\alpha). \vdash a : coop : !\alpha \dashv$ 

```

Figure 11: Typing of the *async* and *await* function in $Haffel_{os}$

In the *async* function we pattern match on the result of the effect call and if the handler replies immediately with the result of the computation we firstly wrap it in a new promise. For the *await* function we repeatedly call the effect until the handler replies with its result instead of a promise.

By zooming in the typing judgement of the first perform call we can see how the recursively polymorphic signature is used. The following partial derivation shows that the effect argument will instantiate the signature variable θ with the signature $coop$. In addition, the effect argument is a computation of return type $!\alpha$, thus the effect signature is instantiated with α at type variable β . The result type of the effect call

must be compatible with this signature instantiation and thus the result type uses type α .

$$\text{Do} \frac{\frac{\vdots}{c : () \xrightarrow{\text{coop}} !\alpha \vdash \text{injL } c : \text{coop} : (() \xrightarrow{\text{coop}} !\alpha) + \text{promise } (!\alpha) \dashv}}{c : () \xrightarrow{\text{coop}} !\alpha \vdash \text{do } (\text{injL } c) : \text{coop} : (\text{promise } !\alpha) + !\alpha \dashv}}$$

3.5 Typing Rules

Figure 12 and fig. 13 show the typing rules of $Haffel_{os}$.

Polymorphic Lambda Calculus The typing rules for the base types UNIT, BOOL and INT state that the environment is not used and that they do not produce effects. Indeed values are not effectful because they are not active effect expressions $\mathbf{eff} \ v \ N$ and can not get further evaluated. Variable expressions are also non-effectful as witnessed by the VAR rule. This is because the call-by-value evaluation strategy of $Haffel_{os}$ guarantees that variables can only be substituted with values during execution. Importantly, variables in the environment must be used affinely and thus variable x does not appear in the final environment.

The typing of applications is shown in rule APP. All three computations, the expression e_1 that produces an abstraction, the produced abstraction itself and the argument expression e_2 must conform to the same effect signature to ensure that the overall signature is σ . The initial environment Γ_1 is also threaded between sub-derivations according to the evaluation order.

There are two typing rules for the function types, AFF-FUN for the affine and URES-FUN for the unrestricted case. In both cases the initial environment is split into two, Γ_1 represents the part that is used by the body of the abstraction and Γ_2 for the unused part that is forwarded to the final environment. Interestingly, both function types have an empty final environment in their sub-derivation. Since we do not evaluate under an abstraction, evaluation of the abstraction body only happens during application. As a result, the final environment produced from evaluating the body will not be available and so it can not be referenced in the final environment of $\lambda x. e$.

The AFF-FUN typing rule shows that a function typed with the affine function arrow can access any variable in its environment. Since it may access variables with a substructural type, the affine function type itself should be treated as substructural. To make the unrestricted function type non-substructural, in URES-FUN there is the additional restriction that the environment that is used by the abstraction is copyable. Recursive functions must also be given an unrestricted function type because they allow themselves to be called inside their body and can thus be called multiple times.

To give an expression a polymorphic type the rules FORALL-TYPE-INTRO and FORALL-SIG-INTRO are used. Both require the inner expression e to be well-typed with type τ where the binder (α or θ) is set to be free. We assume the Barendregt convention and thus α and θ are assumed to be distinct from any other type variables found in environment Γ_1 or in type τ [Bar84]. Instantiation is typed using the rules FORALL-TYPE-ELIM and FORALL-SIG-ELIM. We note that an expression e which has a polymorphic effect type $\forall\theta. \tau$ can instantiate θ with an effectful signature $\hat{\sigma}$ that is different from its own signature σ . The reason for this lies in the signature introduction typing rule FORALL-SIG-INTRO, where its sub-derivation states that expression e is non-effectful for any instantiation of signature θ .

The OPERATION rule concerns operations. The typing $\vdash_{Op} \odot : \tau \rightarrow \kappa$ states that \odot is a valid binary operation that takes arguments of the corresponding types. These operations are non-effectful and can be performed multiple times thus the unrestricted function type is used. We assume that the judgment \vdash_{Op} is defined only with integer argument types for operations $+$, $-$ and $<$. We further assume that equality $==$ is only defined for base types closed over product, sum and unrestricted type constructors.

In the typing rule IF-ELSE all sub-derivations must adhere to the same effect signature and the environment is passed along according to the evaluation order. Importantly, only one of the two expressions e_2 and e_3 will be evaluated and so we can share the context Γ_2 between the two sub-derivations.

Products and Sums The typing rules for sum constructors LEFT-INJ and RIGHT-INJ, sum eliminator MATCH, pair constructor PAIR, and pair eliminator PAIR-ELIM follow a similar pattern where the environment is threaded along the sub-derivations according to the right-to-left evaluation order. In all cases, the effect signature of the expression must agree with that of its sub-expressions. The additional constraint in PAIR-ELIM that x_1 and x_2 do not appear in the final environment Γ_3 ensures that the variables do not escape their scope.

References and Recursive Types Reference allocation has a standard typing rule ALLOC with the environment and effect signature depending on the inner expression. The typing rule for reading from a reference does not offer any surprises apart from the fact that inner type τ must be copyable. As seen from rule WRITE, the resulting type of a reference write operation is type τ which represents the previous value stored in the reference. Together with the fact that reference types are treated non-substructurally, it allows us to store one-shot continuations behind references allowing them to be shared between different expressions. This is a crucial feature of *Haffel_{os}* that we use to type the Cooperative Concurrency example from section 2.3.

Recursive types are typed in the usual way, rule REC-FOLD requires the sub-expression to be typed with the unfolded once version of $\mu\alpha. \tau$ and rule REC-UNFOLD is its inverse.

Effects The DO rule describes the typing of an effect call. To perform an effect the signature has to be $\mu\theta. \forall\alpha. \iota \Rightarrow \kappa$ for some types ι and κ , which indicates that the expression is indeed effectful. The type of the argument passed to the effect call must be type ι and the effect handler can only respond by calling the continuation with a value of type κ which will take the place of the do e expression. Since the signature is recursive over variable θ and polymorphic over type variable α , the argument and result type can depend on both of these variables. To handle signature recursion, the recursive variable θ is unfolded once at both the argument and result type. For signature polymorphism, the type of the argument passed to the effect call will instantiate α with a concrete type τ . The result type of the whole expression do e must also reflect this instantiation.

In both handlers, the return branch will depend on a variable x that represents the value that expression e has evaluated to. In addition, the signature and result type of the return branch must match with that of the handler as a whole. To type the effect branches, the effect call value x is required to have type τ and the continuation k must have argument type κ due to the effect signature being $\mu\theta. \forall\alpha. \tau \Rightarrow \kappa$. Signature recursion is handled by unfolding once at the recursion variable θ . For signature polymorphism, the effect branches must be well-typed for any instantiation of type variable α . To show this we let α be a free type variable in the effect branch. As with the previous typing rules, we assume the Barendregt convention and thus α will be distinct from all other type variables found in ι , κ and Γ thus indeed showing that the effect branch is well-typed for any instantiation.

Due to the non-local evaluation order of effectful expressions, in both handlers the environments are threaded along the sub-derivations in an unusual way. The initial environment is split into Γ_1 and Γ with Γ_1 being forwarded to the expression e and Γ towards the branches of the effect handler. The inner expression is the first to be evaluated and when a value is reached the return branch of the handler is taken thus explaining the presence of Γ_2 in the initial environment of the return branch. The effect branch on the other hand can only access the Γ environment and not Γ_2 . At the moment when the effect branch takes control, the expression e has not finished executing and thus the environment Γ_2 is not yet available.

The two typing rules differ in two important ways. The shallow handler does not reinstate itself in the continuation so only one of its two branches is taken and it used just once. As a result, the environment Γ can be shared between the two sub-derivations as is. In contrast, the effect branch of the deep-handler can be called multiple times when more than one effect call occurs. This forces us to make the environment Γ copyable and thus allow its variables to be used in an arbitrary way. Additionally the return type and effect signature of the continuations are different in the two typing rules. A deep handler is persistent because it reinstates itself in the continuation and as a result it inherits the return type and effect signature of the handler.

Subtyping and Framing The SUB rule utilises the relations defined in section 3.3.3. The environment relation $\Gamma < \Gamma'$ can be read as an implication and so from the environment Γ we can deduce that Γ' also holds. This explains the direction of the relation between the initial environments and the final environments. For $\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2$ to hold we assume that the initial environment Γ_1 holds and in order to utilise the derivation $\Gamma'_1 \vdash e : \hat{\sigma} : \hat{\kappa} \dashv \Gamma'_2$ we must first show that the environment Γ'_1 also holds. In contrast, by assumption we have that the final environment Γ'_2 holds after evaluation of expression e and must thus have $\Gamma'_2 < \Gamma_2$ to show Γ_2 . Similarly, the relations $\sigma < \hat{\sigma}$ and $\tau < \hat{\kappa}$ indicate the subsumption order, and so if expression e performs effects of signature σ it also performs $\hat{\sigma}$ and if it has type τ it also has type $\hat{\kappa}$.

The rule FRAME allows an environment Γ' to be framed by requiring a sub-derivation where Γ' does not appear in the initial nor the final environment. The frame rule makes typing modular by allowing expressions to be typed with only the part of the environment they use.

$$\boxed{\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}$$

Polymorphic Lambda Calculus

<p>UNIT $\Gamma \vdash () : \perp : () \dashv \Gamma$</p>	<p>BOOL $\Gamma \vdash b : \perp : \mathbf{bool} \dashv \Gamma$</p>	<p>INT $\Gamma \vdash i : \perp : \mathbf{int} \dashv \Gamma$</p>
<p>VAR $x : \tau, \Gamma \vdash x : \perp : \tau \dashv \Gamma$</p>	<p>APP $\frac{\Gamma_2 \vdash e_1 : \sigma : \tau \xrightarrow{\sigma} \kappa \dashv \Gamma_3 \quad \Gamma_1 \vdash e_2 : \sigma : \tau \dashv \Gamma_2}{\Gamma_1 \vdash e_1 e_2 : \sigma : \kappa \dashv \Gamma_3}$</p>	
<p>AFF-FUN $\frac{x : \tau, \Gamma_1 \vdash e : \sigma : \kappa \dashv}{\Gamma_1; \Gamma_2 \vdash \lambda x. e : \perp : \tau \xrightarrow{\sigma} \kappa \dashv \Gamma_2}$</p>	<p>URES-FUN $\frac{\text{Copy } \Gamma_1 \quad x : \tau, f : \tau \xrightarrow{\sigma} \kappa, \Gamma_1 \vdash e : \sigma : \kappa \dashv}{\Gamma_1; \Gamma_2 \vdash \mathbf{rec } f x. e : \perp : \tau \xrightarrow{\sigma} \kappa \dashv \Gamma_2}$</p>	
<p>FORALL-TYPE-INTRO $\frac{\text{Copy } \Gamma_1 \quad \Gamma_1 \vdash e : \perp : \tau \dashv}{\Gamma_1; \Gamma_2 \vdash \Lambda. e : \perp : \forall \alpha. \tau \dashv \Gamma_2}$</p>	<p>FORALL-TYPE-ELIM $\frac{\Gamma_1 \vdash e : \sigma : \forall \alpha. \tau \dashv \Gamma_2}{\Gamma_1 \vdash e \langle \rangle : \sigma : \tau [\kappa/\alpha] \dashv \Gamma_2}$</p>	
<p>FORALL-SIG-INTRO $\frac{\text{Copy } \Gamma_1 \quad \Gamma_1 \vdash e : \perp : \tau \dashv}{\Gamma_1; \Gamma_2 \vdash \Lambda. e : \perp : \forall \theta. \tau \dashv \Gamma_2}$</p>	<p>FORALL-SIG-ELIM $\frac{\Gamma_1 \vdash e : \sigma : \forall \theta. \tau \dashv \Gamma_2}{\Gamma_1 \vdash e \langle \rangle : \sigma : \tau [\hat{\sigma}/\theta] \dashv \Gamma_2}$</p>	
<p>OPERATION $\frac{\vdash_{\text{Op}} \odot : \tau \rightarrow \kappa \rightarrow \iota \quad \Gamma_1 \vdash e_2 : \sigma : \kappa \dashv \Gamma_2 \quad \Gamma_2 \vdash e_1 : \sigma : \tau \dashv \Gamma_3}{\Gamma_1 \vdash e_1 \odot e_2 : \sigma : \iota \dashv \Gamma_3}$</p>	<p>IF-ELSE $\frac{\Gamma_1 \vdash e_1 : \sigma : \mathbf{bool} \dashv \Gamma_2 \quad \Gamma_2 \vdash e_2 : \sigma : \tau \dashv \Gamma_3 \quad \Gamma_2 \vdash e_3 : \sigma : \tau \dashv \Gamma_3}{\Gamma_1 \vdash \mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3 : \sigma : \tau \dashv \Gamma_3}$</p>	

Products and Sums

<p>LEFT-INJ $\frac{\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}{\Gamma_1 \vdash \mathbf{injL } e : \sigma : \tau + \kappa \dashv \Gamma_2}$</p>	<p>RIGHT-INJ $\frac{\Gamma_1 \vdash e : \sigma : \kappa \dashv \Gamma_2}{\Gamma_1 \vdash \mathbf{injR } e : \sigma : \tau + \kappa \dashv \Gamma_2}$</p>
<p>MATCH $\frac{\Gamma_1 \vdash e_1 : \sigma : \tau + \kappa \dashv \Gamma_2 \quad \Gamma_2 \vdash e_2 : \sigma : \tau \xrightarrow{\sigma} \iota \dashv \Gamma_3 \quad \Gamma_2 \vdash e_3 : \sigma : \kappa \xrightarrow{\sigma} \iota \dashv \Gamma_3}{\Gamma_1 \vdash \mathbf{match } e_1 \mathbf{ with } e_2 \mid e_3 : \sigma : \iota \dashv \Gamma_3}$</p>	<p>PAIR $\frac{\Gamma_2 \vdash e_1 : \sigma : \tau \dashv \Gamma_3 \quad \Gamma_1 \vdash e_2 : \sigma : \kappa \dashv \Gamma_2}{\Gamma_1 \vdash (e_1, e_2) : \sigma : \tau * \kappa \dashv \Gamma_3}$</p>
<p>PAIR-ELIM $\frac{\Gamma_1 \vdash e_1 : \sigma : \tau * \kappa \dashv \Gamma_2 \quad x_2 : \kappa, x_1 : \tau, \Gamma_2 \vdash e_2 : \sigma : \iota \dashv \Gamma_3 \quad x_1, x_2 \notin \text{Dom } \Gamma_3}{\Gamma_1 \vdash \mathbf{let } (x_1, x_2) = e_1 \mathbf{ in } e_2 : \sigma : \iota \dashv \Gamma_3}$</p>	

References

<p>ALLOC $\frac{\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}{\Gamma_1 \vdash \mathbf{ref } e : \sigma : \mathbf{ref } \tau \dashv \Gamma_2}$</p>	<p>READ $\frac{\text{Copy } \tau \quad \Gamma_1 \vdash e : \sigma : \mathbf{ref } \tau \dashv \Gamma_2}{\Gamma_1 \vdash !e : \sigma : \tau \dashv \Gamma_2}$</p>	<p>WRITE $\frac{\Gamma_1 \vdash e_2 : \sigma : \tau \dashv \Gamma_2 \quad \Gamma_2 \vdash e_1 : \sigma : \mathbf{ref } \tau \dashv \Gamma_3}{\Gamma_1 \vdash e_1 := e_2 : \sigma : \tau \dashv \Gamma_3}$</p>
---	---	--

Figure 12: Typing rules (Part 1)

$$\boxed{\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}$$

Recursive Types

$$\begin{array}{c} \text{REC-FOLD} \\ \frac{\Gamma_1 \vdash e : \sigma : \tau [\mu\alpha. \tau/\alpha] \dashv \Gamma_2}{\Gamma_1 \vdash \text{fold } e : \sigma : \mu\alpha. \tau \dashv \Gamma_2} \end{array} \qquad \begin{array}{c} \text{REC-UNFOLD} \\ \frac{\Gamma_1 \vdash e : \sigma : \mu\alpha. \tau \dashv \Gamma_2}{\Gamma_1 \vdash \text{unfold } e : \sigma : \tau [\mu\alpha. \tau/\alpha] \dashv \Gamma_2} \end{array}$$

Effects

$$\begin{array}{c} \text{Do} \\ \sigma = (\mu\theta. \forall\alpha. \iota \Rightarrow \kappa) \\ \frac{\Gamma_1 \vdash e : \sigma : \iota [\sigma/\theta] [\tau/\alpha] \dashv \Gamma_2}{\Gamma_1 \vdash \text{do } e : \sigma : \kappa [\sigma/\theta] [\tau/\alpha] \dashv \Gamma_2} \end{array}$$

$$\begin{array}{c} \text{SHALLOW-HANDLER} \\ \sigma = (\mu\theta. \forall\alpha. \iota \Rightarrow \kappa) \quad \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2 \quad x, k \notin \text{Dom } \Gamma_3 \\ \frac{x : \tau, \Gamma_2; \Gamma \vdash r : \hat{\sigma} : \tau' \dashv \Gamma_3 \quad x : \iota [\sigma/\theta], k : \kappa [\sigma/\theta] \xrightarrow{\sigma} \tau, \Gamma \vdash h : \hat{\sigma} : \tau' \dashv \Gamma_3}{\Gamma_1; \Gamma \vdash \text{try } e \text{ with } \lambda x k. h \mid \lambda x. r : \hat{\sigma} : \tau' \dashv \Gamma_3} \end{array}$$

$$\begin{array}{c} \text{DEEP-HANDLER} \\ \sigma = (\mu\theta. \forall\alpha. \iota \Rightarrow \kappa) \quad \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2 \quad x, k \notin \text{Dom } \Gamma \quad \text{Copy } \Gamma \\ \frac{x : \tau, \Gamma_2; \Gamma \vdash r : \hat{\sigma} : \tau' \dashv \Gamma \quad x : \iota [\sigma/\theta], k : \kappa [\sigma/\theta] \xrightarrow{\hat{\sigma}} \tau', \Gamma \vdash h : \hat{\sigma} : \tau' \dashv \Gamma}{\Gamma_1; \Gamma \vdash \text{deep-try } e \text{ with } \lambda x k. h \mid \lambda x. r : \hat{\sigma} : \tau' \dashv \Gamma} \end{array}$$

Sub-Typing

$$\begin{array}{c} \text{SUB} \\ \frac{\Gamma'_1 \vdash e : \sigma : \kappa \dashv \Gamma'_2 \quad \Gamma_1 < \Gamma'_1 \quad \Gamma'_2 < \Gamma_2 \quad \sigma < \hat{\sigma} \quad \kappa < \tau}{\Gamma_1 \vdash e : \hat{\sigma} : \tau \dashv \Gamma_2} \end{array}$$

Framing

$$\begin{array}{c} \text{FRAME} \\ \frac{\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}{\Gamma'; \Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma'; \Gamma_2} \end{array}$$

Figure 13: Typing rules (Part 2)

4 Complete Language

The one-shot language $Haffel_{os}$ described in section 3 is able to type programs that utilise effect handlers while at the same time guarantee that continuations associated with effects are used at most once. While this language is powerful enough to type all three examples from section 2, here we consider the full-fledged language $Haffel$ which extends $Haffel_{os}$ in two ways:

1. **Multi-shot Effects:** Allow multi-shot use of continuations and thus increase the expressivity of the language.
