Affect: An Affine Type and Effect System

ORPHEAS VAN ROOIJ, Radboud University Nijmegen, Netherlands and University of Edinburgh, UK
ROBBERT KREBBERS, Radboud University Nijmegen, Netherlands

Effect handlers form a powerful construct that can express complex programming abstractions. They are a generalisation of exception handlers, but allow resumption of the continuation from where the effect was raised. Allowing continuations to be resumed at most once (*one-shot*) or an arbitrary number of times (*multi-shot*) has far-reaching consequences. In addition to performance considerations, multi-shot effects break key rules of reasoning and thus render certain standard transformation/optimisations unsound, especially in languages with mutable references (such as OCaml 5). It is therefore desirable to statically track whether continuations are used in a one-shot or multi-shot discipline, so that a compiler could use this information to efficiently implement effect handlers and to determine what optimizations it may perform.

We address this problem by developing a type and effect system–called **Affect**–which uses affine types to track the usage of continuations. A challenge is to soundly deal with advanced programming features—such as references that store continuations and nested continuations—which are crucial to support challenging examples from the effects literature (such as *control inversion* and *cooperative concurrency*). Another challenge is to support generic type signatures of polymorphic effectful functions. We address these challenges by using and extending Rust's `Cell` type and Wadler's *use types*. To prove soundness of Affect we model types and judgements semantically via a logical relation in the Iris separation logic framework in Coq.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Type structures**; **Control primitives**.

Additional Key Words and Phrases: Effect handlers, Substructural types, Semantic typing, Iris, Coq

1 Introduction

Algebraic effects and handlers, originally devised by Plotkin, Power and Pretnar [42–44], have received increasing attention due to their broad range of applications. Algebraic effects are an effective tool to describe the semantics of programming languages, and together with handlers they form a powerful programming construct that can express complex abstractions (such as state, cooperative concurrency, backtracking, and probabilistic programming) as derived constructs [19, 32, 45]. Mainstream languages such as OCaml 5 have retrofitted effects and handlers [49], while research-oriented languages such as Eff [9], Effekt [13], Links [22, 51], Koka [37, 38] and Frank [14] take a more ubiquitous approach by centering their design around them.

A good way to gain an understanding of effect handlers is to compare them to exception handlers `try` $x \Leftarrow e_1$ `in` e_2 `unless` $E \Rightarrow \lambda y. e_3$ (using the syntax of Benton and Kennedy [10]). Evaluation begins with e_1. If no exception is raised, the result of e_1 is bound to x and e_2 is evaluated. If an exception E is raised, the exception value is bound to y and the exception branch e_3 is evaluated.

Authors' Contact Information: Orpheas van Rooij, Radboud University Nijmegen, Netherlands and University of Edinburgh, UK, orpheas.vanrooij@ed.ac.uk; Robbert Krebbers, Radboud University Nijmegen, Netherlands, mail@robbertkrebbers.nl.

An effect handler has the form $\mathsf{handle}\ e_1\ \mathsf{by}\ op\ y\ k.\ e_3\ |\ \mathsf{ret}\ x.\ e_2$, where e_1 is an effectful expression, whose result is finally bound to x in e_2. Like throwing an exception, calling an effect $\mathsf{do}\ op\ v$ in e_1 causes the handler e_3 to be evaluated, with y being bound to v. The key difference is the *continuation* k in the handler, which can be seen as a special function that continues the evaluation of e_1 from the point where the effect op was performed. The argument applied to the continuation k is a value that will take the place of the $\mathsf{do}\ op\ v$ expression, thus providing a way for the handler to communicate with the effectful program. Let us consider a simple example:

$$handle\_choice := \lambda f.\ \mathsf{handle}\ f\ ()\ \mathsf{by}\ choice\ ()\ k.\ k\ \mathsf{false}\ +\!\!+\ k\ \mathsf{true}\ |\ \mathsf{ret}\ x.\ [x]$$

$$example_1 := handle\_choice\ (\lambda\ ().\ \mathsf{let}\ r = \mathsf{ref}\ \mathsf{true}\ \mathsf{in}\ r := \mathsf{do}\ choice\ ();\ !r)$$

The effectful function $example_1$ allocates a reference r, performs the effect *choice* and stores its result in r, and finally returns the value of r. To interpret this effect, a handler $handle\_choice$ evaluates the associated continuation k with both boolean values and collects the results in a list. In this example the continuation k is bound to the function $\lambda y.\ r := y;\ !r$ and is evaluated in the context of $handle\_choice$ (deep handler semantics [23]). Resumption of $k\ \mathsf{false}$ results in $[\mathsf{false}]$, and $k\ \mathsf{true}$ results in $[\mathsf{true}]$, hence the final result is $[\mathsf{false};\mathsf{true}]$.

Problem. Continuations are characterised by the number of times they are resumed: *one-shot* continuations allow at most one call, and *multi-shot* continuations allow zero or more calls. The continuation in $example_1$ is multi-shot since it is resumed twice. Whether to allow continuations to be multi-shot has far-reaching consequences related to performance and program reasoning, especially in languages with mutable references (such as OCaml 5).

Effect handlers can be implemented in different ways, ranging from continuation passing style (CPS) translations and evidence-passing approaches, to lower-level approaches that adopt a specialised stack structure [24, 49, 61]. In the latter, as done in OCaml, a one-shot continuation can be resumed by directly restoring itself to the current stack without any copying. However, multi-shot continuations require an (expensive) copy before they are resumed—otherwise it is unsound to resume them again. Treating all continuations as multi-shot (so always copy before resumption) is an easy way to ensure soundness, but performance will suffer. The problem is exacerbated when one-shot continuations are captured by multi-shot ones, as that implicitly makes them multi-shot.

In addition, certain standard program transformations (and thus compiler optimisations) are unsound in the presence of multi-shot continuations because the rules of reasoning that justify them no longer hold [17, 52]. The key rule that is broken is that every code block entered is exited at most once. Consider a variation of the aforementioned example:

$$example_2 := handle\_choice\ (\lambda\ ().\ \mathsf{let}\ r = \mathsf{ref}\ \mathsf{true}\ \mathsf{in}\ r := \mathsf{do}\ choice\ ()\ \&\&\ !r;\ !r)$$

Instead of simply updating the reference r with the result of *choice*, we use the short-circuited logical 'and' operation $\&\&$ with the previous value of r. Notice that if we optimize this expression by propagating the reference initialization value (as r is not used in $\mathsf{do}\ choice\ ()$) and simplify the 'and' operation $\&\&$, then $example_2$ becomes exactly $example_1$:

$$\mathsf{let}\ r = \mathsf{ref}\ \mathsf{true}\ \mathsf{in}\ r := \mathsf{do}\ choice\ ()\ \&\&\ \mathsf{true};\ !r \qquad \text{Propagate value of } r$$

$$\mathsf{let}\ r = \mathsf{ref}\ \mathsf{true}\ \mathsf{in}\ r := \mathsf{do}\ choice\ ();\ !r \qquad \text{Optimize } \&\&$$

The semantics of $example_1$ and $example_2$ are different, however. In $example_2$, resumption of $k\ \mathsf{false}$ results in $[\mathsf{false}]$. But since the reference r now has value false, resumption of $k\ \mathsf{true}$ also results in $[\mathsf{false}]$ due to the $\&\&$ with $!r$. Hence the final result is $[\mathsf{false};\mathsf{false}]$, which is different from the result $[\mathsf{false};\mathsf{true}]$ of $example_1$.

For the aforementioned reasons, OCaml 5 forbids multi-shot continuations altogether by throwing a run-time error when continuations are resumed more than once.

Solution. The problems with multi-shot continuations can be solved with a substructural type system that tracks the call usage of continuations, and therefore distinguishes between effects that are handled in a one-shot and multi-shot discipline. With the call usage information, a compiler can represent continuations in the most efficient way (*e.g.,* by allowing destructive resumptions for one-shot continuations). Additionally, it can leverage the effect type information to ensure that only sound optimisations are applied to expressions that produce multi-shot effects.

In light of this observation, we propose an affine type and effect system—called **Affect**—for a language inspired by OCaml with mutable references, which distinguishes between one-shot and multi-shot continuations by treating the former as affine functions. As part of the design of Affect, we need to address a number of challenges that we outline below.

Challenge #1: References containing continuations. Prior research into combining effect and substructural type systems [25, 51, 55] has not investigated mutable references, while mainstream languages such as OCaml 5 do support them. Mutable references are important for programming patterns such as *control inversion* [17, 48] and the implementation of *cooperative concurrency* as a library [19]. In these use cases, the situation is further complicated because references do not merely store first-order data (such as integers or lists of these), they store continuations. For example, the scheduler for the cooperative concurrency library uses references to keep track of the waiting computations, which are represented as continuations.

Enforcing the one-shot discipline in the presence of references and proving any formal guarantees about them is a non-trivial task: the key purpose of references is to share data, but naively this means that one can use a reference to store a continuation and read it multiple times. To enable a sound treatment of references, we take inspiration from Rust's Cell type [47]: one can only read a value that is treated affinely (such as a continuation) from a Cell if one "moves it out" by replacing it with another value. Our key observation is that Cell can be integrated into a type and effect system, and provides the necessary expressivity for aforementioned programming patterns and libraries. To prove that our Cell-based references do not break the one-shot guarantees of continuations, we need a different approach as discussed in 'Challenge #3' below.

Challenge #2: Combining one-shot and multi-shot effects. One might expect that supporting both one- and multi-shot effects can simply be done by having two distinct signatures and have the type system restrict the continuations of one-shot signatures to be affine. As observed in prior research [51, 55], this restriction is insufficient as one additionally needs a *capturing condition* which enforces that when calling a multi-shot effect (such as do op v; e), the remaining computation (e) does not capture one-shot continuations. This poses two challenges.

First, in ML-like languages such as OCaml, sequencing is not limited to the ";"/"let" operators, but almost any operator (with arity ≥ 2) implicitly performs sequencing, and their typing rules thus need to enforce the capturing condition. Although a fine-grained call-by-value language with a single sequencing construct avoids the problem entirely [51], we show that *two-context typing judgements* [27, 28, 58] are well-suited to formalize the capturing condition for an ML-like language.

Second, similar to the typing rules, the types of polymorphic functions need to express substructural dependencies between effect and type variables to maintain the capturing condition. Multiple approaches to bounded polymorphism in substructural type systems have been explored in the literature, based on kinds and qualified types [2, 51, 54, 55], or generalised bang types $!^u$ [57]. We show that the latter—which up to our knowledge has not been studied for effect handlers—is well-suited to give generic types to common programming patterns. For example, consider the

function for iteration over a list with elements α. Depending on whether α is affine or unrestricted, different conditions arise on how often the iteration can be started or replayed. Affect can capture these conditions in a single type signature. Our approach is based on Wadler [57]'s *use types*, but extended with a novel *flip-bang* operator ¡ on effect signatures and rows, which enjoys rules that are dual to the well-known bang operator ! from linear logic.

Challenge #3: Proving type soundness. Affect combines many features to statically enforce the one-shot discipline of advanced programming patterns in an OCaml-inspired language. Particularly, Affect includes one-shot and multi-shot effects, effect rows [36–38], affine types, mutable references (inspired by Rust's `Cell` [47]), the bang and flip-bang operators, subtyping, polymorphism over types, rows and modes (inspired by Wadler [57]'s use types), and equi-recursive effects. As usual when combining features, one needs to ensure they do not interact 'badly' with each other. This can be guaranteed by a proof of type soundness for the *whole language* instead of a subset, mechanised in a proof assistant to obtain the highest level of confidence.

To make this challenge feasible, we depart from the syntactic approach to type soundness [21, 41, 60] (a.k.a. Progress and Presevation) as used in prior work on substructural effect systems [25, 51] and use a semantic approach that interprets types, effects and type judgements in the higher-order separation logic Iris [29–31, 34, 53]. This approach provides a number of benefits. First, Iris provides an abstract form of step-indexing [1, 4, 5, 20] that makes it possible to support the combination of recursive types, impredicative polymorphism and mutable references. Second, to support effects and handlers, we build on extensions of Iris for effect handlers by de Vilhena and Pottier [15–17], called Hazel and Maze. Third, Iris's machinery for impredicative invariants [50] and ghost state is crucial to interpret our `Cell`-inspired references that are unrestricted but store values of affine types. And finally, Iris provides support to mechanise all our proofs in Coq [33, 35].

Contributions and outline. We introduce **Affect**—a typed language with effect handlers in the style of OCaml 5, which statically ensures that continuations of one-shot effects are resumed at most once. After giving an overview of Affect and showcasing it on challenging examples such as control inversion [17, 48] and cooperative concurrency [19] (§ 2), we present our contributions:

- We show how Rust's `Cell` type can be integrated into a type and effect system to support the combination of one-shot effects and mutable references (§ 3).
- We show how one-shot and multi-shot effects can be combined into a single language. We provide concise typing rules that enforce the *capturing condition* using a *two-context judgement*, and extend Wadler [57]'s *use types* with a novel *flip-bang* operator ¡ on effect signatures and rows (§ 4).
- We prove type soundness of the whole Affect language using a semantic approach that interprets types and effects as semantic objects in Iris, building on the extensions of Iris for effect handlers by de Vilhena and Pottier [15–17] (§ 5).