2. **Named Effects:** Group multiple effect signatures into effect rows and thus enhance the effect specifications.

Multi-shot effects, that is effects that have their continuation resumed more than once, offer additional expressive power over their one-shot counterparts which allows them to express even more abstractions such as backtracking, probabilistic programming and process duplication. In addition, for certain class of programs, implementations that make use of multi-shot effects offer asymptotically better performance than the one-shot or non-effectful versions [HLL20]. It is thus beneficial for a language to support both kinds of effects but it has to be done in a controlled manner: the type system must be able to distinguish between one- and multi-shot effects by tracking how continuations are called. We will extend $Haffel_{os}$ by enhancing its substructural type system to track this call usage of continuations. In this way, we can achieve the best of both worlds by increasing the scope of what can be expressed in the language and at the same time alleviate the problems of multi-shot continuations by making the compiler aware of how they are used. With the presence of this usage information, the compiler can implement continuations in the most efficient way, and it can choose to apply only sound optimisations to expressions according to whether they produce multi-shot effects.

In addition, $Haffel_{os}$ supports only a single effect signature and in order to distinguish between different effects we must encode them using sum types in a slightly awkward and unsatisfactory manner. This contrasts with usual presentations of effect systems that utilise some form of effect rows to group multiple signatures together [HL16; Lei14; Bie+17; VP23]. In such systems, effect signatures are referenced by a name that the user selects to perform the desired effect. Along this line of work, we scale the effect system of $Haffel_{os}$ by introducing named effects in the form of effect rows. Our approach to effect rows is inspired by the calculus of Leijen where duplicate effect names are allowed and are distinguished from their non-duplicate counterparts [Lei17]. We further extend effect polymorphism to range over rows using the notion of effect lifting which is a concept introduced by Biernacki et al. [Bie+17].

The $Haffel$ language is based on the untyped $Hazel$ language that allows both one- and multi-shot use of unnamed effects. The $Hazel$ language is formalised in the Coq proof assistant but it is not described in any published work [VP20]. In essence, it is a fusion of two untyped languages, HH that allows only one-shot unnamed effects and $Maze$ that allows unnamed multi-shot effects [VP21; Vil22].

Section 4.1 begins with an overview of the key features of $Haffel$, section 4.2 gives its syntax and semantics, section 4.3 gives the sub-typing relation and lastly section 4.4 gives the typing rules.

4.1 Overview

Multi-Shot Effects We illustrate through an example the main idea on how we type multi-shot effects. The *decide* effect describes a non-deterministic computation that branches the computation into two possible execution paths. It can be used to

implement backtracking behaviour by resuming the associated continuation of the effect more than once according to whether the first execution path has found a solution. For this reason, *decide* is a multi-shot effect when used to implement backtracking.

Figure 14 shows the type annotated version of the program that was introduced in the introduction. The *decide* effect is given a multi-shot signature with result type `bool` and its handler resumes the continuation with both boolean values followed by a logical *or* operation on the results. Since effect specifications are no longer a single signature but are instead effect rows, we wrap the effect into a singleton row and use the row $\langle \rangle$ to describe the absence of effects.

```

let main : ()  $\xrightarrow{\langle decide : () \Rightarrow bool \rangle}$  bool = (λ ().
  let x : ref bool = ref true
  in ⊢ x := doM decide () && !x : ⟨decide : () ⇒ bool⟩ : bool ⊢;
    ⊢ !x : ⟨⟩ : bool ⊢) in

deep-try ⊢ main () : ⟨decide : () ⇒ bool⟩ : bool ⊢ with decide by
  λ () (k : bool → ()) . ⊢ k false || k true : ⟨⟩ : bool ⊢
  | λ (x : bool) . ⊢ x : ⟨⟩ : bool ⊢

```

Figure 14: Typing of the stateful factorial function in *Haffel*

The key idea lies in the typing of the handler. The *decide* effect is given a multi-shot signature $(() \Rightarrow \text{bool})$ instead of the one-shot version $(() \Rightarrow \text{bool})$ and as a result we can type the continuation as an unrestricted function. We can not type *decide* as a one-shot effect as that would give the continuation an affine function type $\text{bool} \multimap ()$ which can only be used once. Of course to ensure type safety, we additionally need to make sure the continuation does not capture non-substructural resources (such as other one-shot continuations). This is done through the typing of the *main* function, by ensuring that an expression that produces a multi-shot effect does not capture substructural resources. In the above example, *main* does not capture any substructural resources and thus it can be given a multi-shot effect.

This example shows that the type system satisfies one of its main requirements: one-shot effects can only be handled in a one-shot way. As a result, a compiler is free to apply optimisations to expressions that perform one-shot effects since it can be certain that the effect will not be handled in a multi-shot way.

Named Effects The examples of section 3.4 show how we can encode multiple effects via the single effect signature of *Haffel*_{os}. The idea is that we can use a sum type as the argument type of the effect signature and to select an effect we use the appropriate injection into the sum. For instance, in the State example (section 3.4), we use the following signature to encode the *get* and *put* signature:

$$st := (() + \text{int}) \Rightarrow \text{int}$$

The first injection corresponds to the *get* effect whereas the second injection corresponds to the *put* effect. A limitation of this approach is that the result type has to be shared between the two effects. Ideally, we would like that every effect has its own signature, with its own argument and result types. So with proper named effects, we would have the following effect row:

$$st := \langle (get : () \Rightarrow \text{int}), (put : \text{int} \Rightarrow ()) \rangle$$

Named effects allow effects to be referenced by name and each effect name has its own signature associated with it. As a result, the result type of the effects is not shared and we can express the two effects, *get* and *put*, with their own distinct signatures.

Cooperative Concurrency: A more compelling use case of named effects that showcases their enhanced specification language is the Cooperative Concurrency example from section 3.4.

In *Haffel_{os}* we have used the following effect signature to describe the *async* and *await* effects:

$$coop := \mu\theta. \forall\alpha. (\ () \stackrel{\theta}{\circ} !\alpha) + promise (!\alpha) \Rightarrow (promise !\alpha) + !\alpha$$

There are two problems with this approach. Firstly, the two effects need to share the same result type even though one has the type $(promise !\alpha)$ and the other has $!\alpha$. A sum type is used to capture both result types, but it does not guarantee the desired distinction such as that calling *await* will not return a promise. Additionally, effects are distinguished by the appropriate injection which must be inferred from the signature: the first injection represents the *async* effect and the second the *await* effect. Ideally we would like to reference the effects by their name and thus provide more informative effect specifications.

In *Haffel* we can express *async* and *await* in a direct way that solves the aforementioned problems. By utilising effect labels and equi-recursive effect rows, we can redefine the *coop* effect as follows:

$$coop := \mu\theta. (async : \forall\alpha. (\ () \stackrel{\theta}{\circ} \alpha) \Rightarrow Promise !\alpha) \cdot (await : \forall\alpha. Promise !\alpha \Rightarrow !\alpha) \cdot \langle \rangle$$

Instead of using a single effect signature, we utilise effect rows that properly distinguish between the two effects. Additionally, recursive effect rows which generalise recursive signatures allow us to recursively define the *coop* effect and thus allow the argument type of *async* to perform *coop* effects.

4.2 Syntax and Semantics

Syntax Figure 15 shows the syntax of expressions in *Haffel*. The highlighted expressions indicate the changes or additions to the language.

To accommodate multi-shot continuations, we adopt the approach of *Hazel* where effect related constructs are generalised with a mode m that syntactically distinguishes between one- and multi-shot effects. To motivate why this is needed, recall that in *Haffel_{os}* a continuation is represented as the value `cont ℓ N` which stores a location ℓ that tracks whether the continuation has been called. This location is only present to model by means of a stuck expression the run-time error that occurs when one-shot continuations are called more than once. We would like that safe expressions in *Haffel* also provide this one-shot guarantee and so we need to operationally distinguish between evaluating a one- or multi-shot effect. For this reason, we parameterise the `do` construct and its associated active effect construct `eff` with a mode m . This syntactic distinction allows us to have different reduction rules for each kind of effect. Just like with *Hazel*, we introduce a new continuation value `kont N` to represent multi-shot continuations that allows arbitrary number of calls.

To support named effects we further adapt the syntax of `do` and `eff` to take an additional effect name op . The handler construct also takes an effect name to indicate which effect should be handled. To address the issue of name clashes that can arise in the presence of effect polymorphism the `lift` and `unlift` constructs are introduced. Intuitively, every active effect will have a level i associated with it that represents the number of enclosing handlers it will skip before it can be handled. The `lift op e`

expression will increase by one the level of effects with name op produced from evaluation of e . The `unlift op e` is the inverse operation where the level of op effects is reduced by one. It should be noted that `unlift` is a run-time expression, introduced in the evaluation of a handler, and so a user would not use `unlift` directly. The `lift` construct is directly inspired by the calculus of Biernacki et al. [Bie+17], with the difference that we introduce effect levels in the language to implement lifting instead of their approach of having constraints on the captured evaluation contexts. The `unlift` operation is not present in their calculus. We also expand the neutral contexts in the language to encompass the `lift` and `unlift` constructs.

Lastly, we introduce the `let-in` construct explicitly in the language instead of being derived as in $Haffel_{os}$. As will be described in section 4.4, it can be given a useful typing rule that can not be derived solely from the application typing rule.

$m, m' ::= O \mid M$	(Mode)
$e ::= x \mid \mathbf{rec} f x. e \mid e_1 e_2 \mid \mathbf{let} x = e_1 \mathbf{in} e_2 \mid \Lambda. e \mid e \langle \rangle$	(λ -calculus)
$\mid \mathbf{true} \mid \mathbf{false} \mid (i \in \mathbb{Z})$	(Literals)
$\mid \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3 \mid e_1 \odot e_2$	(Operations)
$\mid () \mid (e_1, e_2) \mid \mathbf{let} (x_1, x_2) = e_1 \mathbf{in} e_2$	(Products)
$\mid \mathbf{injL} e \mid \mathbf{injR} e \mid (\mathbf{match} e_1 \mathbf{with} e_2 \mid e_3)$	(Sums)
$\mid \mathbf{fold} e \mid \mathbf{unfold} e$	(Iso-recursive Types)
$\mid (\ell \in Loc) \mid \mathbf{ref} e \mid !e \mid e_1 := e_2$	(Mutable State)
$\mid \mathbf{do}_m op e \mid \mathbf{eff}_m (op, i) v N \mid \mathbf{lift} op e \mid \mathbf{unlift} op e$	(Effects)
$\mid \mathbf{cont} \ell N \mid \mathbf{kont} N$	(Continuations)
$\mid (\mathbf{deep-try} e_1 \mathbf{with} op \mathbf{by} e_2 \mid e_3)$	(Handlers)
$N ::= \mathbf{lift} op N \mid \mathbf{unlift} op N \mid \dots$	

Figure 15: Changes to the syntax of expressions in $Haffel$.

The syntax of types, effect signatures and effect rows is shown in fig. 16. Effect signatures are generalised to take a mode m that tracks whether an effect is one- or multi-shot. In essence, the continuation associated with a one-shot effect can be resumed at most once, whereas with multi-shot effects it can be resumed multiple times. To make signatures less verbose we introduce the following notations:

$$\begin{aligned} \tau \Rightarrow^O \kappa &::= \tau \Rightarrow^O \kappa \\ \tau \Rightarrow^M \kappa &::= \tau \Rightarrow^M \kappa \end{aligned}$$

To support named effects we introduce effect rows that group multiple named effect signatures together. Effect rows are list-like structures as witnessed by the Nil row ($\langle \rangle$) which states the absence of effects, and the Cons row $((op : \sigma) \cdot \rho)$ which inserts a named effect op with signature σ to the row ρ . The order of named effects in effect rows matters only up to duplicate effect names. This means that the following effect rows are equivalent when $op \neq op'$:

$$(op : \sigma) \cdot (op' : \sigma') \cdot \rho \equiv (op' : \sigma') \cdot (op : \sigma) \cdot \rho$$

On the other hand, two consecutive named effects that use the same name op can not be swapped:

$$(op : \sigma) \cdot (op : \sigma') \cdot \rho \not\equiv (op : \sigma') \cdot (op : \sigma) \cdot \rho$$

$\tau, \kappa, \iota ::= \alpha$	(Type Variables)
<code>void</code> <code>()</code> <code>bool</code> <code>int</code>	(Base Types)
$\tau \xrightarrow{\rho} \kappa$ $\tau \xrightarrow{\rho} \kappa$	(Function Types)
$\tau * \kappa$ $\tau + \kappa$	(Product and Sum Types)
<code>ref</code> τ <code>!</code> τ	(Reference and Unrestricted Type)
$\forall \alpha. \tau$ $\forall \theta. \tau$	(Universal Types)
$\mu \alpha. \tau$	(Recursive Types)
$\sigma, \hat{\sigma} ::= \forall \alpha. \tau \Rightarrow^m \kappa$	(Effectful Signature)
$\rho, \hat{\rho} ::= \theta$	(Row Variables)
$\langle \rangle$	(Nil Row)
$(op : \sigma) \cdot \rho$	(Cons Row)
$\mu \theta. \rho^*$	(Recursive Row)
<code>i</code> ρ	(Restricted Row)

*There is a syntactic restriction that the recursive variable θ is only used in effect signatures and not in the spine of the effect row

Figure 16: Syntax of signatures in *Haffel*.

Thus effect rows can have duplicate named effects and the order of these duplicate effect names matters. This identification on effect rows is captured by the sub-typing relation explained in section 4.3. Since effect rows have a list-like structure, we will use $\langle op : \sigma \rangle$ to refer to a singleton effect row $(op : \sigma) \cdot \langle \rangle$ and accordingly $\langle (op : \sigma), (op' : \sigma') \rangle$ and $\langle (op : \sigma), (op' : \sigma'), (\hat{op} : \hat{\sigma}) \rangle$ for effect rows with two and three signatures.

The recursive binder in the effect signatures of *Haffel*_{os} is hoisted to the level of effect rows, as seen by the equi-recursive row $(\mu \theta. \rho)$ which allows the effect row ρ to reference itself via the θ variable. To ensure effect rows are finite, there is a syntactical restriction on a recursive effect row which states that the recursive binder θ should only appear in the effect signatures present in the effect row ρ . Effect rows that utilise the recursive θ binder in the spine of the effect row are not considered well-formed:

$$\begin{aligned}
 & (\mu \theta. \theta), (\mu \theta. (op : \sigma') \cdot \theta), (\mu \theta. i\theta) && \text{(ill-formed rows)} \\
 & (\mu \theta. \langle \rangle), (\mu \theta. (op : (\langle \rangle \xrightarrow{\theta} \langle \rangle) \Rightarrow \langle \rangle) \cdot \langle \rangle) && \text{(well-formed rows)}
 \end{aligned}$$

Lastly, *Haffel* introduces a new restricted effect row that allows effect polymorphism over *one-shot* effect rows. The row `i` ρ states that all signatures in ρ must be handled in a one-shot way. It is in a sense dual to the unrestricted type `!` τ , whereby instead of representing copyable types that can be used multiple times, it represents effect rows that must be handled in a one-shot way. We give further explanations and use cases in section 4.3.

Effect Levels Instantiating an effect polymorphic function may lead to name clashes with other effects in the effect row. This is undesirable as it can lead to an unsound system where effects can have two distinct signatures. Take for instance the following

toYield function that calls its function argument f and passes its result to the *yield* effect:

$$\text{let } toYield = \Lambda. \lambda f. \text{do}_M \text{ yield } (f ())$$

We would like to give it an effect polymorphic type such as the following type. For illustration purposes, we fix the result type of f to be `int` instead of quantifying over the result type.

$$toYield : \forall \theta. (() \xrightarrow{\theta} \text{int}) \xrightarrow{(yield:\text{int} \Rightarrow ()) \cdot \theta} ()$$

A consequence of this is that the effect variable θ might itself be instantiated with the *yield* effect which would lead to a name clash:

$$toYield \langle \rangle : (() \xrightarrow{(yield:() \Rightarrow ())} \text{int}) \xrightarrow{\langle (yield:\text{int} \Rightarrow ()), (yield:() \Rightarrow ()) \rangle} ()$$

This instantiation says that the function can emit the *yield* effect with two different effect signatures. If we attempt to install a handler to capture these effects we would be unable to give it a correct typing since the two *yield* effects have incompatible effect signatures: the type of the effect value is either `int` or `()`. Some effect systems handle this problem by forbidding this instantiation from happening [HL16; Tan+24]. We instead take the approach of Leijen [Lei17] and Biernacki et al. [Bie+17] that allow duplicate effect names (an effect can have multiple signatures) but the order of these effects in the effect row matters. In our system, the effect row of the example distinguishes between the two *yield* effects since the order of duplicate effects matters:

$$\langle (yield : \text{int} \Rightarrow ()), (yield : () \Rightarrow ()) \rangle \not\equiv \langle (yield : () \Rightarrow ()), (yield : \text{int} \Rightarrow ()) \rangle$$

To link the position of an effect in an effect row with its run-time representation as an active effect expression, effect levels are introduced. The level of an effect is stored via the i field in the active effect expression ($\text{eff}_m (op, i) v N$) and it is a natural number that represents the number of duplicate effect names that precede it in an effect row. For instance, the level of $(yield : \text{int} \Rightarrow ())$ in $\langle (yield : \text{int} \Rightarrow ()), (yield : () \Rightarrow ()) \rangle$ is 0 because it appears in the head position and so there are no effect names that precede it. On the other hand, the level of the effect $(yield : () \Rightarrow ())$ is 1, because there exists one duplicate effect that precedes it in the effect row.

Effect levels avoid clashes of effects with different signatures by distinguishing between duplicate effect names. By requiring that handlers only service level 0 effects, we can install a handler for the *yield* effect which would only intercept effect requests with signature $(yield : \text{int} \Rightarrow ())$.

For this approach to work we need to directly manipulate the levels of effects. In the above example, the expression $f ()$ has effect row $\langle yield : () \Rightarrow () \rangle$ (the instantiation of θ) which says that the effect $(yield : () \Rightarrow ())$ emitted is of level 0. However, in the body of *toYield* we require it to produce a level 1 *yield* effect. We thus need to wrap the application with a call to `(lift yield)` in order to raise its level by one:

$$\text{let } toYield = \Lambda. \lambda f. \text{do}_M \text{ yield } (\text{lift } yield (f ()))$$

The *toYield* function now adheres to its effect row by emitting effects $(yield : \text{int} \Rightarrow ())$ at level 0 through the `doM` call and effects $(yield : () \Rightarrow ())$ at level 1 through `(lift yield (f ()))`.

Semantics To enable multi-shot continuations, the small-step semantics of *Haffel_{os}* are extended in a direct way by propagating the mode m found in the `do` construct to the active effect expression `eff`. Additionally, a reduction rule for the multi-shot continuation `kont N` is introduced that simply fills the hole of its neutral context N

with the argument v . Note that in contrast to one-shot continuation values $\text{cont } \ell N$, the $\text{kont } N$ value can be called more than once.

The main added complexity in the operational semantics comes from the handling of named effects. We add reduction rules for the lift and unlift constructs which increase and decrease the level of an active effect expression by one correspondingly. We additionally place them in the neutral context of the active effect expression to make them persistent. Since the lifting operations are also part of the neutral contexts, we need to make sure that the reduction rule for $N[\text{eff}_m(op, i) v N']$ that shallows the surrounding context does not apply to them as that would overlap with the other lifting reduction rules. This is achieved by the insertion of two new side conditions.

The effect level is used in the handler reduction rules, where an active effect expression is only handled when its level is 0. To handle a one-shot active expression ($\text{eff}_O(op, 0) v N$), the effect name op must match with that of the handler and the level must be 0. The reduction in this case is similar to the handler reduction rule in Haffel_{os} where the effect branch h is evaluated by wrapping the application of the continuation with the current handler. Handling of a multi-shot active expression ($\text{eff}_M(op, 0) v N$) is almost identical to the one-shot case with the only difference that a $\text{kont } N$ value is used.

To handle an active effect expression ($\text{eff}_m(op, i) v N$) that has a different op name than that of the handler or when the level i is not 0, we propagate the effect upwards for another handler to service. Since the active effect expression has already reached a handler we must also reduce its level by one which is achieved using the unlift construct. Thus we firstly re-emit the effect ($\text{eff}_m(op, i) v \bullet$) in the context of an unlift , and the result of this effect call is substituted back to the current neutral context N . To make the current handler persist, the resumption of the neutral context N is performed in the context of the same handler.

$$\begin{array}{l}
\text{let } x = v \text{ in } e / \sigma \rightsquigarrow^h e[v/x] / \sigma \\
\text{do}_m op v / \sigma \rightsquigarrow^h \text{eff}_m(op, 0) v \bullet / \sigma \\
N[\text{eff}_m(op, i) v N'] / \sigma \rightsquigarrow^h \text{eff}_m(op, i) v (N[N']) / \sigma \\
\text{lift } op v / \sigma \rightsquigarrow^h v / \sigma \\
\text{unlift } op v / \sigma \rightsquigarrow^h v / \sigma \\
\text{lift } op (\text{eff}_m(op, i) v N) / \sigma \rightsquigarrow^h \text{eff}_m(op, i+1) v (\text{lift } op N) / \sigma \\
\text{unlift } op (\text{eff}_m(op, i+1) v N) / \sigma \rightsquigarrow^h \text{eff}_m(op, i) v (\text{unlift } op N) / \sigma \\
\text{deep-try } (\text{eff}_O(op, 0) v N) \text{ with } op \text{ by } / \sigma \rightsquigarrow^h h[v/x][k'/k] / \sigma[\ell \mapsto \text{false}] \\
\quad \lambda x k. h \quad \text{if } \ell \notin \text{Dom } \sigma \\
\quad | \lambda x. r \quad \text{where} \\
\quad \quad k' := \lambda x. \text{deep-try } (\text{cont } \ell N) x \text{ with } op \text{ by} \\
\quad \quad \quad \lambda x k. h \\
\quad \quad \quad | \lambda x. r \\
\text{deep-try } (\text{eff}_M(op, 0) v N) \text{ with } op \text{ by } / \sigma \rightsquigarrow^h h[v/x][k'/k] / \sigma \\
\quad \lambda x k. h \quad \text{where} \\
\quad | \lambda x. r \quad k' := \lambda x. \text{deep-try } (\text{kont } N) x \text{ with } op \text{ by} \\
\quad \quad \lambda x k. h \\
\quad \quad | \lambda x. r \\
\text{deep-try } (\text{eff}_m(op', i) v N) \text{ with } op \text{ by } / \sigma \rightsquigarrow^h k' (\text{unlift } op (\text{eff}_m(op', i) v \bullet)) / \sigma \\
\quad \lambda x k. h \quad \text{if } op \neq op' \text{ or } i \neq 0 \\
\quad | \lambda x. r \quad \text{where} \\
\quad \quad k' := \lambda x. \text{deep-try } N[x] \text{ with } op' \text{ by} \\
\quad \quad \quad \lambda x k. h \\
\quad \quad \quad | \lambda x. r
\end{array}$$

Figure 17: Changes to the head reduction relation in Haffel

4.3 Relation on Signatures and Rows

The relation on signatures is adapted to relate one- and multi-shot signatures and a new relation on effect rows is introduced (fig. 18). There are five main requirements that the new sub-typing relation addresses:

- Allow a multi-shot effect to be used in contexts where a one-shot is expected (EFF-MODE).
- Allow a non-effectful expression to be given any effect row (NIL).
- Specify that the order of named signatures is irrelevant up-to duplicate effect labels (SWAP).
- Fold and Unfold a recursive effect row (RREC-FOLD and RREC-UNFOLD).
- Translate between a one-shot effect row and a restricted row (RES-INTRO and RES-ELIM).

The rule EFF-MODE allows a multi-shot signature to be used in contexts where a one-shot signature is expected. Multi-shot signatures can be serviced by a handler that uses the associated continuation more than once. It is sound to transform it to a one-shot signature that allows the handler to use the continuation at most once.

The empty signature (\perp) of $Haffel_{os}$ is hoisted to the level of rows as the Nil effect row ($\langle \rangle$). Accordingly, the rule NIL replaces the corresponding rule EMPTY of $Haffel_{os}$ which allows a non-effectful expression to be given any effect.

For Cons rows, the relation is defined component wise (CONS) and the order irrelevance on effect rows which allows two distinct named signatures to be swapped is captured by the rule SWAP.

The rules RREC-FOLD and RREC-UNFOLD allow folding and unfolding of recursive effect rows. We note that folding is only defined for well-formed rows which are rows where the recursive variable is only used in signatures and not in the spine of the row.

Lastly, the introduction rule RES-INTRO and elimination rule RES-ELIM allow translation from and to a restricted row. Any effect row can be sub-typed to become a restricted effect row, but to eliminate the restricted row constructor the row must be one-shot. To state this, the one-shot row constraint ($\text{OneShot } \rho$) is used which requires that all the signatures in row ρ are one-shot, as shown by the derivation rules in fig. 19.

Importantly, a restricted row satisfies the one-shot row constraint $\text{OneShot}(i\rho)$ and for this reason it can be used to define one-shot effect polymorphic functions. For instance, consider the following higher-order identity function that calls the function f before returning x :

$$\Lambda. \Lambda. \lambda f x. f (); x$$

If the function f performs multi-shot effects and the instantiated type of x is non-substructural, the continuation will capture the argument x and the handler can use x multiple times breaking the *at most once* guarantee. The restricted effect row $i\theta$ can be used in such cases to ensure that f can only produce one-shot effects and thus any handler that services the effect will use the continuation at most once.

$$\forall \theta. \forall \alpha. ((\) \xrightarrow{i\theta} ()) \rightarrow \alpha \xrightarrow{i\theta} \alpha$$

An alternative typing that can allow multi-shot effects will instead require that argument x can be used multiple times. Unrestricted types can be used to specify this:

$$\forall \theta. \forall \alpha. ((\) \xrightarrow{\theta} ()) \rightarrow !\alpha \xrightarrow{\theta} !\alpha$$

$\boxed{\Sigma \vdash \sigma < \hat{\sigma}}$	
$\frac{\text{EFF} \quad \Sigma \vdash \tau <: \tau' \quad \Sigma \vdash \kappa' <: \kappa}{\Sigma \vdash \forall \alpha. \tau \Rightarrow^m \kappa <: \forall \alpha. \tau' \Rightarrow^m \kappa'}$	$\text{EFF-MODE} \quad \Sigma \vdash \forall \alpha. \tau \Rightarrow \kappa <: \forall \alpha. \tau \Rightarrow \kappa$
$\boxed{\Sigma \vdash \rho <: \hat{\rho}}$	
$\text{REFL} \quad \Sigma \vdash \rho <: \rho$	$\frac{\text{TRANS} \quad \Sigma \vdash \rho <: \hat{\rho} \quad \Sigma \vdash \hat{\rho} <: \rho'}{\Sigma \vdash \rho <: \rho'}$
$\text{NIL} \quad \Sigma \vdash \langle \rangle <: \rho$	$\frac{\text{CONS} \quad \Sigma \vdash \sigma <: \hat{\sigma} \quad \Sigma \vdash \rho <: \hat{\rho}}{\Sigma \vdash (op : \sigma) \cdot \rho <: (op : \hat{\sigma}) \cdot \hat{\rho}}$
$\text{SWAP} \quad \frac{op \neq op'}{\Sigma \vdash (op : \sigma) \cdot (op' : \sigma') \cdot \rho <: (op' : \sigma') \cdot (op : \sigma) \cdot \rho}$	
$\text{RREC-UNFOLD} \quad \Sigma \vdash \mu\theta. \rho <: \rho [\mu\theta. \rho / \theta]$	$\text{RREC-FOLD} \quad \Sigma \vdash \rho [\mu\theta. \rho / \theta] <: \mu\theta. \rho$
$\text{RES-INTRO} \quad \Sigma \vdash \rho <: i\rho$	$\frac{\text{RES-ELIM} \quad \text{OneShot } \rho}{\Sigma \vdash i\rho <: \rho}$
$\frac{\text{RES-COMP} \quad \Sigma \vdash \rho <: \hat{\rho}}{\Sigma \vdash i\rho <: i\hat{\rho}}$	$\text{RES-CONS} \quad \Sigma \vdash i((op : \tau \Rightarrow \kappa) \cdot \rho) <: (op : \tau \Rightarrow \kappa) \cdot i\rho$

Figure 18: Subsumption relation of signatures and rows in *Haffel*

We note that since none of the two aforementioned types are an instance of the other, *Haffel* does not feature principal types. Principal types are further discussed in section 7.

The introduction and elimination rules of restricted effect rows are dual to that of unrestricted types (URES-INTRO and URES-ELIM). Restricted effect rows can only be handled in a one-shot way whereas unrestricted types can be used multiple times. Additionally, the rule RES-COMP extends the sub-typing relation component-wise to restricted rows and rule RES-CONS distributes the restricted constructor over the tail of a Cons row provided the head is a one-shot signature.

$\frac{\text{OS-NIL}}{\text{OneShot } \langle \rangle}$	$\frac{\text{OS-CONS} \quad \text{OneShot } \rho}{\text{OneShot } ((op : \forall \alpha. \tau \Rightarrow \kappa) \cdot \rho)}$	$\frac{\text{OS-RES}}{\text{OneShot } (i\rho)}$	$\frac{\text{OS-REC} \quad \text{OneShot } \rho}{\text{OneShot } (\mu\theta. \rho)}$
---	---	---	--

Figure 19: Derivation rules for one-shot row constraint.

4.4 Typing Rules

Figure 20 shows a selected list of the typing rules of *Haffel*. The main points that differ from the corresponding typing rules of *Haffel* are highlighted. We note that the remaining typing rules not shown are simple generalisations of the corresponding ones of *Haffel*. Instead of using effect signatures, effect rows are used with no additional one-shot or copyable constraints.

Multi-Shot Effects There are two main features in the typing rules that enable effects to be handled in a multi-shot way. In conjunction they ensure that continuations only capture non-substructural resources:

- The final environment is copyable in rule DO-MS.
- There are multi-shot versions of composite typing rules such as APP-MS, PAIR-MS and WRITE-MS.

The rule DO-MS differs from its one-shot counterpart in that the final environment Γ_2 needs to be copyable. This is the first step towards treating the continuation created as non-substructural. The final environment states the resources that can be used after evaluation of $(\text{do}_M \text{ op } e)$ which are exactly the resources that may be captured by the continuation. To see why this is the case, observe the following partial derivation for the expression $(\text{do}_M \text{ op } (); k ())$ that performs the *op* effect followed by a function call to the substructurally-treated function *k*.

$$\frac{\frac{k : () \multimap () \vdash () : \langle \rangle : () \dashv k : () \multimap ()}{\vdots} \quad \frac{\text{Copy } (k : () \multimap ())}{k : () \multimap () \vdash \text{do}_M \text{ op } () : \langle op : () \Rightarrow () \rangle : () \dashv k : () \multimap ()} \quad \vdots}{k : () \multimap () \vdash \text{do}_M \text{ op } (); k () : \langle op : () \Rightarrow () \rangle : () \dashv}$$

The typing rules will reject this program because the copyable constraint ($\text{Copy } (k : () \multimap ())$) can not be satisfied. Indeed if we try to install a handler for this expression, the neutral context associated with the continuation would be $(\bullet; k ())$. To ensure that *k* is called at most once, the continuation must also be one-shot and thus it would be unsound to give this effect a multi-shot signature. Similarly, the FRAME-MS rule reflects this requirement by ensuring that the final environment is kept non-substructural by appending only copyable types to it ($\text{Copy } \Gamma'$).

In addition, to ensure that the incremental built-up of multi-shot continuations maintains their non-substructural property, separate typing rules are needed for the one- and multi-shot case. In the case of pair typing, the PAIR-MS requires the second component of the pair to be non-substructural ($\text{Copy } \kappa$). To understand why this is required, assume that e_1 produces a multi-shot effect call and thus evaluation of e_1 eventually results in an active effect expression with some effect value v and neutral context N . Since we are

a using right-to-left evaluation order, evaluation of e_2 must have finished and reached a value v_2 . The active effect expression would then swallow its surrounding pair context:

$$(\mathbf{eff}_M(op, i) v N, v_2) / \sigma \rightsquigarrow \mathbf{eff}_M(op, i) v (N, v_2) / \sigma$$

The value v_2 will be captured in the continuation and so it must be of a copyable type ($\mathbf{Copy} \kappa$) to ensure the continuation can be called multiple times.

Similarly, application requires its argument to be of a non-substructural type but also its final environment Γ_3 . The extra requirement on the environment is due to effect sub-typing, specifically the rule \mathbf{NIL} . A judgement of the form $\Gamma_2 \vdash e : \rho : \tau \dashv \Gamma_3$ does not guarantee that Γ_3 is copyable even if ρ includes a multi-shot effect. When e is a non-effectful expression, we could type it with the \mathbf{Nil} effect row, $\Gamma_2 \vdash e : \langle \rangle : \tau \dashv \Gamma_3$, and thus Γ_3 may capture substructural resources since it satisfies the one-shot constraint ($\mathbf{OneShot} \langle \rangle$). We could then apply the subsumption rule \mathbf{SUB}' to give it the ρ effect row. This situation arises in the typing of abstractions, since they are not evaluated can thus be typed with any final environment Γ_3 . During application, the body of the abstraction may call a multi-shot effect and as a result resources in the final environment Γ_3 may be captured in the continuation. Take for instance the following expression $((\lambda (). \mathbf{do}_M op () (); k ()))$. The substructurally treated function k can be called more than once if the effect op is handled in a multi-shot way. But as the following derivation for the abstraction shows, we may type the abstraction with the non-copyable environment $(k : () \multimap ())$:

$$\frac{\vdots}{\frac{\frac{\vdash \lambda (). \mathbf{do}_M op () : \langle \rangle : () \quad \langle op : () \Rightarrow () \rangle_{\circ} () \dashv}{\text{Frame-OS}}}{k : () \multimap () \vdash \lambda (). \mathbf{do}_M op () : \langle \rangle : () \quad \langle op : () \Rightarrow () \rangle_{\circ} () \dashv k : () \multimap ()}{\text{Sub}'}}{k : () \multimap () \vdash \lambda (). \mathbf{do}_M op () : \langle op : () \Rightarrow () \rangle : () \quad \langle op : () \Rightarrow () \rangle_{\circ} () \dashv k : () \multimap ()}$$

The $\mathbf{Copy}(\Gamma_3)$ constraint in the application rule is required to ensure continuations produced by the evaluation of the abstraction do not capture substructural resources. In this example, the program will be rejected due to the unsatisfiable constraint ($\mathbf{Copy}(k : () \multimap ())$).

Writing to a reference ($\mathbf{WRITE-MS}$) has corresponding one- and multi-shot versions just like pair and application typing for the same reason described.

The one-shot versions of the rules, $\mathbf{PAIR-OS}$, $\mathbf{APP-OS}$, $\mathbf{WRITE-OS}$ and $\mathbf{FRAME-OS}$, do not require copyable types and environments but instead that the effect row is one-shot ($\mathbf{OneShot} \rho$). This ensures that the continuations from the effect calls can only be handled in a one-shot way.

Interestingly, the \mathbf{LET} does not require any one-shot row or copyable constraints. The typing judgments for expressions e_1 and e_2 will implicitly require that their final environments are copyable if the expressions produce a multi-shot effect.

The deep handler also has two separate typing rules depending on the signature of the effect it handles. Their only difference is in the effect branch which states how the continuation can be used. When the effect handled is one-shot, it must be used at most once whereas in the multi-shot case it becomes an unrestricted function.

Named Effects To support named effects the typing rules are generalised to take an effect row instead of an effect signature. In non-effect related typing rules this is a simple syntactical replacement of signature meta-variable σ with row meta-variable ρ without any additional restrictions. The dual nature of typing rules for pairs, application and reference writing, together with the additional one-shot row constraint restriction is related to multi-shot effects and not directly to supporting named effects.

The DO-OS and DO-MS rules require a Cons effect row $((op : \sigma) \cdot \rho)$ which gives the signature of the effect to be performed. Due to effect sub-typing, we can perform any effect in the effect row as long as there are no duplicate effect names earlier in the effect row: we can only perform the first duplicate effect. The reason for this is that calling an effect produces a level 0 active effect expression and as a result only the first duplicate effect can be called, the rest have higher effect levels. To call a effect with a higher level, the **lift** operation can be used.

The LIFT rule shows the typing of the **lift** construct. Given an effectful expression e , $(\mathbf{lift} \ op \ e)$ will lift all the effects with label op to one higher level. We can type it with effect row $((op : \sigma) \cdot \rho)$ because $(op : \sigma)$ has level 0 and **lift** $op \ e$ can not produce a level 0 op effect. Through this rule, we can call higher-level effects and most importantly add effects to effect polymorphic effect rows. For instance, the *toYield* function from section 4.2 can be typed via the following derivation:

$$\begin{array}{c}
\vdots \\
\hline
f : () \xrightarrow{\theta} \mathbf{int} \vdash f () : \theta : \mathbf{int} \dashv \quad \text{App-MS} \\
\hline
f : () \xrightarrow{\theta} \mathbf{int} \vdash \mathbf{lift} \ yield (f ()) : (yield : \mathbf{int} \Rightarrow ()) \cdot \theta : \mathbf{int} \dashv \quad \text{Lift} \\
\hline
f : () \xrightarrow{\theta} \mathbf{int} \vdash \mathbf{do}_M \ yield (\mathbf{lift} \ yield (f ())) : (yield : \mathbf{int} \Rightarrow ()) \cdot \theta : () \dashv \quad \text{Do-MS} \\
\hline
f : () \xrightarrow{\theta} \mathbf{int} \vdash \mathbf{do}_M \ yield (\mathbf{lift} \ yield (f ())) : \langle \rangle : (()) \xrightarrow{\theta} \mathbf{int} \dashv \quad \text{URes-Fun}' \\
\hline
\vdash \lambda f. \mathbf{do}_M \ yield (\mathbf{lift} \ yield (f ())) : \langle \rangle : (()) \xrightarrow{\theta} \mathbf{int} \dashv \quad \text{Forall-Row-Intro} \\
\hline
\vdash \Lambda. \lambda f. \mathbf{do}_M \ yield (\mathbf{lift} \ yield (f ())) : \langle \rangle : \forall \theta. (()) \xrightarrow{\theta} \mathbf{int} \dashv
\end{array}$$

The function call $(f ())$ has effect row θ , and through the **lift** operation we give it the effect row $((yield : \mathbf{int} \Rightarrow ()) \cdot \theta)$ which is compatible with the effect row of the outer **do** call. Note that effect sub-typing would not allow us to sub-type θ to $(yield : \mathbf{int} \Rightarrow ()) \cdot \theta$ due to the row variable θ . Indeed it would be unsound to do so, since there may be clashes of level 0 *yield* effects with different signatures.