We conclude the paper with related work (§ 6) and future work (§ 7). Our artifact includes a mechanisation of our type soundness proof in the Coq proof assistant [56].

2 Overview of Affect

This section gives an overview of the Affect language. We start by demonstrating how affine types are used to enforce a one-shot discipline of well-known one-shot effects such as exceptions and state (§ 2.1). We then show Affect's support for mutable references based on Rust's `Cell` type (§ 2.2), and Affect's support for polymorphic and recursive effects (§ 2.3). In § 2.2 and 2.3 we showcase Affect on the challenging examples of control inversion [17, 48] and cooperative concurrency [19].

We finally show how to combine one-shot and multi-shot effects (§ 2.4), as well as illustrate *mode polymorphism* by introducing Affect's novel *flip-bang* operator (§ 2.5).

2.1 Enforcing a One-Shot Discipline via Affine Types

Let us consider some well-known effects from the literature:

$$\textbf{Except } \alpha := (\textit{throw} : \forall \gamma. \, \alpha \multimap \gamma) \quad \textbf{Counter} := (\textit{inc} : \textbf{N} \multimap \textbf{N}) \quad \textbf{Choice} := (\textit{choose} : \textbf{1} \multimap \textbf{B})$$

The intended semantics is that *throw* aborts computation by throwing an exception, *inc* increments a counter by the given number and returns its previous value (*inc* is an instance of the state effect), and *choose* non-deterministically returns a boolean. These effects can be used as follows:

$$example_3 : \textbf{1} \xrightarrow{\langle \textbf{Except N,Counter} \rangle} \textbf{N} := \lambda \, (). \; \texttt{if do } inc \; 10 == 0 \; \texttt{then do } throw \; 40 \; \texttt{else do } inc \; 10$$
$$example_4 : \textbf{1} \xrightarrow{\langle \textbf{Counter,Choice} \rangle} \textbf{N} := \lambda \, (). \; \texttt{if do } choose \; () \; \texttt{then do } inc \; 10 \; \texttt{else do } inc \; 20$$

The do keyword performs an effect. Function types $\xrightarrow{\rho}$ are annotated with an *effect row* ρ [36–38], which is a list-like structure that tracks which effects are used. Effect rows only give the types of operations, their semantics is given by a handler. For example:

$$handle\_except : \forall \alpha. \forall \theta. \, (\textbf{1} \xrightarrow{\textbf{Except } \alpha \cdot \theta} \alpha) \xrightarrow{\theta} \alpha :=$$
$$\lambda f. \, \texttt{handle } f \, () \, \texttt{by } throw \, x \, (k : \gamma \xrightarrow{\theta} \alpha). \, x \mid \texttt{ret } x. \, x$$
$$handle\_counter : \forall \alpha. \forall \theta. \, (\textbf{1} \xrightarrow{\textbf{Counter} \cdot \theta} \alpha) \xrightarrow{\theta} \alpha :=$$
$$\lambda f. \, (\texttt{handle } f \, () \, \texttt{by } inc \, n \, (k : \textbf{N} \xrightarrow{\theta} \textbf{N} \xrightarrow{\theta} \alpha). \, \lambda m. \, k \, m \, (n + m) \mid \texttt{ret } x. \, \lambda m. \, x) \, 0$$

The first handler turns exceptions into an ordinary return value. The second handler threads through the state m of the counter (which is initially 0). Both handlers take a computation f, which uses respectively the **Except** and **Counter** effect, and remove this effect from the row in their return type. Similar to Koka [37, 38], the order of effects with unequal names does not matter, *e.g.,* given $example_3$, we can use *handle_except* and *handle_counter* in either order.

Both these handlers enjoy the one-shot discipline: they resume the continuation k zero and one time, respectively. Affect gives continuations the *affine function type* $\xrightarrow{\theta}$ instead of the *unrestricted function type* $\xrightarrow{\rho}$, allowing the type system to enforce at most-once usage. Hence, the following handler for choice, which resumes the continuation k twice, is ill-typed:

$$handle\_choice : \forall \alpha. \forall \theta. \, (\textbf{1} \xrightarrow{\textbf{Choice} \cdot \theta} \alpha) \xrightarrow{\theta} \textbf{List } \alpha :=$$
$$\lambda f. \, \texttt{handle } f \, () \, \texttt{by } choose \, () \, (k : \textbf{B} \xrightarrow{\theta} \textbf{List } \alpha). \, k \, \texttt{false} \mathbin{+\!\!+} k \, \texttt{true} \mid \texttt{ret } x. \, [x]$$

Recall that Affect rejects this handler for a good reason—as discussed in § 1, multi-shot continuations are less efficient and prohibit standard program optimisations. Some languages (*e.g.,* OCaml 5) even abort with a run-time error when a continuation is resumed multiple times. It is therefore desirable to *statically* ensure the one-shot discipline, but as we will show throughout this section, this becomes increasingly challenging when considering mutable references (§ 2.2), polymorphic and recursive effects (§ 2.3), and a combination of one-shot and multi-shot effects (§ 2.4, Affect supports *handle_choice*, provided *choose* is tagged as multi-shot). A key contribution of our paper is the type soundness theorem, which ensures that Affect is doing its job:

THEOREM 2.1. *Every closed well-typed program is safe w.r.t. the operational semantics.*

Crucial to this theorem is the notion of *safety* (§ 3.2). Unlike standard effect systems, safety does not only capture that operations are called with the correct arguments and all effects are handled, it also captures that continuations corresponding to one-shot effects are resumed at most once. The latter is modelled in the operational semantics using a trick by de Vilhena and Pottier [17].

2.2 Storing Continuations in References

Enforcing the one-shot discipline in the presence of references is challenging: the key purpose of references is to share data, but we need to ensure that continuations in references are resumed at most once to maintain type soundness (Theorem 2.1). To achieve that, we take inspiration from Rust's Cell type [47]: Affect's references **Ref** τ correspond to &Cell<τ>, and are thus unrestricted. Depending on whether τ is unrestricted or affine, different operations can be performed on **Ref** τ:

- The store operation ($e_1 \coloneqq e_2$) can be used for any type τ.
- The load operation ($!e$) is limited to unrestricted types τ such as basic types (unit, booleans, natural numbers) and unrestricted functions. This condition is crucial—otherwise one can put a continuation in a reference, load it multiple times, and break the one-shot discipline.
- The replace operation (replace $e_1 e_2$), which stores e_2 in e_1 and returns the previously stored value of e_1, can be used for any type τ.