The **unlift** operation that lowers an effect by one level does not have a corresponding typing rule. The effect rows are not expressive enough to describe this effect level lowering behaviour in a sound way that leads to no clashes. Regardless, the **unlift** is still used in a sound way during evaluation of the deep handler. This is where the semantic approach to typing shines, by allowing us to prove type safety for the language even in the absence of a valid typing rule for **unlift**. In alternative syntactical proofs of progress and preservation, this lack of typing would lead to a problem.

Handling of a named effect happens in a similar manner to performing, by requiring a Cons effect row $((op : \sigma) \cdot \rho')$ which gives the signature of the effect to be handled. Any level 0 effect can be handled by applying the effect sub-typing rule SWAP. The resulting effect row of the handler must be compatible with the tail of the effect row, namely ρ' . Effects in ρ other than the one handled will be propagated upwards and so the handler's effect row must reflect this. Instead of typing it with ρ' directly, we add some flexibility to the typing of the branches by utilising the sub-typing relation $\rho' < \hat{\rho}$. This allows more effects other than ones mentioned in ρ' to be performed.

In addition, higher-level op effects in $(op : \sigma) \cdot \rho'$ will be lowered by one level via the introduction of the **unlift** operation in the evaluation of the handler. This is reflected in the typing rule by taking the tail of the effect row ρ' which has op effects lowered by one level compared to $(op : \sigma) \cdot \rho'$.

$$\boxed{\Gamma_1 \vdash e : \sigma : \tau \dashv \Gamma_2}$$

$$\text{PAIR-OS} \quad \text{OneShot } \rho \quad \frac{\Gamma_2 \vdash e_1 : \rho : \tau \dashv \Gamma_3 \quad \Gamma_1 \vdash e_2 : \rho : \kappa \dashv \Gamma_2}{\Gamma_1 \vdash (e_1, e_2) : \rho : \tau * \kappa \dashv \Gamma_3}$$

$$\text{PAIR-MS} \quad \text{Copy } \kappa \quad \frac{\Gamma_2 \vdash e_1 : \rho : \tau \dashv \Gamma_3 \quad \Gamma_1 \vdash e_2 : \rho : \kappa \dashv \Gamma_2}{\Gamma_1 \vdash (e_1, e_2) : \rho : \tau * \kappa \dashv \Gamma_3}$$

$$\text{LET} \quad \frac{\Gamma_1 \vdash e_1 : \rho : \tau \dashv \Gamma_2 \quad x : \tau, \Gamma_2 \vdash e_2 : \rho : \kappa \dashv \Gamma_3}{\Gamma_1 \vdash \text{let } x = e_1 \text{ in } e_2 : \rho : \kappa \dashv \Gamma_3}$$

$$\text{APP-OS} \quad \text{OneShot } \rho \quad \frac{\Gamma_2 \vdash e_1 : \rho : \tau \stackrel{\rho}{\dashv} \kappa \dashv \Gamma_3 \quad \Gamma_1 \vdash e_2 : \rho : \tau \dashv \Gamma_2}{\Gamma_1 \vdash e_1 e_2 : \rho : \kappa \dashv \Gamma_3}$$

$$\text{APP-MS} \quad \text{Copy } \tau \quad \text{Copy } \Gamma_3 \quad \frac{\Gamma_2 \vdash e_1 : \rho : \tau \stackrel{\rho}{\dashv} \kappa \dashv \Gamma_3 \quad \Gamma_1 \vdash e_2 : \rho : \tau \dashv \Gamma_2}{\Gamma_1 \vdash e_1 e_2 : \rho : \kappa \dashv \Gamma_3}$$

$$\text{WRITE-OS} \quad \text{OneShot } \rho \quad \frac{\Gamma_1 \vdash e_2 : \rho : \tau \dashv \Gamma_2 \quad \Gamma_2 \vdash e_1 : \rho : \text{ref } \tau \dashv \Gamma_3}{\Gamma_1 \vdash e_1 := e_2 : \rho : \tau \dashv \Gamma_3}$$

$$\text{WRITE-MS} \quad \text{Copy } \tau \quad \frac{\Gamma_1 \vdash e_2 : \rho : \tau \dashv \Gamma_2 \quad \Gamma_2 \vdash e_1 : \rho : \text{ref } \tau \dashv \Gamma_3}{\Gamma_1 \vdash e_1 := e_2 : \rho : \tau \dashv \Gamma_3}$$

$$\text{DO-OS} \quad \frac{\rho = (op : \forall \alpha. \iota \Rightarrow \kappa) \cdot \hat{\rho} \quad \Gamma_1 \vdash e : \rho : \iota[\tau/\alpha] \dashv \Gamma_2}{\Gamma_1 \vdash \text{do}_O op e : \rho : \kappa[\tau/\alpha] \dashv \Gamma_2}$$

$$\text{DO-MS} \quad \frac{\rho = (op : \forall \alpha. \iota \Rightarrow \kappa) \cdot \hat{\rho} \quad \text{Copy } \Gamma_2 \quad \Gamma_1 \vdash e : \rho : \iota[\tau/\alpha] \dashv \Gamma_2}{\Gamma_1 \vdash \text{do}_M op e : \rho : \kappa[\tau/\alpha] \dashv \Gamma_2}$$

$$\text{LIFT} \quad \frac{\Gamma_1 \vdash e : \rho : \iota \dashv \Gamma_2}{\Gamma_1 \vdash \text{lift } op e : (op : \sigma) \cdot \rho : \tau \dashv \Gamma_2}$$

$$\text{DEEP-HANDLER-OS} \quad \frac{\rho = (op : \forall \alpha. \iota \Rightarrow \kappa) \cdot \rho' \quad \rho' < \hat{\rho} \quad \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2 \quad x, k \notin \text{Dom } \Gamma \quad \text{Copy } \Gamma \quad x : \tau, \Gamma_2; \Gamma \vdash r : \hat{\rho} : \tau' \dashv \Gamma \quad x : \iota, k : \kappa \stackrel{\hat{\rho}}{\dashv} \tau', \Gamma \vdash h : \hat{\rho} : \tau' \dashv \Gamma}{\Gamma_1; \Gamma \vdash \text{deep-try } e \text{ with } op \text{ by } \lambda x k. h \mid \lambda x. r : \hat{\rho} : \tau' \dashv \Gamma}$$

$$\text{DEEP-HANDLER-MS} \quad \frac{\rho = (op : \forall \alpha. \iota \Rightarrow \kappa) \cdot \rho' \quad \rho' < \hat{\rho} \quad \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2 \quad x, k \notin \text{Dom } \Gamma \quad \text{Copy } \Gamma \quad x : \tau, \Gamma_2; \Gamma \vdash r : \hat{\rho} : \tau' \dashv \Gamma \quad x : \iota, k : \kappa \stackrel{\hat{\rho}}{\dashv} \tau', \Gamma \vdash h : \hat{\rho} : \tau' \dashv \Gamma}{\Gamma_1; \Gamma \vdash \text{deep-try } e \text{ with } op \text{ by } \lambda x k. h \mid \lambda x. r : \hat{\rho} : \tau' \dashv \Gamma}$$

$$\text{SUB}' \quad \frac{\Gamma'_1 \vdash e : \rho : \kappa \dashv \Gamma'_2 \quad \Gamma_1 < \Gamma'_1 \quad \Gamma'_2 < \Gamma_2 \quad \rho < \hat{\rho} \quad \kappa < \tau}{\Gamma_1 \vdash e : \hat{\rho} : \tau \dashv \Gamma_2}$$

$$\text{FRAME-OS} \quad \text{OneShot } \rho \quad \frac{\Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2}{\Gamma'; \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma'; \Gamma_2}$$

$$\text{FRAME-MS} \quad \text{Copy } \Gamma' \quad \frac{\Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2}{\Gamma'; \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma'; \Gamma_2}$$

Figure 20: Selected typing rules of *Haffel*

5 Semantic Typing

The purpose of this section is to prove that the type and effect system of *Haffel* is sound. Type soundness ensures that well-typed programs are safe to execute:

$$\vdash e : \langle \rangle : \tau \dashv \Longrightarrow \text{safe } e$$

As mentioned in section 3.3.1, a safe expression cannot get *stuck* which in our context gives us three guarantees: (i) the expression either diverges or reaches a value, (ii) one-shot continuations are called at most once and (iii) no effect is left unhandled.

We take a semantic approach to proving type soundness, which means that we interpret types, signatures, effect rows and typing judgements into a semantic domain that captures the notion of safe expressions. Our semantic domain is the program logic *Hazel* that is built on top of the higher-order separation logic of *Iris* [Jun+18]. It can thus be considered as a logical approach to semantic type soundness.

We begin the section with a brief overview of semantic type soundness (section 5.1) and a short introduction to the Iris Logic (section 5.2) before explaining the program logic *Hazel* (section 5.3) and lastly giving the semantic definition of the type system (section 5.4).

5.1 Overview

Syntactic Type Soundness The most common means of proving type soundness is through syntactic proofs such as Progress and Preservation [Har16] which is based on earlier syntactical approaches by Wright and Felleisen [WF94]. Syntactical Progress and Preservation proofs are two lemmas which specify that a (syntactically) well-typed program is either a value or it can take a step in the execution, and further that types are preserved during execution. Together these two theorems imply that a well-typed program is safe to execute, in the sense that it cannot get stuck. To prove these theorems some form of induction on the derivation of the syntactical type judgement and evaluation relation is needed hence the syntactic characterisation.

Semantic Type Soundness Semantic type soundness does not rely on induction on the derivation of syntactic type judgements to show safety but instead types and judgements are given an interpretation in semantic domains such as complete partial orders (cpos) [Mil78] or as propositions and predicates in a higher-order logic [AM01; Ahm06; Tim+22]. It relies on building a logical relation, that is, a relation inductively defined on types [Mil78; AM01; Ahm06; Tim+22], that relates types to objects in the semantic domain. In this thesis we take a logical approach to semantic typing with the main idea that types become predicates over values and judgements are propositions.

$$\begin{array}{ccc}
 \vdash e : \langle \rangle : \tau \dashv & & \text{safe } e \\
 \swarrow \text{Fundamental} & & \nearrow \text{Adequacy} \\
 \vDash e : \langle \rangle : \tau \vDash & &
 \end{array}$$

A semantic type judgement which we use the $\Gamma_1 \vDash e : \rho : \tau \vDash \Gamma_2$ notation for, states that e is safe, performs effects according to the semantic effect row ρ and its resulting value (if it terminates) satisfies τ . Safety follows directly from the definition of the type judgement (*Adequacy* arrow). Environments Γ , rows ρ and types τ in a semantic type judgement differ from their syntactic counterparts used in a syntactic type judgement

$\Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2$. They represent objects in our semantic domain and thus in our case are predicates and propositions that capture what it means to be an inhabitant of a type and to perform an effect.

In the semantic world, typing rules become lemmas that we need to prove. The premises of the typing rule become the assumed propositions that we use to derive the semantic interpretation of the rule's conclusion. For instance, the following semantic LET sequencing rule is a theorem we prove.

$$\frac{x : \kappa, \Gamma_2 \vDash e_2 : \rho : \tau \dashv \Gamma_3 \quad \Gamma_1 \vDash e_1 : \rho : \kappa \dashv \Gamma_2}{\Gamma_1 \vDash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \rho : \tau \dashv \Gamma_3} \text{LET}$$

Typing judgements such as $\Gamma_1 \vDash e_1 : \rho : \kappa \dashv \Gamma_2$ are propositions in our logic. This view of typing rules as lemmas is better suited for incremental specification of typing derivations because type safety is proven directly from their semantic definitions. This contrasts with syntactical approaches that first require a complete characterisation of the typing rules to prove type safety.

Every syntactic typing rule has a corresponding *compatibility* lemma which proves the semantic version of the rule. The above semantic LET rule is the compatibility lemma for the syntactic LET rule. The fundamental lemma (*Fundamental* arrow) relates the syntactic type judgement to the semantic one by simply performing induction on the syntactic type judgement and directly applying the compatibility lemmas.

Semantic Typing of *Haffel* We use the higher-order separation logic of *Iris* as our semantic domain and specifically take advantage of existing work on program logics built on top of *Iris* that allow reasoning about effectful programs [VP21; Vil22]. Instead of defining our own program logic from scratch, we utilise the *Hazel* program logic that is designed to reason about programs written in the untyped *Hazel* language [VP20]. The same program logic is used to reason about *Haffel* programs because *Haffel* can be defined purely in terms of *Hazel*⁵: all its constructs can be derived from constructs in *Hazel*. We expand on this encoding at an as-needed basis in sections 5.3, 5.4 and 6.

Throughout this section we will avoid referring to syntactic type, signature and effect row expressions and instead adopt a semantic viewpoint where these expressions are really objects in the logic. We will not define the (straightforward) translation from syntactic to semantic expressions. In fact, this is also the approach taken in the Coq formalisation discussed in section 6 where we do not prove the Fundamental lemma but instead only consider semantic objects.

There are multiple reasons why we chose the semantic approach to type soundness given that syntactic type soundness is more intuitive and easier to explain. These can be summed up as follows:

- Run-time expressions such as `eff` and `unlift` do not have a valid typing rule and this would be a problem in syntactical proofs, specifically in the Preservation proof. One would need a richer specification of effect rows to capture the behaviour of these constructs. With semantic typing we instead rely on the underlying logic to prove safety of these run-time expressions.
- We can prove soundness of recursive features such as our equirecursive effect rows and iso-recursive types in a rather direct way from the fact that the Iris logic is a modal logic afforded by its step-indexed model.
- The support of *impredicative invariants* in the Iris logic allows us to prove soundness of features such as non-structurally treated references that can store one-

⁵With some minor extensions. This is further discussed in section 6.

shot continuations [SB14]. Such a feature is needed to give a correct typing to the Cooperative Concurrency example from section 2.3.

5.2 Iris Logic

Iris is a higher-order concurrent separation logic framework formalised in the Coq proof assistant that facilitates machine-checked verification of concurrent, higher-order imperative programs [Jun+18]. Its support of features such as *impredicative invariants*, *higher-order ghost state* and its overall reductionist methodology has made it a generic and powerful logic that is suited for a wide range of applications [SB14; Jun+16].

Iris is language-agnostic thus making it suitable for formalising and proving properties of all sorts of languages. To keep this discussion short, we will avoid presenting it in its full generality but instead focus on a fragment of the logic instantiated with the untyped *Hazel* language. As already noted, we embed *Haffel* into *Hazel* and so we can use the same logic to reason about *Haffel* programs. In general, the Iris logic is parameterised with a user-defined notion of resources and can be extended by user-defined types and type formers.

Figure 21 shows that the Iris logic extends the simply-typed lambda calculus with types for logical propositions *iProp* and with a set of base types and type formers (their corresponding terms are not shown for reasons of brevity). Apart from the usual propositional logic connectives, it features terms for higher-order quantification and separation logic and Iris-specific connectives. Iris has more types and (primitive) logical connectives such as update modalities and ghost state assertions but we do not present them as we will not directly use them in our semantic definitions (section 5.4). The Iris logic can be described as:

- **Higher-Order:** Allows quantification over types including propositions (*iProp*).
- **A Separation Logic:** Through its support of the usual separation logic connectives such as separating conjunction ($*$) and separating wand (\multimap).
- **A Modal Logic:** Has modalities such as persistence $\Box P$ which states that P is shareable knowledge and later $\triangleright P$ which states that P holds after one step of computation.

$ \begin{aligned} & A, B := 0 \mid 1 \mid \text{Bool} \mid \mathbb{N} \mid \mathbb{Z} \mid \text{Mode} \mid \text{Label} \mid \text{Loc} \mid \text{Var} \mid \text{Val} \mid \text{Expr} \mid \text{iProp} \\ & \quad \mid A \times B \mid A + B \mid \text{List } A \mid A \rightarrow B \mid A \xrightarrow{\text{fin}} B \mid \dots && \text{(Types)} \\ & t, u, P, Q := x \mid \lambda x : A. t \mid t(u) \mid \dots && \text{(Lambda Calculus)} \\ & \quad \mid \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q && \text{(Propositional Logic)} \\ & \quad \mid \forall x : A. P \mid \exists x : A. P \mid t =_A u && \text{(Higher-order logic)} \\ & \quad \mid P * Q \mid P \multimap Q \mid \ell \mapsto v \mid \text{ewp}_{\mathcal{E}} e \langle \Psi \rangle \langle \Psi' \rangle \{ \Phi \} && \text{(Separation Logic)} \\ & \quad \mid \triangleright P \mid \Box P \mid \text{fix}(x : A). t \mid \boxed{P}^{\mathcal{N}} \mid \dots && \text{(Iris-specific connectives)} \end{aligned} $
--

Figure 21: A fragment of the syntax of Iris (instantiated with the *Hazel* language)

A semantic model of Iris that proves soundness of the logic is based on Ordered Family of Equivalences (OFEs), that is, sets with a downwards-closed, step-indexed equality relation [DM02]. Crucially, the step-indexed nature of OFEs allow it to model *higher-order ghost state* and further allows taking fixpoints of functions under certain

conditions. For a detailed description of the Iris logic including its model and proof rules we refer the reader to Jung et al. [Jun+18]. Below we highlight the main logical connectives that we will use throughout this section.

Program Logic Connectives The points-to $\ell \mapsto v$ and extended weakest precondition $\text{ewp}_\varepsilon e \langle \Psi \rangle \langle \Psi' \rangle \{ \Phi \}$ connectives are related to the program logic *Hazel*. They are derived from the primitive logical connectives of Iris and are based on a specific resource instantiation that models the heap. These two connectives are described in section 5.3.

Later Modality The later modality (\triangleright) allows reasoning about the future defined in terms of computation steps. A proposition $\triangleright P$ states that P holds after one step of computation. It introduces the Löb rule $(\triangleright P \rightarrow P) \vdash P$ which is essential in proving properties about recursive, possibly non-terminating programs. It allows a form of inductive reasoning based on the number of steps instead of on the structure of the program’s inputs.

The later modality additionally allows us to take fixpoints of guarded recursive definitions. For instance, an abstraction $\lambda x. P$ where all its uses of variable x are guarded by a later modality ($\triangleright x$) in proposition P , has a fixpoint specified by $\text{fix}(x : iProp). P$ (See Section 5.5 and 5.6 in [Jun+18] for further information).

Persistence Modality The persistence modality (\square) asserts non-exclusive ownership of a proposition. Iris is a substructural (affine) logic and the persistence modality is used to represent non-substructurally treated propositions. Its main proof rule that allows us to duplicate propositions is: $\square P \vdash \square P * P$.