To see references in action, we consider two interfaces to traverse a data structure—iterators and generators (also called sequences or cascades)—and convert these into each other:

$$\textbf{Iter } \alpha \coloneqq \forall \theta. (\alpha \xrightarrow{\theta} \mathbf{1}) \xrightarrow{\theta} \mathbf{1} \qquad\qquad \textbf{Gen } \alpha \coloneqq \mathbf{1} \to \textbf{Option } \alpha$$

(Note the quantification over the effect row θ in **Iter**, allowing a client to use an iterator i with an effectful function f, so that $i \, f$ has the same effects as f.) It is straightforward to define iterators and generators for lists. The function $list2iter : \textbf{List } \alpha \to \textbf{Iter } \alpha$ simply calls the function $f : \alpha \xrightarrow{\theta} \mathbf{1}$ on each element of the list. The function $list2gen : \textbf{List } \alpha \to \textbf{Gen } \alpha$ stores the list in a local reference, and returns a function $\mathbf{1} \to \textbf{Option } \alpha$ (Affect's unrestricted references are needed here). This function checks if the list is empty and returns none. Otherwise it returns the head (as some) and updates the local reference to the tail. Converting a generator into an iterator is also easy. The function $gen2iter : \textbf{Gen } \alpha \to \textbf{Iter } \alpha$ is implemented by repeatedly calling the generator and calling the iteration function on each generated element.

Converting an iterator into a generator is more challenging. To preserve the lazy behaviour (which is necessary for infinite data structures), the conversion can be done using *control inversion* [17, 48], which uses the combination of references *and* effect handlers. Let us first consider the simpler case of using an iterator to retrieve the first element:

$iter2first : \textbf{Iter } \alpha \to \textbf{Option } \alpha \coloneqq$
  $\qquad \lambda \, i. \, handle\_except \, (\lambda \, (). \, i \, \langle \textbf{Except } (\textbf{Option } \alpha) \rangle \, (\lambda \, x. \, \text{do } throw \, (\text{some } x)); \, \text{none})$

We use the **Except** effect to throw an exception after the first iteration. The handler $handle\_except$ discards the continuation, aborting the iteration, and returning the first value.

To implement $iter2gen : \textbf{Iter } \alpha \to \textbf{Gen } \alpha$ we use the power of effect handlers to resume the continuation and continue the iteration. Similar to the way $list2gen$ stores the remaining list in a reference, we now store the continuation in a reference. The code when written in Affect is:

$iter2gen : \forall \alpha. \, \textbf{Iter } \alpha \to \textbf{Gen } \alpha \coloneqq$
  $\qquad \lambda \, i. \, \text{let } (r : \textbf{Ref } (\mathbf{1} \multimap \textbf{Option } \alpha)) = \text{ref } (\lambda \, (). \, \text{none}) \text{ in}$
  $\qquad\quad r \coloneqq (\lambda \, (). \, \text{handle } i \, \langle \textbf{Yield } \alpha \rangle \, (\lambda \, x. \, \text{do } yield \, x) \text{ by}$
  $\qquad\qquad\qquad yield \, x \, (k : \mathbf{1} \multimap \textbf{Option } \alpha). \, r \coloneqq k; \, \text{some } x$
  $\qquad\qquad\quad | \, \text{ret } x. \, \text{none});$
  $\qquad\quad \lambda \, (). \, \text{replace } r \, (\lambda \, (). \, \text{none}) \, ()$

We use the effect **Yield** $\alpha \coloneqq \langle yield : \alpha \rightarrowtail \mathbf{1} \rangle$. Since the local reference r contains an affine value we cannot simply read it but can only replace it with something else. Thus the reference r alternates between two states, it either has the dummy value ($\lambda \, (). \, \text{none}$) or stores a meaningful computation.

The meaningful computation at the start is $(\lambda\,()\,.$ handle $\ldots)$. While the generator is executing the content of the reference r is replaced with the dummy value, and as soon as the iteration is paused (by a call to *yield*) we store the iteration's resumption.

The version of *iter2gen* in Affect is only a bit more verbose than the version in OCaml 5 [48] due to the use of replace instead of an ordinary load operation. The benefit, however, is that Affect enforces the one-shot discipline statically through type checking, whereas OCaml 5 would abort the program if the one-shot discipline is violated.

2.3 Polymorphic and Recursive Effects

We now present another prominent example from the literature on effect handlers, cooperative concurrency implemented as a library [19]. This example again uses references, but brings additional challenges w.r.t. polymorphism and recursive effects. A library for cooperative concurrency in Affect provides the following functions:[1]

$$async : \forall\theta.\,\forall\alpha.\,(1 \xrightarrow{\textbf{Coop }\theta} !\alpha) \xrightarrow{\textbf{Coop }\theta} \textbf{Promise }\theta\,!\alpha$$

$$await : \forall\theta.\,\forall\alpha.\,\textbf{Promise }\theta\,!\alpha \xrightarrow{\textbf{Coop }\theta} !\alpha$$

$$handle\_coop : \forall\theta.\,\forall\alpha.\,(1 \xrightarrow{\textbf{Coop }\theta} !\alpha) \xrightarrow{\theta} !\alpha$$

The types of these functions are quite a mouthful, so let us ignore the bang operator (!) and effect row (θ) at first. The function *async* 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 *await* we can perform a blocking wait on a promise to signal that we depend on the result of the asynchronous computation to continue evaluation. The handler *handle_coop* takes an effectful computation and returns its result. It internally keeps track of a queue of computations and schedules these accordingly. Similar to the handlers in § 2.1, *handle_coop* removes the handled effect **Coop** from the effect row.

The type **Promise** and the effect row **Coop** are parameterised by the remaining effects θ. This is necessary because promises internally store computations, whose effects we need to track. In the implementation of the **Coop** library, a promise is defined as a reference to either a finished computation, or a list of computations that depend on its result:

$$\textbf{Promise }\theta\,\alpha := \textbf{Ref}\,(\alpha + \textbf{List}\,(\alpha \xrightarrow{\theta} 1)) \qquad\qquad \text{Multi}\,(\textbf{Promise }\theta\,\tau)$$

Finally, let us explain the curious *bang* operator (!). By virtue of being references, Promises are unrestricted regardless of τ (Multi constraint), a key property needed by the library's implementation. To maintain the one-shot discipline, we need to prevent calling *async* with a computation that results in a one-shot continuation, and then use *await* to retrieve that continuation multiple times. Using the bang we restrict promises to unrestricted types.

The bang $!\tau$ describes values of type τ that do not capture any continuations, and its key rules are part of the subtyping relation. The elimination rule $!\tau \lessdot \tau$ holds for any type τ, while the introduction rule $\tau \lessdot !\tau$ is limited to unrestricted types. The bang makes it possible to describe functions that are polymorphic in an unrestricted type as $\forall\alpha.\,\tau$, where every occurrence of α in τ is below a bang.

To define the effect row **Coop** we need one more feature—recursive effect signatures:

$$\textbf{Coop }\theta := \mu\theta'.\,(async : \forall\alpha.\,(1 \xrightarrow{\theta'} !\alpha) \multimap \textbf{Promise }\theta\,!\alpha) \cdot (await : \forall\alpha.\,\textbf{Promise }\theta\,!\alpha \multimap !\alpha) \cdot \theta$$

Effect signatures in Affect are equi-recursive, giving us the equality:

$$\textbf{Coop }\theta = (async : \forall\alpha.\,(1 \xrightarrow{\textbf{Coop }\theta} !\alpha) \multimap \textbf{Promise }\theta\,!\alpha) \cdot (await : \forall\alpha.\,\textbf{Promise }\theta\,!\alpha \multimap !\alpha) \cdot \theta$$

---

[1]For simplicity, we omit the *yield* function, which gives control to the next scheduled computation. It could either be defined as *yield* $:= \lambda\,()\,.\,await\,(async\,(\lambda\,()\,.\,()))$ or added as a primitive.

The input of *async* is a computation $1 \xrightarrow{\textbf{Coop } \theta} !\alpha$ that can perform effects **Coop** θ, making it possible to have asynchronous computations that call *async* itself ($async\ (\lambda\ ().\ \ldots async\ldots)$). The computation passed to *async* can also perform effects θ, so the $\mu\theta'$ binder must capture the tail θ. Finally, we use polymorphic effect signatures ($\forall\alpha$) for *async* and *await* to allow the user to start asynchronous computations of different *unrestricted* types, instead of requiring them to all have the same type.

We should emphasize that the operations *async* and *await* have a one-shot signature. Hence the Affect type system ensures that the corresponding continuations are resumed at most once, while still allowing them to be stored in the references that represent promises.

2.4 Combining One-Shot and Multi-Shot Effects

Affect combines one-shot and multi-shot effects in the same language by tagging effects with a *mode* $m \in$ Mode, which is either *at most once* (O) or *zero or more times* (M). Effects such as **Except**, **Counter** and **Coop** are tagged O, while **Choice** is tagged M. A compiler could use the modes to implement effects in an efficient (one-shot) or less efficient (multi-shot) manner.

At first glance, one might think this simply achieved by adapting the typing rule of handlers to ensure that the continuation is affine if the effect is one-shot, and unrestricted if the effect is multi-shot. As observed in prior research [51, 55], this restriction alone is insufficient to maintain type soundness (Theorem 2.1). Let us reconsider the choice effect (§ 2.1) and change $\Rightarrow$ into $\Rightarrow$ to make its signature multi-shot, *i.e.,* we redefine **Choice** := ($choose : 1 \Rightarrow \textbf{B}$). Now consider:

$$handle\_choice\ (\lambda\ ().\ \textsf{do}\ choose\ ();\ k\ ())$$

Here, k is an affine function (*e.g.,* a continuation of a one-shot effect). Since the handler of *choose* (given in § 2.1) resumes its continuation twice, the function k is also resumed twice. So this program should be rejected by the Affect type system.

The *capturing condition* we should enforce is as follows: when performing an effect $\textsf{do}\ op\ e_1;\ e_2$, either the effect op is one-shot, or the effect op is multi-shot and the remainder of the computation e_2 is unrestricted (*i.e.,* e_2 does not capture any affine variables). Since Affect is an ML-style functional language, constructs such as application and binary arithmetic operators implicitly perform sequencing, so their corresponding typing rules need to incorporate this condition. We leave the design of our typing rules using a two-context typing judgement for § 4.3, and now highlight how the combination of one-shot and multi-shot effects impacts polymorphic functions.

2.5 Mode Polymorphism

Consider a higher-order identity function $g := \lambda f\ x.\ f\ ();\ x$ that calls the function f before returning the argument x. We would like to give this function a polymorphic type over the effect of f and type of argument x. A naive attempt is as follows:

$$\forall\theta.\ \forall\alpha.\ (1 \xrightarrow{\theta} 1) \to \alpha \xrightarrow{\theta} \alpha$$

Unfortunately, this type is unsound, as we can adapt the counterexample from § 2.4:

$$handle\_choice\ (\lambda\ ().\ g\ (\lambda\ ().\ \textsf{do}\ choose\ ();\ ())\ k\ ())$$

To give a sound type to our function g we need a similar capturing condition as we use for sequencing: either θ solely contains one-shot effects, or θ contains a multi-shot effect and the type α is unrestricted. These conditions can be described by the following two types:

$$\forall\theta.\ \forall\alpha.\ (1 \xrightarrow{\mathrm{i}\theta} 1) \to \alpha \xrightarrow{\mathrm{i}\theta} \alpha \qquad\qquad \forall\theta.\ \forall\alpha.\ (1 \xrightarrow{\theta} 1) \to !\alpha \xrightarrow{\theta} !\alpha$$

The second type uses the pattern we have seen in § 2.3: using the bang (!) we limit the polymorphism to unrestricted types α. The first type uses our novel *flip-bang* operator (i) on the effect row θ,

which limits the polymorphism to one-shot effects. The flip-bang enjoys dual rules compared to the bang. Particularly, the introduction rule $\rho \lessdot \mathsf{i}\rho$ holds for any effect row ρ, making it possible to handle multi-shot effects with a one-shot handler. The elimination rule $\mathsf{i}\rho \lessdot \rho$ only holds for effect rows ρ that solely contain one-shot effects.