Throughout the upcoming sections we call a proposition P *persistent* when it can be freely copied, thus satisfies the following rule: $P \vdash \square P$. Common examples of persistent propositions are equalities and invariant assertions $\boxed{P}^\mathcal{N}$. Naturally we will use persistent propositions to model copyable types in our language. See Section 5.3 in [Jun+18] for further elaboration.

Impredicative Invariants Propositions $\boxed{P}^\mathcal{N}$ allow us to place invariants on the state of resources. Once an invariant is allocated, proposition P will hold in all future steps. To ensure the invariant holds, a user can only open an invariant for the duration of one atomic step of execution and they must ensure that the invariant holds at the end. Invariants $\boxed{P}^\mathcal{N}$ are shareable knowledge in the sense that they are persistent: $\boxed{P}^\mathcal{N} \vdash \square \boxed{P}^\mathcal{N}$. This allows us to treat proposition P non-substructurally under the conditions placed by invariant opening and closing.

Invariants in *Iris* are impredicative, meaning that proposition P can mention other invariants itself. This crucial property allows us to model *Haffel’s* higher-order references in a non-substructural way even in the presence of impredicative type quantification [Ahm06].

The \mathcal{N} superscript is an invariant denotes its namespace and it is used to ensure soundness of the logic by tracking enabled invariants so that they are not reopened. We refer the reader to Sections 2.2 and 7.1 in [Jun+18] for further explanations.

5.3 Program Logic

To allow high-level reasoning about effectful programs we utilise the program logic *Hazel*. The *Hazel* program logic is an extension of the generic program logic of Iris for OCaml-like programs, that introduces protocols to reason modularly about programs that perform one- and multi-shot effects [VP20]. In this section we give a quick overview of the program logic by going through a simple program specification (section 5.3.1) and

the main proof rules of the logic (section 5.3.2), followed by an introduction to effect protocols (section 5.3.3) and lastly by giving its adequacy theorem (section 5.3.4). For a complete and thorough description of the program logic we refer the reader to Vilhena [Vil22].

5.3.1 Specifications

Hazel is a separation logic and thus contains all the separation logic connectives such as separating conjunction ($*$) and wand (\multimap). It can thus be viewed as a Hoare logic for reasoning about safety and (partial) program correctness using proof rules that capture the evaluation behaviour of expressions. The proof rules themselves are based on the operational semantics of the language and on a notion of state that models the heap.

Our specifications will have the form $P \vdash Q$, where P and Q are (*Iris*) propositions that can be intuitively seen as predicates over heaps. We use logical entailment (\vdash) to relate the two by requiring that Q follows from P . In the predicates over heap setting, we define entailment as stating that for any heap σ , $P\sigma$ implies $Q\sigma$. Defined locations in the heap can be specified using the *points-to* connective $\ell \mapsto v$ which states that location ℓ is defined and stores the value v . We can compose multiple independent locations together by joining them using the separating conjunction: $\ell \mapsto v * \ell' \mapsto w$. Note that due to the modular nature of separation logic, we must have that ℓ and ℓ' are distinct locations in the heap for this proposition to hold.

Extended Weakest Preconditions, $(\text{ewp } e \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{v. \Phi v\})$, are propositions in the logic which state that expression e is safe to execute and if it reaches a value, that value satisfies predicate Φ . In addition, it can perform two kinds of effects according to protocols Ψ and Ψ' . The weakest precondition gives us the guarantee that one-shot continuation values $\text{cont } \ell N$ are only created according to protocol Ψ and multi-shot continuations $\text{kont } N$ only according to Ψ' . Non-effectful expressions are given the \perp protocol which states that no effects are possible. We use the notation $\text{ewp } e \{v. \Phi v\}$ instead of $\text{ewp } e \langle \perp \rangle \langle \langle \perp \rangle \rangle \{v. \Phi v\}$ to state this and likewise use $\text{ewp } e \langle \Psi \rangle \{v. \Phi v\}$ instead of $\text{ewp } e \langle \Psi \rangle \langle \langle \perp \rangle \rangle \{v. \Phi v\}$. Protocols capture the communication behavior of expression and handler such as the values they are allowed to exchange with each other. Additionally, they track the transfer of resource ownership in the sense that if during verification of expression e we have ownership of proposition P we can transfer this proposition to the verification of the handler using the protocol (or vice versa). In this sub-section we will concern ourselves only with non-effectful expressions and defer their description to section 5.3.3.

Using logical entailment and extended weakest preconditions we can define Hoare triples as a derived construct as follows:

$$\{P\} e \{\Phi\} := P \vdash \text{ewp } e \{\Phi\}$$

To illustrate how program specifications are written we will use the following expression *callCont* that is reminiscent of calling a one-shot continuation by getting stuck when a location ℓ maps to true.

$$\text{callCont}(\ell) := \text{if } !\ell \text{ then } 1 \ 2 \ \text{else } \ell := \text{true}$$

The specification we want to show is that given location ℓ is defined in the heap and points to the value false, *callCont*(ℓ) is safe to execute and if it terminates the location is updated to true. Note that the free variables in the proposition are implicitly quantified over and we ignore the resulting value of the expression:

$$\ell \mapsto \text{false} \vdash \text{ewp } (\text{if } !\ell \text{ then } 1 \ 2 \ \text{else } \ell := \text{true}) \{v. \ell \mapsto \text{true}\}$$

Notice that $callCont$ is not a well-typed expression due to the application of literal 1 with 2. However we are only concerned with safety and program correctness and so the above specification is derivable in the logic since $callCont$ is safe when ℓ stores false.

5.3.2 Proof Rules

WP-VAL $\Phi v \vdash \text{ewp } v \{ \Phi \}$	WP-PURE $\frac{\forall \sigma. e_1 / \sigma \rightsquigarrow e_2 / \sigma}{\triangleright (\text{ewp } e_2 \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \}) \vdash \text{ewp } e_1 \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \}}$
WP-BIND $\text{ewp } e \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ v. \text{ewp } N[v] \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \} \} \vdash \text{ewp } N[e] \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \}$	
WP-LOAD $\ell \mapsto v * \triangleright (\ell \mapsto v * \Phi v) \vdash \text{ewp } !\ell \{ \Phi \}$	WP-STORE $\ell \mapsto v * \triangleright (\ell \mapsto w * \Phi v) \vdash \text{ewp } (\ell := w) \{ \Phi \}$
WP-FRAME $R * \text{ewp } e \langle \Psi \rangle \{ \Phi \} \vdash \text{ewp } e \langle \Psi \rangle \{ v. \Phi v * R \}$	
WP-PERS-FRAME $\square R * \text{ewp } e \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \} \vdash \text{ewp } e \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ v. \Phi v * R \}$	

Figure 22: Selected proof rules of *Hazel*

To describe how the $callCont$ specification can be derived we must first give the key proof rules of *Hazel* that allow us to reason about *Hazel* programs. Shown in fig. 22, the WP-VAL, WP-PURE, WP-BIND, WP-LOAD, WP-STORE and WP-FRAME rules are similar to the ones found in the *Iris* generic program logic with the exception of the existence of protocols.

The WP-VAL rule is used for values and expects a non-effectful extended weakest precondition since values are not evaluated and so do not produce effects. The WP-PURE rule allows taking a single pure step in the execution by requiring that after one step (due to later modality) the resulting weakest precondition for e_2 holds.

Recall from fig. 5 that an evaluation context is neutral if it does not span through a handler. Interestingly, the WP-BIND rule that allows modular reasoning is restricted to only hold for neutral contexts N instead of arbitrary contexts K . This is needed to ensure that no intermediary handlers are captured in the context which can alter the execution behaviour of expression e .

The WP-LOAD and WP-STORE rules are the standard proof rules for reading and storing from a reference respectively. Since in a proposition $P * Q$, the separating conjunction ensures that the heap that P and Q access is disjoint, we use the separating wand for the proof of Φ in the premise to specify that we can assume ownership of location ℓ during its proof.

There are two rules for framing propositions through the weakest precondition, WP-FRAME and WP-PERS-FRAME. Continuations of effects have a weakest precondition associated with them which inherits the postcondition Φ of the expression that performed it. If an effect is produced by protocol Ψ' , the associated weakest precondition is persistent to ensure that it can be called multiple times. For this reason, to extend the postcondition, proposition R must be persistent. We further expand on this in section 5.3.3.

We can now utilise these proof rules to prove the *callCont* specification. In the following derivation, we implicitly apply transitivity of logical entailment between each application of a proof rule.

$$\begin{array}{c}
\frac{}{\vdash\text{-REFL}} \\
\frac{\ell \mapsto \text{true} \vdash \ell \mapsto \text{true}}{\vdash \ell \mapsto \text{true} \text{ } * \ell \mapsto \text{true}} \text{-*}\text{-INTRO} \\
\frac{}{\vdash \ell \mapsto \text{false} \vdash \text{ewp}(\ell := \text{true}) \{v. \ell \mapsto \text{true}\}} \text{WP-STORE,*-MONO-RE} \\
\frac{}{\ell \mapsto \text{false} \vdash \triangleright (\text{ewp}(\ell := \text{true}) \{v. \ell \mapsto \text{true}\})} \triangleright\text{-INTRO} \\
\frac{}{\ell \mapsto \text{false} \vdash \text{ewp}(\text{if false then 1 2 else } \ell := \text{true}) \{v. \ell \mapsto \text{true}\}} \text{WP-PURE} \\
\frac{}{\vdash \ell \mapsto \text{false} \text{ } * \text{ewp}(\text{if false then 1 2 else } \ell := \text{true}) \{v. \ell \mapsto \text{true}\}} \text{-*}\text{-INTRO} \\
\frac{}{\vdash \triangleright (\ell \mapsto \text{false} \text{ } * \text{ewp}(\text{if false then 1 2 else } \ell := \text{true}) \{v. \ell \mapsto \text{true}\})} \triangleright\text{-INTRO} \\
\frac{}{\ell \mapsto \text{false} \vdash \text{ewp} ! \ell \{u. \text{ewp}(\text{if } u \text{ then 1 2 else } \ell := \text{true}) \{v. \ell \mapsto \text{true}\}\}} \text{WP-LOAD,*-MONO-RE} \\
\frac{}{\ell \mapsto \text{false} \vdash \text{ewp}(\text{if } ! \ell \text{ then 1 2 else } \ell := \text{true}) \{v. \ell \mapsto \text{true}\}} \text{WP-BIND}
\end{array}$$

The WP-BIND rule decomposes the weakest precondition by allowing us to firstly focus on the load expression. The postcondition of the load expression requires that its result substituted back to the neutral context can be safely evaluated and it satisfies the original postcondition. We load from location ℓ using WP-LOAD by showing that we have ownership of $\ell \mapsto \text{false}$ and that the remaining expression satisfies the specification. We note that we implicitly apply the following monotonicity property of the separating conjunction:

$$\frac{\text{*}\text{-MONO-RE} \quad \vdash Q}{P \vdash P * Q}$$

We remove later modalities using the later introduction rule $P \vdash \triangleright P$ which says that if P holds for all steps then it will also hold after one step of execution. We additionally introduce propositions into the context using the separating wand introduction rule:

$$\frac{\text{-*}\text{-INTRO} \quad P * Q \vdash R}{P \vdash Q \text{ } * R}$$

5.3.3 Protocols

The extended weakest precondition of *Hazel* relies on the notion of protocols to describe the communication between effectful expression and handler. A protocol Ψ is a function from values Val and predicates over values $Val \rightarrow iProp$ to Iris propositions $iProp$. In a proposition $\Psi v \Phi$, v will be the value passed to the `do` construct and Φ will correspond to the predicate that the continuation of the effect satisfies.

$$\begin{array}{c}
\Psi \in Prot := Val \rightarrow (Val \rightarrow iProp) \rightarrow iProp \\
\Psi \sqsubseteq \Psi' := \square (\forall v \Phi. \Psi v \Phi \text{ } * \Psi' v \Phi)
\end{array}$$

Protocols can be ordered in a standard way by requiring that for any value v and predicate Φ , $\Psi v \Phi$ implies $\Psi' v \Phi$. Protocol ordering has an associated proof rule which allows us to utilise the relation in the following way. An expression e that adheres to protocols Ψ_1 and Ψ_2 also satisfies the more general protocols Ψ'_1 and Ψ'_2 :

$$\text{WP-SUB} \quad \Psi_1 \sqsubseteq \Psi'_1 \text{ } * \Psi_2 \sqsubseteq \Psi'_2 \text{ } * \text{ewp } e \langle \Psi_1 \rangle \langle \langle \Psi_2 \rangle \rangle \{ \Phi \} \vdash \text{ewp } e \langle \Psi'_1 \rangle \langle \langle \Psi'_2 \rangle \rangle \{ \Phi \}$$

As an example of how protocols can be used to give specifications to effectful programs consider a simple handler that when passed an integer as effect value it responds (by calling the continuation) with the value incremented by one. The protocol *inc* that we use to specify this behaviour is:

$$inc = \lambda (v : Val) (\Phi : Val \rightarrow iProp). \exists i \in \mathbb{Z}. v = i * \Phi(i + 1)$$

It says that the effect value must be an integer and the predicate of the continuation is satisfied when called with the successor of the argument.

Protocol Monotonicity The argument predicate Φ of a protocol represents the continuation and will be of the form $\lambda v. \text{ewp } N[v] \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi' \}$ with N being the captured context. A protocol is characterised as monotonic or the weaker notion persistently monotonic, according to how this continuation predicate can be extended. This extension happens when we apply the frame rule and when the context N is enlarged during evaluation of an active effect expression $\text{eff}_m v N$.

$$\begin{aligned} Mono(\Psi) &:= \forall v \Phi \Phi'. (\forall v. \Phi v * \Phi' v) * \Psi v \Phi * \Psi v \Phi' \\ PersMono(\Psi) &:= \forall v \Phi \Phi'. \Box (\forall v. \Phi v * \Phi' v) * \Psi v \Phi * \Psi v \Phi' \end{aligned}$$

A monotonic protocol states that its continuation predicate can be extended in a non-persistent way whereas a persistently monotonic protocol requires the extension itself to be persistent. Every monotonic protocol is also persistently monotonic which can be derived using the elimination rule: $\Box P \vdash P$. The *inc* protocol above is monotonic because proposition $\Phi(i + 1)$ is non-persistent. If the application of predicate Φ was wrapped with a persistence modality, $\Box \Phi(i + 1)$, it would only have been persistently monotonic.

We previously mentioned that a weakest precondition, $\text{ewp } e \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \}$, gives us the guarantee that one-shot continuation values $\text{cont } \ell N$ are produced by Ψ and multi-shot ones by Ψ' . In conjunction with the above monotonic characterisation of protocols, we define one- and multi-shot effects as being:

- One-shot effect Ψ : Appears in a $\langle \Psi \rangle$ at the ewp and satisfies $Mono(\Psi)$.
- Multi-shot effect Ψ : Appears in a $\langle \langle \Psi \rangle \rangle$ at the ewp and satisfies $PersMono(\Psi)$.

As seen below, the proof rules for performing a one- or multi-shot effect simply require that the corresponding protocol satisfies effect value v and postcondition Φ . Evaluation of a do_m expression creates an active effect expression $\text{eff}_m \bullet v$ with an empty context \bullet that simplifies to a value when filled. Instead of requiring a predicate $\lambda w. \text{ewp } \bullet [w] \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \}$, the predicate is Φ is used which can derive the former by applying WP-VAL.

$$\begin{array}{ll} \text{WP-Do-OS} & \text{WP-Do-MS} \\ \Psi v \Phi \vdash \text{ewp } (\text{do}_O v) \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \} & \Psi' v \Phi \vdash \text{ewp } (\text{do}_M v) \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{ \Phi \} \end{array}$$

The monotonicity properties allow us to derive the two frame rules WP-FRAME and WP-PERS-FRAME. In essence, when applying the Frame rule, we need to show that the continuation predicate associated with the effect can be extended by framing resource R . In WP-FRAME, when expression e performs an effect, its continuation predicate will be of the form $\lambda v. \text{ewp } N[v] \langle \Psi \rangle \{ \Phi \}$ for some N and we have to extend it to $\lambda v. \text{ewp } N[v] \langle \Psi \rangle \{ v. \Phi v * R \}$ which can be shown using the monotonicity property and an appropriate induction hypothesis. In WP-PERS-FRAME we have to extend the continuation predicate in the same way but because the protocol is only persistently

monotonic the resource itself must be persistent. The reader may notice that the framing rules do not require the protocol to be monotonic or persistently monotonic even though we make use of these properties in their proof. The reason for this lies in the notion of (persistently) upwards closure of a protocol that is embedded in the definition of the weakest precondition. It can transform any protocol to a (persistently) monotonic one which is equivalent to the original protocol when the monotonicity property holds for it. This improves the usability of the proof rules by avoiding having to constantly prove monotonicity properties and instead push the proof obligation to the handler. We note that we will briefly introduce the upwards closure in section 5.4.1.

To keep this discussion short and more intuitive, we have avoided presenting the proof rules for handlers. We refer the reader again to Vilhena [Vil22] for a through explanation on these.

5.3.4 Adequacy

The adequacy theorem states that under certain assumptions, extended weakest preconditions imply safety of the expression they are parameterised over. The formal statement is as follows:

$$(\text{True} \vdash \text{ewp } e \{v. \Phi v\}) \implies \text{safe } e$$

It says that if for all heaps the proposition $\text{ewp } e \{v. \Phi v\}$ is derivable (since True satisfies all heaps), we have that expression e is safe. Safety gives us that the program cannot get stuck and so it either reaches a value or it diverges. As a corollary, we have that one-shot continuations are executed at most once which follows from the operational semantics of the language which lack a reduction rule for already called one-shot continuations. In addition, we have that no effects are unhandled. Unhandled effects are modelled in the language in the form of expression e evaluating to an active effect expression eff . Safety of e implies that e does not evaluate to a eff expression because it is not a value and no reduction rule applies to it because it resides at the root level. It is thus a stuck expression which is forbidden to occur.

The adequacy theorem follows from the definition of the extended weakest precondition which for reasons of brevity we chose not to present. Intuitively, it is defined in a way that bakes-in safety in its definition.

5.4 Semantic Definitions

In this section we give the semantic definition of the substructural type and effect system using the program logic *Hazel*. As already mentioned, the *Hazel* program logic can be used to reason about *Haffel* programs because we can encode *Haffel* to *Hazel*. We will present the type system in four parts, starting with the definition of types, signatures and effect rows (section 5.4.1), continuing with semantic typing judgements (section 5.4.2), and lastly giving the definitions of relations and constraints (sections 5.4.3 and 5.4.4).

5.4.1 Semantic Types, Signatures and Rows

Figure 23 shows the definitions of semantic types, signatures and rows. Most of the definitions are rather standard and are only adapted to incorporate effects. We refer the reader to Timany et al. [Tim+22] for a more thorough discussion on this logical approach to typing but we note that contrary to Timany et al. [Tim+22] we take a direct semantic approach and avoid interpreting syntactic expressions as they do (in the spirit of Appel and McAllester [AM01] and Ahmed [Ahm06]).

Types are functions from values to Iris propositions and thus a value satisfies a type if the (heap-indexed) proposition holds. As already noted, an Iris proposition can be

seen as a predicate over heaps and thus we can specify requirements such as “location ℓ references allocated memory”.