Since none of the two aforementioned types are an instance of the other, we wish to write a more general type. Affect allows this using *mode polymorphism*, inspired by Wadler [57]'s *use types*:

$$\forall \nu\, \nu'.\, \forall \theta.\, \forall \alpha.\, (\mathbf{1} \xrightarrow{\mathsf{i}_\nu \theta}_{\nu'} \mathbf{1}) \rightarrow {!}_\nu\, \alpha \xrightarrow{\mathsf{i}_\nu \theta}_{\nu'} {!}_\nu\, \alpha$$

The meaning of the mode-indexed bang is ${!}_{\mathrm{O}}\, \tau := \tau$ and ${!}_{\mathrm{M}}\, \tau := {!}\tau$, and dually the meaning of the mode-indexed flip-bang is $\mathsf{i}_{\mathrm{O}}\, \rho := \mathsf{i}\rho$ and $\mathsf{i}_{\mathrm{M}}\, \rho := \rho$. We let $\tau \xrightarrow{\rho}_m \kappa$ be sugar for ${!}_m\, (\tau \xrightarrow{\rho} \kappa)$. Using ν' we state that a partial application $g\, f$ for some function f can be used at most as much as f can.

Finally, we circle back to the iterator example from § 2.2. In the presence of multi-shot effects, the type of *list2iter* is unsound for a similar reason that the type of g is unsound:

handle_choice $\big(\lambda\, ().\, \textit{list2iter}\, [\texttt{inl}\, (); \texttt{inr}\, k]\, (\texttt{function inl}\, ().\, \texttt{do}\, \textit{choose}\, (); () \mid \texttt{inr}\, k'.\, k'\, ())\big)$

Here, k is an affine function, which is resumed twice due to the use of *choose* in the first iteration. We give a correct type using mode polymorphism:

$$\mathbf{Iter}\, \nu\, \alpha := \forall \theta.\, ({!}_\nu\, \alpha \xrightarrow{\mathsf{i}_\nu \theta} \mathbf{1}) \xrightarrow{\mathsf{i}_\nu \theta}_\nu \mathbf{1} \qquad \textit{list2iter} : \forall \nu.\, \forall \alpha.\, \mathbf{List}\, ({!}_\nu\, \alpha) \rightarrow \mathbf{Iter}\, \nu\, \alpha$$

The type $\mathbf{Iter}\, \mathrm{O}\, \alpha$ allows the iteration to be started and resumed at most once, and its value α can be used at most once, while $\mathbf{Iter}\, \mathrm{M}\, \alpha$ allows the iteration to be started and resumed multiple times, and its value α can also be used more than once.

3 One-Shot Affect Language

We describe Affect in stages, starting with Affect$_{\mathrm{OS}}$—a minimal language that only allows only one-shot effects. Affect$_{\mathrm{OS}}$ is still expressive enough to type the control inversion (§ 2.2) and cooperative concurrency (§ 2.3) examples. We describe the syntax and operational semantics (§ 3.1), type soundness statement (§ 3.2), types and type judgements (§ 3.3), and finally the subtyping (§ 3.4) and typing (§ 3.5) rules. In § 4 we present the full Affect language with multi-shot effects.

3.1 Syntax and Operational Semantics

Affect is an OCaml-inspired call-by-value λ-calculus based on the Hazel language of de Vilhena and Pottier [16, 17]. The syntax of expressions is shown in Fig. 1. It supports booleans literals and boolean eliminator (`if` e `then` e `else` e), mutable state in the form of ML-style references (`ref` e for allocation, `!`e for loading, and `replace` e e for replacing a reference's contents), `do` op e to call an effect, and `handle` e `by` op x $k.\, e$ | `ret` $x.\, e$ to handle an effect (the continuation k persists in the handler, *i.e.,* we use deep handlers [23]). Sequencing $e_1; e_2$ is sugar for `let _ =` e_1 `in` e_2, and the store $e_1 \coloneqq e_2$ for `replace` e_1 e_2; (). The version in Coq also supports numbers, sums, products and iso-recursive types, which we omit for brevity.

Locations ℓ, run-time effects `eff` op v N, and continuation values `cont` ℓ N are dynamic expressions, meaning they only play a role in the operational semantics. Users of Affect can only use static expressions to write programs (*i.e.,* there are no typing rules for dynamic expressions).

The operational semantics of Affect is inspired by the deep handler semantics of Kammar et al. [32]. It distinguishes between neutral contexts $N1$ that do not span through handlers, and general evaluation contexts $K1$ that can. The reduction relation is defined on configurations $e\, /\, \sigma$, where the heap σ is a finite map that associates locations ℓ with values v. The semantics is given using a

Expressions (dynamic constructs in gray)

$e \in$ Expr ::= $x \mid \lambda x. e \mid e\ e \mid$ let $x = e$ in $e \mid$
true $\mid$ false $\mid$ if e then e else $e \mid$
$(\ell \in$ Loc$) \mid$ ref $e \mid\ !e \mid$ replace $e\ e \mid$
do $op\ e \mid$ eff $op\ v\ K \mid$ cont $\ell\ K \mid$
(handle e by $op\ x\ k.\ e \mid$ ret $x.\ e$)

Values and contexts

$v \in$ Val ::= $\lambda x. e \mid$ true $\mid$ false $\mid \ell \mid$ cont $\ell\ K$
$N1 \in$ NeutralCtx ::= $e\ [] \mid [\ ]\ v \mid$ let $x = [\ ]$ in $e \mid$
if $[\ ]$ then e else $e \mid$ ref $[\ ] \mid\ ![\ ] \mid$
replace $e\ [] \mid$ replace $[\ ]\ v \mid$ do $op\ []$
$K1 \in$ EvalCtx ::= $N1 \mid$ (handle $[\ ]$ by $op\ x\ k.\ e \mid$ ret $x.\ e$)
$N \in$ NeutralCtx* ::= $[\ ] \mid N1[N]$ $K \in$ EvalCtx* ::= $[\ ] \mid K1[K]$

Selected head reduction rules

$$\text{Red-Do} \ \frac{\text{do } op\ v\ /\ \sigma}{\text{eff } op\ v\ [\ ]\ /\ \sigma}_\text{h} \qquad \text{Red-Eff} \ \frac{N1[\text{eff } op\ v\ K]\ /\ \sigma}{\text{eff } op\ v\ N1[K]\ /\ \sigma}_\text{h} \qquad \text{Red-Cont} \ \frac{(\text{cont } \ell\ K)\ v\ /\ \sigma[\ell \mapsto \text{false}]}{K[v]\ /\ \sigma[\ell \mapsto \text{true}]}_\text{h}$$

$$\text{Red-Handler} \ \frac{\text{handle } (\text{eff } op\ v\ K)\ \text{by } op\ x\ k.\ h \mid \text{ret } x.\ r\ /\ \sigma \qquad \ell \notin Dom\ \sigma}{h[v/x, (\lambda x.\ \text{handle } (\text{cont } \ell\ K)\ x\ \text{by } op\ x\ k.\ h \mid \text{ret } x.\ r)/k]\ /\ \sigma[\ell \mapsto \text{false}]}_\text{h}$$

$$\text{Red-Handler-Neq} \ \frac{\text{handle } (\text{eff } op\ v\ K)\ \text{by } op'\ x\ k.\ h \mid \text{ret } x.\ r\ /\ \sigma \qquad op \neq op'}{\text{eff } op\ v\ (\text{handle } K\ \text{by } op'\ x\ k.\ h \mid \text{ret } x.\ r)\ /\ \sigma}_\text{h}$$

Fig. 1. Syntax and selected reduction rules of Affect$_{\text{OS}}$.

head reduction $\rightarrow_\text{h}$, which is extended to full-program reduction $\rightarrow$ using evaluation contexts:

$$\frac{e\ /\ \sigma \rightarrow_\text{h} e'\ /\ \sigma'}{K[e]\ /\ \sigma \rightarrow K[e']\ /\ \sigma'}$$

Selected rules of the head reduction relation are shown in Fig. 1. The reduction rules for the λ-calculus fragment, booleans and references are standard, and thus omitted.

Red-Do evaluates an effect call do $op\ v$ to the run-time effect eff $op\ v\ []$, which will swallow its surrounding context up to the enclosing handler. This is achieved using repeated applications of Red-Eff, which transfers its surrounding neutral context $N1$ to the effect expression eff $op\ v\ N1[K]$. When the run-time effect reaches a handler that matches its operation op (Red-Handler), the effect branch is taken with the effect value v and the reified version of K as a one-shot continuation substituted in. We use a deep handler semantics so the resumption is wrapped with the same handler. If the operation in the handler does not match (Red-Handler-Neq), the handler is swallowed into the captured context. We use de Vilhena and Pottier [17]'s trick to ensure that resuming a one-shot continuation more than once results in a stuck expression: a location ℓ is associated with continuation values cont $\ell\ N$, and the heap is abused to track if the continuation has been resumed. Application of a one-shot continuation (Red-Cont) is only defined when it has not been called yet (when its location ℓ points to false).

3.2 Type Soundness

There are three guarantees that the type and effect system of Affect should enforce. Firstly, one-shot continuations (*i.e.,* values cont $\ell\ N$) should not be resumed more than once. Secondly, expressions that are annotated as non-effectful should not produce unhandled effects, and lastly no regular type errors such as applying true to itself can occur.

All three guarantees are modeled using the notion of expression safety. An expression is considered *safe* when starting from an empty heap it always leads to a configuration $e'\ /\ \sigma'$ that is not

stuck, in the sense that e' is not a value and no reduction rule applies:

$$\textit{safe } e := \forall e' \, \sigma'. \, e \; / \; \varnothing \rightarrow^* e' \; / \; \sigma' \implies e' \in \mathsf{Val} \vee (\exists e'' \, \sigma''. \, e' \; / \; \sigma' \rightarrow e'' \; / \; \sigma'')$$

A call to an already resumed one-shot continuation is a stuck expression since rule RED-DO is only defined when $\sigma[\ell \mapsto \mathtt{false}]$. Additionally, safety implies that no effects are left unhandled because run-time effect expressions $\mathtt{eff} \; op \; v \; N$ are not values and no reduction rule applies to them when they are at the top level. Lastly, regular type errors are ruled out by the very definition of safety, which states the expression will either reach a value or diverge.

Using this notion of safety, we define type soundness to mean that well-typed programs are safe. More specifically, each non-effectful expression (row $\langle \rangle$) that is typed in the empty context is safe:

$$\vdash e \; : \; \langle \rangle \; : \; \tau \dashv \implies \textit{safe } e$$

In the next subsections we introduce the types and typing judgesments of Affect$_{\mathrm{OS}}$.

3.3 Types and Type Judgements

The types, signatures and rows of Affect$_{\mathrm{OS}}$ are defined as follows:

$$\tau, \kappa, \iota \in \mathsf{Type} ::= \alpha \mid \mathbf{B} \mid !\tau \mid \forall \alpha. \, \tau \mid \forall \theta. \, \tau \mid \tau \xrightarrow{\rho} \tau \mid \mathbf{Ref} \, \tau$$

$$\sigma, \hat{\sigma} \in \mathsf{Signature} ::= \forall \vec{\alpha}. \, \tau \Rrightarrow \kappa$$

$$\rho, \hat{\rho} \in \mathsf{Row} ::= \theta \mid \langle \rangle \mid (op : \sigma) \cdot \rho \mid \mu\theta. \, \rho$$

Types are either type variables α that range over a countably infinite set, boolean types $\mathbf{B}$, bang types $!\tau$ that denote unrestricted types, type and row polymorphic types $\forall \alpha. \, \tau$ and $\forall \theta. \, \tau$, affine function types $\tau \xrightarrow{\rho} \tau$, or reference types $\mathbf{Ref} \, \tau$. Function types are tagged with an effect row ρ to track their effects. Unrestricted functions $\tau \xrightarrow{\rho} \kappa$ are defined as $!(\tau \xrightarrow{\rho} \kappa)$. We drop the empty row $\langle \rangle$ from non-effectful functions, *i.e.,* we write $\tau \multimap \kappa$ and $\tau \rightarrow \kappa$.

Effect signatures $\forall \vec{\alpha}. \, \tau \Rrightarrow \kappa$ give the type specification of effect operators. They can be polymorphic over type variables $\vec{\alpha}$, which are instantiated at the effect call site.

Effect rows group multiple effect signatures together into a list-like structure ($\langle \rangle$ is the empty row, $(op : \sigma) \cdot \rho$ is the cons row). Similar to Koka [36–38], the order of effects in a row is irrelevant up to distinct operations. Thus the effect row $\langle op : \sigma, op : \sigma' \rangle$ that has two effects with the same operation op is distinguished from its reversed version $\langle op : \sigma', op : \sigma \rangle$. Rows can be polymorphic over a single row variable that can only appear at the end. Effect rows that end with a row variable are referred to as *open*, whereas *closed* rows end with the empty row. We often use list notations such as $\langle op : \sigma \rangle$ and $\langle op : \sigma, op' : \sigma' \rangle$ for closed effect rows.