Signatures are pairs where the first component is a *Mode* (O or M) and the second a protocol (as described in section 5.3.3). To distinguish between one- and multi-shot signatures an explicit mode field needs to be introduced. The reasoning behind this is explained in section 5.4.2.

Effect Rows are finite maps from pairs of labels and natural numbers to effect signatures. Labels represent the name of an effect and the natural number represents its effect level. By representing effect rows as finite maps instead of lists we get that the order (up-to duplicate effect names) does not matter and instead such effect rows are identified.

$$\begin{aligned}
\tau, \kappa \in Ty &:= Val \rightarrow iProp \\
\sigma \in Sig &:= Mode \times Prot \\
\rho \in Row &:= (Label \times \mathbb{N}) \xrightarrow{\text{fin}} Sig \\
\\
\text{bool} : Ty &:= \lambda v. \exists b : Bool. v = b \\
\text{ref } \tau : Ty &:= \lambda v. \exists \ell : Loc. v = \ell * \boxed{\exists w. \ell \mapsto w * \tau w}^{\mathcal{N}_\ell} \\
\tau \xrightarrow{\rho} \kappa : Ty &:= \lambda v. \forall w. \tau w \multimap \text{ewp}'(v w) \langle \rho \rangle \{ \kappa \} \\
\mu\alpha. \tau : Ty &:= \lambda v. \text{fix}(\alpha : Ty). \triangleright (\tau v) \\
\forall\theta. \tau : Ty &:= \lambda v. \forall \theta : Row. \square (\text{ewp}'(v \langle \rangle) \{ \tau \}) \\
! \tau : Ty &:= \lambda v. \square (\tau v) \\
\\
\forall\alpha. \tau \Rightarrow^O \kappa : Sig &:= (O, \lambda v \Phi. \exists \alpha. \triangleright (\tau v) * (\forall w. \triangleright (\kappa w) \multimap \Phi w)) \\
\forall\alpha. \tau \Rightarrow^M \kappa : Sig &:= (M, \lambda v \Phi. \exists \alpha. \triangleright (\tau v) * \square (\forall w. \triangleright (\kappa w) \multimap \Phi w)) \\
\\
\langle \rangle : Row &:= \emptyset \\
(op : \sigma) \cdot \rho : Row &:= \bigcup \{ \{ (op, S i) \mapsto \sigma' \} \mid \rho(op', i) = \sigma' \wedge op = op' \} \cup \\
&\quad \bigcup \{ \{ (op', i) \mapsto \sigma' \} \mid \rho(op', i) = \sigma' \wedge op \neq op' \} \cup \\
&\quad \{ (op, 0) \mapsto \sigma \} \\
\mu\theta. \rho : Row &:= \text{fix}(\theta : Row). \rho \\
i\rho : Row &:= \bigcup \{ \{ (op, i) \mapsto (\pi_1 \sigma, \pi_2 (\uparrow \sigma)) \} \mid \rho(op, i) = \sigma \}
\end{aligned}$$

Figure 23: Semantic definitions of selected types, signatures and effect rows.

Semantic Types Base types such as `bool` are pure propositions in the sense that they do not reference any constraints on the heap. Instead they only require that the value is an element of the semantic base type such as it being a boolean.

Reference types require the value to be a location ℓ and further utilise the points-to connective to state that the location is defined and points to a value that satisfies the inner type predicate (τ). The points-to connective and τw proposition are stored behind the invariant \mathcal{N}_ℓ in order to be able to treat reference types non-substructurally. As a result, after a reference ℓ is allocated, it must always point to a value of type τ .

Affine function types $\tau \xrightarrow{\rho} \kappa$ are defined in a standard way by requiring applications where the argument satisfies τ to be safe, perform effects ρ and finally the resulting value to satisfy κ . Instead of using directly the extended weakest precondition $\text{ewp } e \langle \Psi \rangle \langle \Psi' \rangle \{ \Phi \}$ (section 5.3), we use $\text{ewp}' e \langle \rho \rangle \{ \Phi \}$ which is a wrapper around the *Hazel* one that interprets the effect row into two protocols Ψ and Ψ' . We avoid defining it here and instead elaborate on it in section 5.4.2.

Recursive types are defined in a direct way by guarding the whole type predicate with a later modality and thus allowing us to take its fixpoint. A consequence of this guardedness property is that an inhabitant of a recursive type is only valid after taking one step in the execution. This is not a problem though since to unfold a recursive type we need to perform `unfold` which takes an execution step.

A polymorphic effect type $\forall\theta. \tau$ states that the instantiation of its value $v \langle \rangle$ is a safe non-effectful expression and further its resulting value satisfies τ for any effect row θ (note that θ is free in τ). The universal quantification in the definition is a logical connective whereas the quantification in the semantic type $\forall\theta. \tau$ is only notation.

Lastly, since the persistence modality represents non-substructural propositions in the logic we use it to define unrestricted types $!\tau$ that are non-substructural.

Semantic Signatures Multi-shot signatures $\forall\alpha. \tau \Rightarrow^M \kappa$ differ from one-shot ones in that the first component is an M mode and its protocol includes an additional persistence modality. Recall from section 5.3.3, value v represents the argument to the effect call and Φ represents the predicate that the continuation satisfies. Thus the protocol expects its argument v to satisfy τ for some instantiation of α and further that the predicate of the continuation holds for any value w that satisfies the resulting type κ (with the same α instantiation). The only difference between the two protocols is that in the multi-shot case the predicate is persistent and as a result it is freely duplicable. This translates to the continuation being allowed to be called multiple times. It is easy to check that the protocol of the one-shot signature is monotonic whereas the multi-shot one is only persistently monotonic (defined in section 5.3.3).

Suprisingly, both type predicates are guarded by a later modality even though signatures are not recursive themselves. This is due to the recursive nature of effect rows which require the recursive occurrences to be guarded by a later modality. The presence of this later modality is what allows us to use recursive row variables inside effect signatures.

Semantic Row The definitions of semantic effects rows are given in set notation to improve readability. Since we are implicitly quantifying over finite functions the sets we define are themselves finite.

The Nil effect row ($\langle \rangle$) is simply notation for the empty map that does not associate any effects with a signature. The Cons effect row $((op : \sigma) \cdot \rho)$ is more involved as it needs to account for all the duplicate effects and their associated levels. To do this, we firstly select all (op, i) effects from ρ and increase their level by one, we then union the result with all the other effects that have a different effect label (op') and lastly we insert signature σ to key $(op, 0)$. It is clear from the definition that all the singleton maps that we specify are disjoint since they either differ in the effect label or in the effect level. In this way we capture the implicit effect levels of the list-like notation that we use to specify effect rows and translate that to their semantic definition as a finite map.

Recursive effect rows are defined as the fixpoint of $(\lambda\theta. \rho)$. There is an implicit assumption here that allows us to take its fixpoint. The recursive effect row seen as a function $(\lambda\theta. \rho)$ has to be contractive, which in our case translates to requiring that all uses of θ are guarded by a later modality. Syntactically we can enforce this by restricting the use of variable θ to only effect signatures since their types are guarded. Semantically we cannot have such a syntactic restriction and thus we leave it as an implicit assumption that follows directly if we only use θ in signatures.

Lastly, the restricted effect row $i\rho$, is defined as taking the upwards closure of the protocol of every signature defined in ρ . The upward closure of an arbitrary protocol Ψ

is a monotonic protocol that is defined in the following way:

$$(\uparrow \Psi) v \Phi := \exists \Phi'. \Psi v \Phi' * (\forall w. \Phi' w \multimap \Phi w)$$

The upwards closure of a monotonic protocol has the property that it is equivalent to the protocol itself:

$$\text{Mono}(\Psi) \vdash \uparrow \Psi \equiv \Psi$$

where the equivalence is defined according to protocol ordering \sqsubseteq . Since the protocol of one-shot signatures is monotonic, all one-shot signatures in ρ are equivalent to their mapped ones in $i\rho$. This is what allows us to sub-type a restricted effect row $i\rho$ to ρ in RES-ELIM rule.

5.4.2 Semantic Typing Judgement

The semantic type judgement differs from its syntactic counterpart in that it is a proposition which states that its parameterised expression is safe to execute, performs only effects described by the effect row and its resulting value (if it ever reaches one) satisfies the specified type.

To show how such a proposition is defined we firstly introduce the extended weakest precondition of *Haffel*, ($\text{ewp}' e \langle \rho \rangle \{\Phi\}$), shown in Figure 24. It is defined in terms of the weakest precondition of *Hazel* by interpreting an effect row ρ as both a one- and multi-shot protocol. The $\xi(\rho)$ function interprets an effect row ρ from a finite map of signatures to a protocol in *Hazel*.

The *Haffel* language supports named effects and has the notion of effect levels while the *Hazel* language only features unnamed effects. To translate effect rows into protocols, we must firstly encode named effects in *Hazel* which we do in the following way:

<i>Haffel</i>	<i>Hazel</i>
$\text{do}_m \text{op } e$	$\text{do}_m ((\text{op}, 0), e)$
$\text{eff}_m (\text{op}, i) v N$	$\text{eff}_m ((\text{op}, i), v) N$

The do construct of *Haffel* that generates a level 0 effect of label op is encoded to *Hazel* by taking a triple $((\text{op}, 0), e)$ as effect value. Similarly, the active effect expression is encoded in a direct way that is compatible (in terms of the step reduction relation) with the do construct. Thus to encode named effects in *Hazel*, we simply add the effect label and level to the effect value.

For this reason, to define the protocol interpretation of an effect row $\xi(\rho)$, we require that the effect value has the form $((\text{op}, i), w)$ which is compatible with the encoding. Additionally, when the effect op at level i is requested, the protocol must ensure that it is present in the effect row $(\rho(\text{op}, i) = \sigma)$ and the protocol of signature σ satisfies the value w and predicate Φ .

Using the protocol interpretation function we can define the extended weakest precondition of *Haffel* in terms of the one in *Hazel* by interpreting the effect row into two protocols. Recall that the weakest precondition, $\text{ewp } e \langle \Psi \rangle \langle \langle \Psi' \rangle \rangle \{\Phi\}$, features two protocols with the guarantee that one-shot continuation values are produced only by protocol Ψ and multi-shot continuation values only by protocol Ψ' .

This guarantee is essential to prove the typing rule of multi-shot handlers which requires that the continuation value produced from a multi-shot signature is of the form $\text{kont } N$. As a result, we can type it with the unrestricted function type and use it multiple times in the effect branch body. Based on this observation, we interpret only one-shot signatures for the protocol Ψ using a filter operation. Interestingly, we still allow one-shot signatures in the protocol Ψ' as we interpret the whole effect row $\xi(\rho)$. This means that when handling a one-shot signature, we might get both continuation

values: $\text{cont } \ell N$ and $\text{kont } N$. The typing rule for the effect branch guarantees that the continuation will be used at most once and so the typing rule is still sound. Additionally, this additional presence of one-shot signatures in Ψ' is what allows us to sub-type a multi-shot signature to a one-shot one (using EFF-MODE).

The filtering operation that is performed on the effect row is also the reason for the presence of a mode in a signature. If we had only used a *Hazel* protocol to represent effect signatures, we could not have defined the filter operation because the protocols of one- and multi-shot signatures are not necessarily distinct. For instance, a property such as monotonicity $\text{Mono}(\Psi)$ would be insufficient to select only the one-shot signatures from a row ρ . Multi-shot signatures such as $(\text{C}) \Rightarrow^M \text{void}$ are still monotonic and thus we would not have the guarantee that multi-shot signatures create only multi-shot continuation values.

$$\begin{aligned} \S(\rho) : \text{Prot} &:= \lambda v \Phi. \exists (w : \text{Val}) (op : \text{Label}) (i : \mathbb{N}) (\sigma : \text{Sig}). \\ &\quad v = ((op, i), w) * \rho(op, i) = \sigma * (\pi_2 \sigma) w \Phi \\ \text{ewp}' e \langle \rho \rangle \{ \tau \} : i\text{Prop} &:= \text{ewp } e \langle \S(\text{filter } (\lambda \sigma. \pi_1 \sigma = \text{O}) \rho) \rangle \ll \S(\rho) \gg \{ \tau \} \end{aligned}$$

Figure 24: The extended weakest precondition of *Haffel*.

Using the newly defined extended weakest precondition of *Haffel*, we can define semantic typing judgements as shown in fig. 25. To do this, we firstly define the interpretation of an environment as a predicate from a finite map of variables to values into a proposition. In essence, for the proposition $\Gamma \models \gamma$ to hold, the finite map γ must define a mapping for every variable mentioned in Γ and further the value that it maps to must satisfy the specified type.

The judgement $\Gamma_1 \models e : \rho : \tau \ni \Gamma_2$ is a proposition defined using the weakest precondition of *Haffel*. It requires that for any finite mapping of variables to values (γ) that satisfy initial environment Γ_1 , the expression e with the values substituted in $\gamma(e)$ is safe to execute, performs only effects of row ρ and the resulting value is of type τ . Additionally, the final environment Γ_2 must also be satisfied for the same variable map γ after evaluation of e . The fact that proposition $\Gamma_2 \models \gamma$ appears in the postcondition is what allows us to use the variables in Γ_2 after evaluation of e .

$$\begin{aligned} \Gamma \in \text{Env} &:= \text{List}(\text{Var} \times \text{Ty}) \\ \gamma \in \text{VarMap} &:= \text{Var} \xrightarrow{\text{fin}} \text{Val} \\ \Gamma \models \gamma &: i\text{Prop} \\ \square \models \gamma &:= \text{True} \\ (x, \tau), \Gamma \models \gamma &:= \exists v. \gamma(x) = v * \tau v * \Gamma \models \gamma \\ \Gamma_1 \models e : \rho : \tau \ni \Gamma_2 : i\text{Prop} &:= \square (\forall \gamma. \Gamma_1 \models \gamma \rightarrow * \\ &\quad \text{ewp}' \gamma(e) \langle \rho \rangle \{ v. \tau v * \Gamma_2 \models \gamma \}) \end{aligned}$$

Figure 25: Definition of semantic type judgements and semantic environments.

5.4.3 Semantic Relations

The sub-typing relations on type, signatures, rows and environments are defined directly as propositions in the logic. As shown in fig. 26, relations are persistent propositions which allows us to use them multiple times ($(m < m')$ is also persistent because it consists of equalities).

The relation on types ($\tau < \kappa$) is defined as requiring that every value that is satisfied by τ is also satisfied by κ . To relate environments we firstly interpret them into the logic and then require that the second environment follows from the first. In essence it simply extends the relation on types to environments.

The $m < m'$ relation states that for two modes to be related, they are either the same or they are of the form $M < O$. It is used to define the signature relation which uses the mode relation for the first component of the signatures and the protocol ordering for the second. The signature relation is extended to effect rows ($\rho < \hat{\rho}$) by requiring that every signature in ρ referenced by label op and level i has a corresponding related signature in $\hat{\rho}$ according to relation $<$. Since effect rows are finite maps semantically, the existential quantification is unique: there can only be one signature that has key (op, i) .

$$\begin{array}{l}
 \tau < \kappa : iProp := \square (\forall v. \tau v \multimap \kappa v) \\
 \Gamma_1 < \Gamma_2 : iProp := \square (\forall \gamma. \Gamma_1 \models \gamma \multimap \Gamma_2 \models \gamma) \\
 \\
 m < m' : iProp := m = m' \vee m = M \\
 \sigma < \hat{\sigma} : iProp := \square (\sigma.1 < \sigma.2 * \sigma.2 \sqsubseteq \hat{\sigma}.2) \\
 \rho < \hat{\rho} : iProp := \square (\forall op\ i\ \sigma. \rho(op, i) = \sigma \multimap \exists \hat{\sigma}. \hat{\rho}(op, i) = \hat{\sigma} * \sigma < \hat{\sigma})
 \end{array}$$

Figure 26: Definition of semantic relations

Relation judgments $\Sigma \vdash A \leq B$ do not have a corresponding semantic version, but instead they correspond to the proposition $A \leq B$ (for any of the four relations we define) with all the relations in Σ being assumed propositions. Intuitively, it is equivalent to defining the judgement by induction on the context and assuming each proposition using the separating wand:

$$\begin{array}{l}
 \models A \leq B := A \leq B \\
 A \leq B, \Sigma \vdash C \leq D := A \leq B \multimap \Sigma \vdash C \leq D
 \end{array}$$

From these relations we can prove the semantic version of the typing rule SUB'. To show $\Gamma_1 \models e : \rho : \tau \equiv \Gamma_2$ from $\Gamma'_1 \models e : \hat{\rho} : \kappa \equiv \Gamma'_2$ we can utilise the type and environment relations directly as they are simply defined using a separating wand. To show that expression e also satisfies the $\hat{\rho}$ effect row we utilise the WP-SUB proof rule. This boils down to showing that the following protocol orderings hold:

$$\begin{array}{l}
 \S(\rho) \sqsubseteq \S(\hat{\rho}) \\
 \S(\text{filter}(\lambda \sigma. \pi_1 \sigma = o) \rho) \sqsubseteq \S(\text{filter}(\lambda \sigma. \pi_1 \sigma = o) \hat{\rho})
 \end{array}$$

We can prove the first relation because protocol interpretation is a monotone operation that respects the $<$ relation. For the second relation we can show that any signature with mode o in ρ must also have mode o in $\hat{\rho}$ due to $o /< M$. This makes the filter operation monotone in the $<$ relation as well and thus protocol ordering follows from transitivity.

5.4.4 Semantic Constraints

The copyable constraints specify that the type or environment they mention can be treated non-substructurally. In the logic substructurality corresponds to persistent propositions, which allow multiple use through the proof rule $(\Box P \vdash \Box P * P)$. As shown in fig. 27 we define the copyable constraint ($\text{Copy } \tau$) by requiring that any value v that satisfies τ is also a persistent proposition. We extend this definition to environments by taking its logical interpretation.

$\begin{aligned} \text{Copy } \tau &:= \Box (\forall v. \tau v \multimap \Box (\tau v)) \\ \text{Copy } \Gamma &:= \Box (\forall \gamma. \Gamma \models \gamma \multimap \Box (\Gamma \models \gamma)) \\ \\ \text{OneShot } \rho &:= \text{Mono}(\S(\rho)) \end{aligned}$
--

Figure 27: Definition of semantic constraints

We define the ($\text{OneShot } \rho$) one-shot row constraint as requiring the protocol $\S(\rho)$ to be monotonic. It is not hard to see that protocol $\S(\rho)$ is monotonic when every protocol of the signatures in ρ is. As a result we can show that the OS-CONS in the one-shot row derivation rules holds (fig. 19) since protocols of one-shot signatures are monotonic. The rule OS-RES also holds because the upwards closure of a protocol results in a monotonic protocol. The rule for recursive effect rows follows from the fact that the recursive variable is only used in effect signatures.

The one-shot row constraint is used in the one-shot versions of certain typing rules (fig. 20) to ensure that multi-shot effects cannot happen. Semantically we use the monotonic property in the proofs of these typing rules by utilising its framing-like property that allows us to extend the predicate that is applied to the protocol (see section 5.3.3).

6 Coq Formalisation

The *Haffel* language and its substructural type and effect system has been fully formalised in the *Coq* proof assistant. We have proved that the system satisfies type safety by taking a semantic approach to typing as described in section 5. In addition, we have verified that all three examples from section 2 are well-typed terms in *Haffel*.

Mechanisation. The *Haffel* formalisation depends on the *Coq* formalisation of the *Iris* logic and the *Hazel* program logic built on top of it. *Iris* is formalised via a *shallow-embedding* approach where logical connectives are defined directly according to their semantic definitions in the *Iris* model. This contrasts with *deep-embedding* approaches that explicitly define the syntax of the logic which is then interpreted in the model. The shallow-embedding approach offers numerous advantages such as allowing the data types and corresponding lemmas of *Coq* to be freely used in the logic, and avoids the need for binder encoding approaches such as De Bruijn indices. We refer the reader to Jung et al. [Jun+18] for a more thorough discussion on the two embedding approaches.

To prove the typing rules of *Haffel* we make use of the Iris Proof Mode [Kre+18; KTB17] which augments the standard *Coq* proof mode with separation logic related features to allow higher-level reasoning and hides obtrusive details relating to the embedding of *Iris* in *Coq*. In addition, we make use of type classes to automatically infer constraints such whether a type is copyable and whether a row is one-shot.

We additionally utilise the formalisation of the *Hazel* language and program logic [VP20]. We avoid defining the programming language semantics of the *Haffel* language explicitly but instead define its embedding to the existing *Hazel* language with only a few extensions. This allows us to reuse all the existing proof rules and tactics of *Hazel* to reason about *Haffel* programs. When proving the proof rules of *Haffel*, we make extensive use of the tactical support of MoSeL for reasoning about separation logic as well as tactics offered by the *Hazel* logic.

Semantic Approach. We take a purely semantic approach to typing, as advocated by Appel and McAllester [AM01] and Ahmed [Ahm06], in the sense that (1) we prove type safety by interpreting types and judgments into a semantic domain and (2) we avoid defining syntactic expressions for types, signatures, effect rows and typing judgements but instead view them directly as semantic objects in our logic. As a result of the latter, we do not prove a fundamental lemma for our system (see section 5), which defines the interpretation of the syntactic expressions into the semantic domain.

We have elaborated on the benefits of semantic type safety in section 5.1. In addition, the purely semantic approach which avoids dealing with syntactic expressions reduces the formalisation effort needed. Binders for type and effect quantification in the language are represented directly using the binders of *Coq*. This circumvents the daunting problems of dealing with variable binders and avoids first-order binder representations such as De Bruijn indices. In addition, by virtue of using the binders of *Coq*, we do not have to explicitly ensure that created type and effect variables are fresh which is commonly done by incorporating a new environment in the typing judgement that tracks all existing variables.

Embedding. To give an overall idea of how *Haffel* is embedded into *Hazel*, in this paragraph we elaborate on the changes we had to introduce to *Hazel* and additionally explain how effect lifting constructs and handlers are encoded. The *Hazel* language is untyped and supports only unnamed effects. In order to embed *Haffel* into it we firstly introduce three new features to *Hazel* which are described below.

- Introduction of a `Replace` construct that is identical to the store operation on ref-

erences with the only difference that the previous value that was stored is returned (in *Hazel* the store operation returns the unit value). The `Replace` construct is used to represent the store operation ($e_1 := e_2$) in *Haffel*.

- Support for string values which are used to represent effect names in *Haffel*.
- A new `IsContOS` unary operation is introduced that returns true or false according to whether its argument is a continuation value of the form `cont ℓ N` or `kont N`. This operation is used to define the handlers of *Haffel*.

The `Replace` operation allows us to store one-shot continuation values in references that are treated non-substructurally by being able to update and retrieve its old value in one atomic step. To allow references to be treated non-substructurally, we need to place an invariant in the semantic definition of reference types which ensures that references always point to values of the same type (see fig. 23). The invariant stores the non-persistent proposition (τw) , and we can only access this proposition for the duration of one atomic step after which we must give back ownership.

The replace operation is the only way to access the contents of a reference that stores non-copyable types. The load operation requires that proposition (τw) is persistent (i.e., τ is copyable) because it needs to be duplicated: it is needed in both the result type of the expression and to close the invariant. We also can not define the replace operation in terms of a composite load followed by store operation using the standard invariants of *Iris* because it would not be atomic and the invariant cannot be satisfied at each step⁶.

We additionally introduce string values to *Hazel*, which we use in our embedding to encode effect names. No associated string operations are defined as they are not needed for the embedding.

Lastly, in order to encode the handlers of *Haffel* in *Hazel* a new `IsContOS` operation is introduced. The handlers of *Haffel* are parametrised with an effect name which is used to selectively propagate effects upwards according to whether the name of the received effect matches with that of the handler. In contrast, the handlers of *Hazel* service all the effects they receive since there are no effect names to distinguish them by.

We encode the handlers of *Haffel* into *Hazel* in three steps. The handlers of *Hazel* take only a single effect branch that is called with both continuation values. We would like to distinguish between the mode of an effect and thus we introduce mode-aware handlers `try-mode` that can handle one- and multi-shot effects differently. To implement them in *Hazel*, we utilise the `IsContOS` operation which checks if a continuation is one- or multi-shot. It calls the appropriate one-shot h_o , or multi-shot branch h_m according to the result of the operation. We note that the handler on the right specified with the `try e with e | e` syntax is a shallow handler in *Hazel*.

```

try-mode e with := try e with
  (λ x k. ho)           λ x k. if (IsContOS k) then ho x k else hm x k
| (λ x k. hm)           | λ x. r
| λ x. r

```

We can encode deep-handler versions of the above mode-aware handler:

```

deep-try-mode e with (λ x k. ho) | (λ x k. hm) | λ x. r

```

in a standard way by making use of recursive functions. The interested reader can consult the formalisation to show how this is done [Roo24].

⁶We could use thread-local invariants of *Iris* to correctly type the composite operation but that would complicate the formalisation.

Secondly, we define effect lifting (**lift**) and unlifting (**unlift**) in *Hazel* directly using a mode-aware deep handler. This contrasts with the semantics of effect lifting presented in fig. 17 which had distinct reduction rules. In the *Coq* formalisation, we instead use mode-aware handlers and thus avoid changing the semantics of *Hazel*.

```

lift op e := deep-try-mode e with   liftEff m op := λ x k. let op' = fst (fst x) in
    liftEff ∘ op                    let s = fst (snd x) in
  | liftEff M op                    let x' = snd x in
  | λ x. x                          if op' == op then
                                    k (dom ((op, s + 1), x'))
                                    else
                                    k (dom x)

```

The effect branch *liftEff* is parametrised with a mode *m* and effect name *op* and it used to define **lift** in terms of a mode-aware deep handler. As already noted in section 5.4.1, we encode effect values in *Hazel* as a triple $((op, s), v)$ with *op* being the effect name of type string, *s* the effect level, and *v* the actual value passed to the perform. Thus the *liftEff* branch inspects if the current effect name is *op*, and if it does it propagates the effect upwards by emitting the effect with one level higher. If it does not match, the effect is emitted with the value left unchanged. The inverse **unlift** operation is defined in the same way as above with the difference that level *s* is decremented by one instead of $(s + 1)$. Although not formally proven, we believe that the semantics of **lift** and **unlift** defined using a deep handler coincide with the semantics presented in Figure 17.

Using a mode-aware handler and the newly defined **unlift** construct we can recover the deep handler of *Haffel* in the following way. We define the effect branch *hEff* in a similar way as we did with **lift**, and define the deep handler of *Haffel* that is parametrised by an effect name *op* using the mode-aware deep handler of *Hazel*.

```

deep-try e with op by := deep-try-mode e with
    λ x k. h                    hEff ∘ op
  | λ x. r                      | hEff M op
                                | λ x. r

```

The **if-then** branches of *hEff* correspond to the three deep handler reduction rules in fig. 17. If the effect received has the same effect name *op* and its level is 0, we call the effect branch *h* directly. This corresponds to the first two reduction rules in the figure, since continuation *k* can either be a one-shot value **cont** ℓ *N* if we are at branch *hEff* ∘ *op* and a multi-shot value **kont** *N* we are at *hEff* M *op*. The second branch of the **if-then** expression corresponds to the last reduction rule.

```

hEff m op := λ x k. let op' = fst (fst x) in
                    let s = fst (snd x) in
                    if op' == op and s == 0 then
                        h x k
                    else
                        k (unlift op (dom x))

```

7 Related Work

In this section we discuss work related to substructural type and effect systems. We begin with a brief history of algebraic effects (section 7.1), and semantic typing (section 7.2), followed by work on substructural type and effect systems (section 7.3), we then give an overview of effect polymorphism via row-based approaches (section 7.4), and lastly describe work that interpret effect systems by building logical relations (section 7.5).

7.1 Algebraic Effects and Handlers

The foundations of algebraic effects stem from the influential work of Moggi on computational effects [Mog88]. Moggi realised that the computational behaviour of an effect can be modelled by a mathematical structure called a monad. The idea of representing computations using monads proved to be immensely useful and has been adopted in many functional programming languages with a prime example being Haskell.

Plotkin and Power expanded on Moggi’s work by formulating an operational semantics for a language with arbitrary effects in a way that coincided with the semantics presented by Moggi [PP01]. However, they required an additional assumption on the kind of computational effects the language could perform. The effects had to be algebraic in the sense that a certain commutativity property had to hold. This was the genesis of algebraic effects and subsequent research on the field slowly formulated the description of algebraic effects and their handlers that we currently see in literature.

Certain common effects are not algebraic with a notable example being exception handling. Raising an exception is algebraic but catching and handling one is not. To address this problem, Plotkin and Pretnar invented effect handlers which are dual to effect operations: effect operations produce effects and handlers service them [PP09]. Effect handlers transformed algebraic effects into a powerful programming construct by allowing the semantics of the effects to be defined inside the program. This contrasted with previous approaches that defined their behaviour solely at the mathematical level by an appropriate equational theory or computational monad.

Our work focuses on the programming language perspective of effect handlers in terms of devising typing rules and proving that they satisfy certain properties. We assume no equational theories that describe the computational properties of effects. In addition, we take a purely operational approach to effect handlers instead of a denotational approach to semantics that utilise free model constructions [PP02].

7.2 Semantic Typing

Semantic typing has a long history, originating from seminal work of Milner [Mil78], which proved type soundness for a pure fragment of an ML language. Milner used complete partial orders as the semantic domains which types were interpreted in but such denotational methods struggled to model features such as higher-order state. The work of Appel and McAllester; Ahmed in the Foundational Proof-Carrying code project have scaled semantic typing allowing it to model complex language features such as impredicative polymorphism, recursive types and generic mutable state. Their work is based on a step-indexed logical relation model defined using the operational semantics of the language which allowed them to circumvent the then limitations of denotational models [AM01; Ahm06]. Our semantic approach is based on further developments of this work, that abstracted away from explicit and tedious step-indexing arithmetic and low-level reasoning about the state via the introduction of higher-order modal and separation logics [App+07; Tim+22].

7.3 Substructural Type and Effect Systems

Substructural Type Systems are based on substructural logics that discard either the Exchange, Weakening, Contraction rule or a combination thereof [Pie04]. The *Haffel* type system is affine because it only forbids the Contraction rule, and as a result variables are used at *most* once.

As is common in affine and linear type systems, *Haffel* features an unrestricted type former (!) which closely resembles the one found in linear logic. However, we do not explicitly feature the Dereliction rule since it can be derived by the SUB rule and Promotion is *not* sound in our system [Abr93; Wad91]:

$$\begin{array}{c} \text{DERELICTION} \\ \frac{x : \kappa, \Gamma \vdash e : \rho : \tau \dashv}{x : !\kappa, \Gamma \vdash e : \rho : \tau \dashv} \end{array} \qquad \begin{array}{c} \text{PROMOTION} \\ \frac{!\Gamma \vdash e : \rho : \tau \dashv}{!\Gamma \vdash e : \rho : !\tau \dashv} \end{array}$$

where the notation $!\Gamma$ denotes that all variables in Γ have type $!\tau$ for some τ . Promotion is not sound due to the way one-shot continuations are treated. Consider the expression $\text{do}_O \text{op} ()$ with an installed handler that simply returns its continuation.

$$\text{try} (\text{do}_O \text{op} ()) \text{ with } \text{op} \text{ by } (\lambda () k. k) / \emptyset \rightsquigarrow^* (\text{cont } \ell \bullet) / \ell \mapsto \text{false}$$

This expression would eventually step to a one-shot continuation value $\text{cont } \ell \bullet$ and a new location ℓ would be allocated to point to false . Admitting the Promotion rule would allow us to derive the following judgment:

$$\vdash \text{try} (\text{do}_O \text{op} ()) \text{ with } \text{op} \text{ by } (\lambda () k. k) : \langle \rangle : !(() \frac{\langle \text{op}: () \Rightarrow () \rangle}{()} ()) \dashv$$

Since the expression is closed, it can be typed with an empty context and no effects are observed from the outside. As a result, the continuation value $\text{cont } \ell \bullet$ will be given an unrestricted type which would allow it to be called multiple times breaking type safety!

A type system such as *Haffel* that features the unrestricted type former and a sub-typing relation on unrestricted types (such as $!\tau <: \tau$) does not have principal types [Wad91]. As Wadler explains, a possible way to support principal types in such systems is via the introduction of linear dependencies between arguments. We leave such enhancements to *Haffel* for future work.

Substructural Type and Effect Systems. Hillerström, Lindley, and Longley [HLL20] devise an affine type system for a pure language with effects handlers that allows only one-shot continuations. Their main focus is comparing the asymptotic speedup gained in terms of run-time complexity when multi-shot effects instead of just one-shot effects are used. We instead focus on devising a type and effect system for a more expressive language with mutable state, multi-shot continuations, recursive types and sub-typing.

Tang et al. [Tan+24] present two calculi that soundly combine multi-shot handlers with linear resources. The motivation to do so stems from the desire to resolve a type soundness bug found in the *Links* [HL16] programming language. *Links* is a functional programming language that features both multi-shot effect handlers and session types. Session types require a linear type system to enforce that protocols described by the types are obeyed. The presence of unchecked multi-shot handlers lead to a soundness bug in the language, because the continuation of multi-shot effects could capture linear resources. As a result, the guarantee that linear resources are used exactly once was broken. The systems presented by Tang et al. are similar to ours in the sense that they solve the same problem: devising a systematic way of enforcing that continuations of effects that are used in a multi-shot way do not capture substructural resources.

Despite so, the motivation for our work differs: we devise an (affine) type and effect system for an OCaml-like language that does not feature session types but instead has

higher-order references in the form of *ML*. This work could serve as the basis for an enhancement of OCaml with a proper effect system and multi-shot effects. In the rest of this sub-section we give an overview on how the calculi of Tang et al. compare with *Haffel*.

The first system presented by Tang et al. is the F_{eff}° calculus, a System F-style fine-grained call-by-value language with effect handlers and linear types. In the F_{eff}° calculus, effect types are rows that feature a series of effect names with their signatures and ended with a possible row variable. To distinguish between one- and multi-shot effects as well as support effect polymorphism they adopt a kind-based approach where effect rows R have kind $R : \text{Row}_{\mathcal{L}}^Y$ [HL16]. The Y attribute denotes the linearity of the row, which can either be (\bullet) to mean that at least one signature is multi-shot and (\circ) to mean that all effects are one-shot (\bullet directly corresponds to M and \circ to o).

To relate the Y attribute of an effect row with the continuation produced by one of its effects, *control-flow linearity* is introduced, which tracks how often control may enter a local context. Control-flow linearity is apparent in the typing rule of their LET construct as shown below (computation types $A!\{R\}$ denote computations of type A that produce effects of row R):

$$\frac{\Delta \mid \Gamma_1 \vdash e_1 : B!\{R\} \quad \Delta \mid x : B, \Gamma_2 \vdash e_2 : A!\{R\} \quad \Delta \vdash \Gamma_2 : Y \quad \Delta \vdash R : \text{Row}^Y}{\Delta \mid \Gamma_1; \Gamma_2 \vdash \text{let}^Y x = e_1 \text{ in } e_2 : A!\{R\}}$$

The Y attribute at the LET expression is determined by the kind of row R . It states that if a multi-shot effect is present in R then the associated continuation $\text{let}^Y x = _ \text{ in } e_2$ can be used multiple times. In this case, the Y attribute would be (\bullet) and to ensure the continuation can be called multiple times, it cannot capture linear resources which is enforced by the judgement $\Delta \vdash \Gamma_2 : Y$.

In *Haffel* we avoid defining a separate notion of substructurality such as control-flow linearity. Instead, control-flow linearity is implicitly captured by the use of two-contexts judgements and the threading of contexts in the typing rules, which leads to more succinct typing rules:

$$\frac{x : \kappa, \Gamma_2 \vdash e_2 : \rho : \tau \dashv \Gamma_3 \quad \Gamma_1 \vdash e_1 : \rho : \kappa \dashv \Gamma_2}{\Gamma_1 \vdash \text{let } x = e_1 \text{ in } e_2 : \rho : \tau \dashv \Gamma_3} \text{LET}$$

If e_1 produces a multi-shot effect, its final environment Γ_2 will be copyable and as a result, expression e_2 can be evaluated multiple times since it does not capture sub-structural resources.

To prove that well-typed programs use linear resources exactly once they augment their small-step semantics with additional information that tracks which linear values are introduced and eliminated at each reduction step. The statement of reduction safety is formulated in terms of this augmented semantics and a separate notion of linear safe expressions which are expressions where linear values do not appear in unlimited contexts. This contrasts with our approach, where we model the error of calling a one-shot continuation more than once as a stuck expression. As a result, the statement of type safety is defined in the standard way that it implies expression safety which by definition means that one-shot continuations are called at most once. In addition, whereas Tang et al. prove type soundness via the syntactical approach of Progress and Preservation we instead take a semantic approach which is well-suited for our scenario. We prove soundness of our non-structurally treated references (that can store one-shot continuations) through the support of impredicative invariants in the logic which serves as our semantic domain. In addition, by virtue of using a modal logic we can easily prove soundness of recursive features such as equi-recursive effect rows.

The F_{eff}° calculus has similar expressive power with *Haffel* in the sense that both support effect polymorphism but not polymorphism over the linearity of an effect row.

To address this problem, Tang et al. introduce the Q_{eff}° calculus, which enhances F_{eff}° with qualified linear and effect types. In addition, they present a constraint solving algorithm which infers all linearity constraints without the need for annotations. Their system enjoys principal types due to the introduction of linearity variables and dependencies.

Qualified linear types add abstraction over linearity and allow specification of linear dependencies. To showcase the expressive power of linear dependencies, consider the following *verboseId* example, which is a verbose version of the identity function that additionally performs the print effect.

$$verboseId := \Lambda. \lambda x. \text{do}_M \text{ print } \text{“called”}; x$$

In *Haffel* just as with F_{eff}° we can give two possible types to *verboseId* according to whether *print* is a one- or multi-shot effect (recall that multi-shot effects can be downcasted to one-shot ones).

$$\forall \alpha. \alpha \xrightarrow{\langle \text{print} : \text{Str} \Rightarrow () \rangle} \alpha \qquad \forall \alpha. !\alpha \xrightarrow{\langle \text{print} : \text{Str} \Rightarrow () \rangle} !\alpha$$

When *print* is a one-shot effect, then α can be any arbitrary type. But if it is allowed to be handled in a multi-shot way, then α must be a non-substructural type since the variable x will be captured in the continuation.

Qualified linear types can express this dependency between the substructurality of the argument with respect to the mode of the signature.

$$\forall \alpha Y. Y \leq \alpha \Rightarrow \alpha \xrightarrow{\langle \text{print} : \text{Str} \Rightarrow^Y () \rangle} \alpha$$

Intuitively the inequality $Y \leq \alpha$ would forbid using multi-shot effects with substructural types, thus the instantiated type for α can be used at least as much as Y .

Qualified effect types introduce dependencies between effect rows and can be used to express effect polymorphism over two compatible recursive variables. For instance, we can type the following function that takes two effect polymorphic functions and applies them as follows:

$$\lambda f g. f (); g () : \forall \theta \theta_1 \theta_2. \theta_1 < \theta, \theta_2 < \theta \Rightarrow (() \xrightarrow{\theta_1} ()) \rightarrow (() \xrightarrow{\theta_2} ()) \xrightarrow{\theta} ()$$

Qualified effect types allow the effect sub-typing relation to be taken as a predicate on the function type. We note that a version of qualified effect types was present in the initial version of the *Koka* language but was later removed due to impacts on type inference [Lei14]. Leijen notes that later versions of *Koka* that do not feature such sub-typing constraints are not necessarily more restrictive since the obligation that both functions f and g perform compatible effect rows is pushed to the caller. For instance, we may give it the following function type that utilises a single recursive variable θ and leave it to the responsibility of the caller to apply the sub-typing relation on the two arguments.

$$\lambda f g. f (); g () : \forall \theta. (() \xrightarrow{\theta} ()) \rightarrow (() \xrightarrow{\theta} ()) \xrightarrow{\theta} ()$$

7.4 Effect rows and polymorphism

The Links programming language, and the calculi F_{eff}° and Q_{eff}° support effect polymorphism and address the problem of effect name clashing (see section 4.2) using a kind-based approach [HL16; HLL20; Tan+24]. Effect rows in these systems can be viewed as sets in the sense that they are order irrelevant and no duplicate effect names

can occur. In an effect polymorphic type, the recursive variable is of kind $\text{Row}_{\mathcal{L}}^Y$ where \mathcal{L} represents a set of effect names that cannot be present in any of its instantiations.

To illustrate the difference between a kind-based system and the approach we take in *Haffel*, consider again the *toYield* function from section 4.2 which yields the result of calling its argument:

$$\text{let } toYield = \Lambda. \lambda f. \text{do}_M \text{ yield } (f ())$$

Our system cannot type the *toYield* term with the following type because we cannot extend the effect of f from θ to $(\text{yield} : \text{int} \Rightarrow ()) \cdot \theta$ solely by applying the sub-typing rule. Instead we need to wrap the application of f with a lift operation ($\text{lift } \text{yield } (f x)$) which would raise the effect level of any *yield* effect that may occur in f by one.

$$toYield : \forall \theta. (() \xrightarrow{\theta} \text{int}) \xrightarrow{(\text{yield} : \text{int} \Rightarrow ()) \cdot \theta} ()$$

In the kind-based approach, the θ row variable would be given kind $\text{Row}_{\{\text{yield}\}}^\bullet$ which means that the *yield* effect cannot appear in its body and as a result an instantiation of θ with a row that has the *yield* effect is not well-kinded. This ensures that effect rows always contain unique effect names.

Biernacki et al. [Bie+17] present a type and effect system for their language that utilises effect rows with duplicate effect names as inspired by the *Koka* language [Lei17]. They additionally introduce a new effect-related construct, *lift*, in order to make effect polymorphism more expressive. We adopt similar effect rows to that of *Koka* and the lifting construct of *Haffel* is directly inspired by their calculus. In contrast to *Haffel*, they operationally present effect lifting more directly via constraints on the evaluation context. We could not adopt such an approach because *Haffel* is embedded into *Hazel* (see sections 5 and 6) and thus must be compatible with its operational semantics. As a result, we introduced effect levels to describe the intended behaviour of effect lifting.

TES, a non-substructural type and effect system built for an OCaml-like language with mutable state and effect handlers, takes an alternative approach to effect rows and polymorphism [VP23]. Vilhena and Pottier address the problem of effect name collision through the use of dynamically allocated effect labels and their custom notion of effect rows that assume a disjointness hypothesis.

Just like the kind-based approach, the *TES* system can type the *toYield* function with the aforementioned type as is. Instantiations of the function that lead to duplicate names in the resulting effect row are allowed but renders the function useless: a function with an effect row that has duplicate effect names cannot be properly handled. This is required in their system since the order of entries in an effect row does not matter. Indeed if we instantiate θ in the example with $\langle \text{yield} : \text{int} \Rightarrow \text{bool} \rangle$, we would have that *toYield* performs two yield effects with two distinct signatures. Since in *TES* we can swap entries arbitrarily, the two effect rows are identified:

$$\langle (\text{yield} : \text{int} \Rightarrow ()), (\text{yield} : \text{int} \Rightarrow \text{bool}) \rangle \equiv \langle (\text{yield} : \text{int} \Rightarrow \text{bool}), (\text{yield} : \text{int} \Rightarrow ()) \rangle$$

Such an effect row cannot be properly handled since a handler would not know whether to reply with a unit or a boolean (we assume full run-time type erasure). This contrasts with our approach that distinguishes between the two effect rows thus allowing a handler to be installed.

The way *TES* achieves this requirement that effect rows with duplicate effects are not callable is via the introduction of a new effect label allocation construct and a variation of the sub-typing relation that takes disjointness into consideration. Instead of effect names existing in a global scope, in *TES* effect names have a local scope and a separate *effect op in e* construct is present that allows effect name *op* to be used in expression *e*. In addition, the sub-typing relation allows freely extending an existing row with new entries as well as swapping entries arbitrarily.

7.5 Logical Relations and Effect Handlers

Vilhena and Pottier [VP23] semantically interpret their type and effect system by building a unary logical relation via their program logic build on top of *Iris*. The program logic they use is an extension of the generic program logic of *Iris* that introduces protocols to reason about effectful programs and it is a variation of their *Hazel* [VP20] program logic. The main difference between *Haffel* and *TES* is that the latter is non-substructural and so there is no distinction between one- and multi-shot effects and continuations. Their contribution is an effect system that introduces an alternative approach to addressing effect name collisions. In contrast, our contribution is an effect system that can detect the call usage of continuations.

Biernacki et al. [Bie+17] present a calculus with effect handlers, recursive effect rows and effect sub-typing. Their contribution is building a binary, step-indexed logical relation that allows reasoning about contextual equivalence of programs that make use of effects and handlers. Their work is formalised in the *Coq* proof assistant and is mechanically proven to be correct. By virtue of taking a semantic approach and building a binary logical relation they also prove type safety as a corollary. Although we also build a (unary) step-indexed logical relation to prove type safety, our contribution differs in that our type and effect system is sub-structural and thus is able to track the usage of continuations.

In addition, they use a pure language that does not feature mutable state and as a result their logical relation construction is simpler than ours. The introduction of state has substantial consequences to the complexity of the model. Biernacki et al. define their logical relation using step-indexed propositions whereas we use step-indexed propositions over a resource (the heap) in order to account for the state⁷. To reason about their logical relation at a higher-level and avoid explicit step-indexing arithmetic, they use an LSLR-based logic [DAB11]. LSLR is a second-order intuitionistic modal logic that is suitable for defining binary logical relation through its primitive notion of term relations and support for recursively defined relations through the use of the later (\triangleright) modality. We instead utilise the *Iris* logic which on top of abstracting over step-indexing, it is also a separation logic that allows for modular reasoning about state. It also supports impredicative invariants which are essential to prove type safety of our system by allowing us to treat reference types in a non-substructural way. In *Iris*, impredicative invariants are modelled through its support for higher-order ghost state and thus the *Iris* logic is more involved than the LSLR logic.

⁷In actuality, *Iris* propositions are more complex since they also need to support higher-order ghost state.

8 Conclusion and Future Work

In this work we have presented a substructural type and effect system for a higher-order language with effect handlers and mutable state. It distinguishes between effects that are one- or multi-shot and it guarantees that the continuation produced by a one-shot effect is called at most once. With this guarantee, a compiler can represent continuations in their most efficient form by allowing it to perform a destructive resumption when a continuation is one-shot thus avoiding an unnecessary copy. In addition, it can leverage the effect type information of each expression and thus avoid applying optimisations that are unsound in the presence of multi-shot effects.

The language we consider is expressive enough to be practically useful since apart from effect handlers that have been the main focus of this work, it also features recursive types and effects, type and effect quantification and sub-typing. As is common with substructural type systems, *Haffel* also includes unrestricted types $! \tau$ from linear logic. Interestingly, we have introduced restricted effect rows $i\rho$ which in a certain sense are dual to $! \tau$. Whereas an unrestricted types $! \tau$ can be used an arbitrary number of times, a restricted effect row $i\rho$ must be handled in a one-shot way.

To prove type soundness we have taken a semantic approach by interpreting types and judgments into a semantic domain in way that bakes in safety in their definitions. The semantic domain we used is the *Hazel* program logic which allows reasoning about effectful programs and it is built on top of the *Iris* logic, a higher-order modal and separation logic.

The *Haffel* language and type system has been formalised in the *Coq* proof assistant and all the proofs of type safety and associated typing rules have been mechanically proven to be correct. This mechanised approach to typing has been essential in our work since it has enabled us to prove the typing rules of the system with relative ease through the partial automation offered by the tactical support of the *Iris* and *Hazel* formalisation. In addition, it has relieved us from tedious tasks such as bookkeeping as is common with paper proofs.

We conclude this document with a brief overview of possible enhancements to *Haffel*.

Future Work

Type Checking and Inference. Type derivations in our system are not unique due to the presence of rules such as SUB and FRAME that can be applied at multiple places inside a derivation. This is not ideal for practical purposes since such presentations of type systems do not lend themselves well to algorithms for type inference or even type checking. As a result one important future direction would be devising syntax-directed typing rules by restricting sub-typing and framing to specific places. This would allow us to formulate (and verify) type checking algorithms as well as investigate the possibility of (partial) type inference provided we restrict *Haffel* to Rank-1 Hindley-Milner style polymorphism [Hin+69; Mil78].

Expressiveness. The *Haffel* language supports type and effect polymorphism and additionally features unrestricted types $! \tau$ from linear logic and the restricted row $i\rho$ to allow polymorphism over arbitrary substructural types and arbitrary one-shot effects. These features though cannot capture the relationship between substructural types and the mode of effects. As a result, functions can be given multiple incompatible function types which proves that *Haffel* does not enjoy principal types. The *verboseId* example from section 7 illustrates this more clearly.

The F_{eff}° calculus of Tang et al. [Tan+24] featured similar limitations and for this reason, Tang et al. introduced Qualified Linear and Effect types that can capture the linear dependencies between types and effect signatures. It would be interesting to

explore whether similar approaches can be incorporated to *Haffel* and thus address the aforementioned limitations.

In addition, we can make *Haffel* more expressive by incorporating features such as concurrency and module systems. Incorporating concurrency would not be a major undertaking, since the *Iris* logic is additionally a *concurrent* separation logic. Thus we can directly add concurrency primitives without requiring major adjustments to the semantic model and thus investigate the interaction between effect handlers and concurrency.

Reasoning. An important property used in program reasoning to justify compiler optimisations and prove that abstract data types satisfy their specifications is contextual refinement. An expression e is said to contextually refine expression e' , if any use of e inside a larger program can be replaced with e' without affecting its observable behaviour. Binary logical relations are a common technique for proving contextual refinement due to their semantic approach that avoids direct reasoning about the context that an expression resides in.

In our work, we have constructed a unary logical relation, but future work can extend it to a binary logical relation along the lines of Timany et al. [Tim+22]. This would allow us to prove contextual refinement of programs that utilise both effect handlers and mutable state. Such work would differ from the logical relation of Biernacki et al. [Bie+17] since we would also consider mutable state.

Along this direction, a prototype compiler could also be developed that utilises the usage information of continuations in the way that is intended and advocated in this work. Such a compiler can be formally verified in the sense that the compiled executable behaves exactly as described by the semantics of the source program written in *Haffel*. Additionally, the binary logical relation can be used to prove soundness of optimisations for expressions that make use of one- or multi-shot effects.

References

- [Abr93] Samson Abramsky. “Computational interpretations of linear logic”. In: *Theoretical computer science* 111.1-2 (1993), pp. 3–57.
- [Ahm06] Amal Ahmed. “Step-indexed syntactic logical relations for recursive and quantified types”. In: *European Symposium on Programming*. Springer. 2006, pp. 69–83.
- [AM01] Andrew W Appel and David McAllester. “An indexed model of recursive types for foundational proof-carrying code”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.5 (2001), pp. 657–683.
- [App+07] Andrew W Appel et al. “A very modal model of a modern, major, general type system”. In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2007, pp. 109–122.
- [Bar84] HP Barendregt. *The Lambda Calculus, its Syntax and Semantics*. Amsterdam: North-Holland, 1984.
- [Bie+17] Dariusz Biernacki et al. “Handle with care: relational interpretation of algebraic effects and handlers”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–30.
- [BK01] Nick Benton and Andrew Kennedy. “Exceptional syntax”. In: *Journal of Functional Programming* 11.4 (2001), pp. 395–410.
- [BP15] Andrej Bauer and Matija Pretnar. “Programming with algebraic effects and handlers”. In: *Journal of logical and algebraic methods in programming* 84.1 (2015), pp. 108–123.
- [BSO20] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. “Effects as capabilities: effect handlers and lightweight effect polymorphism”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–30.
- [BV23] Casper Bach Poulsen and Cas Van Der Rest. “Hefty algebras: Modular elaboration of higher-order algebraic effects”. In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 1801–1831.
- [Car+86] Luca Cardelli et al. *Combinators and Functional Programming Languages*. 1986.
- [Con+20] Lukas Convent et al. “Doo bee doo bee doo”. In: *Journal of Functional Programming* 30 (2020), e9.
- [DAB11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. “Logical step-indexed logical relations”. In: *Logical Methods in Computer Science* 7 (2011).
- [DM02] Pietro Di Gianantonio and Marino Miculan. “A unifying approach to recursive and co-recursive definitions”. In: *International Workshop on Types for Proofs and Programs*. Springer. 2002, pp. 148–161.
- [Dol+18] Stephen Dolan et al. “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 18*. Springer. 2018, pp. 98–117.
- [FR20] Kavon Farvardin and John Reppy. “From folklore to fact: comparing implementations of stacks and continuations”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 75–90.

- [Har16] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.
- [HDB90] Robert Hieb, R Kent Dybvig, and Carl Bruggeman. “Representing control in the presence of first-class continuations”. In: *ACM SIGPLAN Notices* 25.6 (1990), pp. 66–77.
- [Hin+69] Roger Hindley et al. “The principal type-scheme of an object in combinatory logic”. In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60.
- [HL16] Daniel Hillerström and Sam Lindley. “Liberating effects with rows and handlers”. In: *Proceedings of the 1st International Workshop on Type-Driven Development*. 2016, pp. 15–27.
- [HLL20] Daniel Hillerström, Sam Lindley, and John Longley. “Effects for efficiency: Asymptotic speedup with first-class control”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–29.
- [Jun+16] Ralf Jung et al. “Higher-order ghost state”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 256–269.
- [Jun+18] Ralf Jung et al. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *Journal of Functional Programming* 28 (2018), e20.
- [KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in action”. In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 145–158.
- [Kre+18] Robbert Krebbers et al. “MoSeL: A general, extensible modal framework for interactive proofs in separation logic”. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–30.
- [KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive proofs in higher-order concurrent separation logic”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 205–217.
- [Lei14] Daan Leijen. “Koka: Programming with row polymorphic effect types”. In: *arXiv preprint arXiv:1406.2061* (2014).
- [Lei17] Daan Leijen. “Type directed compilation of row-typed algebraic effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 486–499.
- [Mil78] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4).
- [Mog88] Eugenio Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, 1988.
- [Par15] Daniel Parker. *JavaScript with Promises: Managing Asynchronous Code*. O’Reilly Media, Inc., 2015.
- [Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
- [Pie04] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2004.
- [PP01] Gordon Plotkin and John Power. “Adequacy for algebraic effects”. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2001, pp. 1–24.

- [PP02] Gordon Plotkin and John Power. “Notions of computation determine monads”. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2002, pp. 342–356.
- [PP09] Gordon Plotkin and Matija Pretnar. “Handlers of algebraic effects”. In: *European Symposium on Programming*. Springer. 2009, pp. 80–94.
- [Roo24] Orpheas van Rooij. *haffel*. <https://gitlab.science.ru.nl/ovrooij/haffel>. 2024.
- [SB14] Kasper Svendsen and Lars Birkedal. “Impredicative concurrent abstract predicates”. In: *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*. Springer. 2014, pp. 149–168.
- [Sel15] Yury Selivanov. “Pep 492–coroutines with async and await syntax”. In: *Python Org* (2015).
- [Siv+21] KC Sivaramakrishnan et al. “Retrofitting effect handlers onto OCaml”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 206–221.
- [Siv15] KC Sivaramakrishnan. *Pearls of Algebraic Effects and Handlers*. <https://kcsrk.info/ocaml/multicore/effects/2015/05/27/more-effects>. 2015.
- [SPL11] Don Syme, Tomas Petricek, and Dmitry Lomov. “The F# asynchronous programming model”. In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2011, pp. 175–189.
- [Tan+24] Wenhao Tang et al. “Soundly Handling Linearity”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (2024), pp. 1600–1628.
- [TB19] Amin Timany and Lars Birkedal. “Mechanized relational verification of concurrent programs with continuations”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–28.
- [Tim+22] Amin Timany et al. “A Logical Approach to Type Soundness”. In: *Unpublished manuscript* (2022).
- [Vil22] Paulo Emílio de Vilhena. “Proof of Programs with Effect Handlers”. Theses. Université Paris Cité, 2022. URL: <https://inria.hal.science/tel-03891381>.
- [VP20] Paulo Emílio de Vilhena and François Pottier. *Hazel*. <https://gitlab.inria.fr/cambium/hazel/>. 2020.
- [VP21] Paulo Emílio de Vilhena and François Pottier. “A separation logic for effect handlers”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–28.
- [VP23] Paulo Emílio de Vilhena and François Pottier. “A type system for effect handlers and dynamic labels”. In: *European Symposium on Programming*. Springer Nature Switzerland Cham. 2023, pp. 225–252.
- [Wad91] Philip Wadler. “Is there a use for linear logic?” In: *ACM SIGPLAN Notices* 26.9 (1991), pp. 255–273.
- [WF94] Andrew K Wright and Matthias Felleisen. “A syntactic approach to type soundness”. In: *Information and computation* 115.1 (1994), pp. 38–94.
- [Wri95] Andrew K Wright. “Simple imperative polymorphism”. In: *Lisp and symbolic computation* 8.4 (1995), pp. 343–355.

- [XL21] Ningning Xie and Daan Leijen. “Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C”. In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–30.
- [Yan+22] Zhixuan Yang et al. “Structured handling of scoped effects”. In: *European Symposium on Programming*. Springer International Publishing Cham. 2022, pp. 462–491.