The equi-recursive row $\mu\theta. \, \rho$ allows the effect row ρ to reference itself via the θ variable. To ensure effect rows are finite, there is a well-formedness restriction, which states that the recursive binder θ should only be used in the effect signatures present in ρ. Effect rows that utilise the recursive θ binder in the spine of the effect row such as $\mu\theta. \, \theta$ and $\mu\theta. \, (op : \sigma) \cdot \theta$ are ill-formed.

Typing judgements $\Gamma_1 \vdash e \; : \; \rho \; : \; \tau \dashv \Gamma_2$ take two typing contexts, where Γ_1 and Γ_2 are called *initial* and *final*, respectively. Contexts are multisets, *i.e.,* the multiplicity is relevant, but the order is not. As opposed to the more conventional two-context algorithmic typing judgments where Γ_2 only tracks the variables that have not been used [58], in Affect the final context tracks which variables from Γ_1 *can* be used after evaluation of e. In the case that e produces an effect, the final context gives the possible variables from Γ_1 that *can* be captured in the continuation. This key property is essential when we incorporate multi-shot effects as it allows us to enforce the *capturing condition* which restricts affine values from being captured in multi-shot continuations § 4.

To make the system substructural we restrict CONTRACTION to variables with unrestricted types, *i.e.,* that can be used more than once (captured by the Multi constraint, defined in § 3.4). WEAKENING

holds for all types because affine (instead of linear) usage of resources is enforced. Contraction and weakening can also be performed on the final context Γ_2 using similar rules as those above.

$$\frac{x : \kappa, x : \kappa, \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2 \qquad \text{Multi } \kappa}{x : \kappa, \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2}$$

CONTRACTION

WEAKENING

$$\frac{\Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2}{x : \kappa, \Gamma_1 \vdash e : \rho : \tau \dashv \Gamma_2}$$

One-shot effects in Affect$_{OS}$ are *affine* instead of *linear*, *i.e.,* their continuations should be resumed *at most* once instead of *exactly* once. Affine effects are necessary to support **Except** from § 2.1 (whose continuation is not resumed) without the need to explicitly 'discontinue'/'free' the continuation.

3.4 Subtyping of Types, Signatures and Rows

The rules for the subtyping relations <: on types, signatures and rows are shown in Figure 2. We omit the standard rules for reflexivity and transitivity. We use <:> to denote that subtyping applies both ways. Function types follow a standard subtyping rule which is contravariant in the argument and covariant in the result (FUN), while effect signatures instead are covariant on argument types and contravariant on result types (SIG).

Introduction of bang types $!\tau$ can only happen to types that are globally treated as unrestricted through rules BANG-INTRO-BOOL, BANG-INTRO-REF and BANG-IDEMP. The Multi constraint represents unrestricted types and is defined accordingly as:

$$\text{Multi } \tau := \tau <: !\tau$$

Similar to Rust's &Cell<τ>, reference types **Ref** τ are treated unrestrictedly regardless of the inner type τ. This allows us to store one-shot continuations in references allowing them to be shared between different expressions, a crucial property for the examples in § 2.2 and 2.3.

Rules BANG-ALLT-COMM and BANG-ALLR-COMM show that bang types commute with polymorphic types. Type polymorphic types can also be subtyped pointwise via TYPE-FORALL by requiring that the subtyping holds for any α (α is free in $\tau <: \kappa$ and we assume the Barendregt [7] variable convention). Row polymorphic types can also be subtyped in the same way via ROW-FORALL.

Subtyping on rows can extend the empty row $\langle \rangle$ via NIL, and subtype a cons row component wise via CONS. The ordering of operations is irrelevant for two distinct operations as witnessed by SWAP. Row extension can only happen for closed rows to ensure that the order of duplicate effects is respected. The equi-recursive nature of effect rows allows unfolding and folding through the subtyping rules (UNFOLD and FOLD).

3.5 Typing Rules

Figure 3 shows selected typing rules of Affect$_{OS}$. We introduce a single context *value typing judgement* $\Gamma_1 \vdash_v e : \tau$ to represent well-typed values which are also well-typed expressions as witnessed by VALTYPED. Observe that we can simply forward the final context in VALTYPED, since Γ_2 gives the variables available *after* evaluation of e and values are not evaluated. Through this judgement we restrict polymorphism to values, akin to the value restriction [59]. For instance, to introduce an effect polymorphic type, ROWINTRO requires a value typing judgement which must eventually lead to an abstraction via ABS. The bang type introduction rule BANGINTRO also requires a value typing judgement since a full promotion rule that is applied to any expression is unsound.

Rule SUB utilises the subtyping relations on types, effect rows and contexts. The subtyping relation on contexts, $\Gamma' <: \Gamma$, extends the relation on types component wise, by requiring every variable $x : \tau \in \Gamma$ to have a corresponding $x : \tau' \in \Gamma'$ such that $\tau' <: \tau$. The typing judgement is contravariant on the initial context and covariant on the final one.

$$\boxed{\tau <: \kappa} \quad \boxed{\sigma <: \hat{\sigma}} \quad \boxed{\rho <: \hat{\rho}}$$

Fun
$$\frac{\tau <: \tau' \qquad \hat{\rho} <: \rho \qquad \kappa' <: \kappa}{\tau' \xrightarrow{\hat{\rho}} \kappa' <: \tau \xrightarrow{\rho} \kappa}$$

Sig
$$\frac{\tau' <: \tau \qquad \kappa <: \kappa'}{\forall \vec{\alpha}.\, \tau' \Rightarrow \kappa' <: \forall \vec{\alpha}.\, \tau \Rightarrow \kappa}$$

Bang-Intro-Bool
$$\mathbf{B} <: !\mathbf{B}$$

Bang-Intro-Ref
$$\mathbf{Ref}\,\tau <: !(\mathbf{Ref}\,\tau)$$

Bang-Idemp
$$!\tau <:> !!\tau$$

Bang-Elim
$$!\tau <: \tau$$

Bang-Comp
$$\frac{\tau <: \kappa}{!\tau <: !\kappa}$$

Bang-AllT-Comm
$$!\forall \alpha.\, \tau <:> \forall \alpha.\, !\tau$$

Bang-AllR-Comm
$$!\forall \theta.\, \tau <:> \forall \theta.\, !\tau$$

Type-Forall
$$\frac{\tau <: \kappa}{\forall \alpha.\, \tau <: \forall \alpha.\, \kappa}$$

Row-Forall
$$\frac{\tau <: \kappa}{\forall \theta.\, \tau <: \forall \theta.\, \kappa}$$

Nil
$$\langle\rangle <: \rho$$

Cons
$$\frac{\sigma' <: \sigma \qquad \hat{\rho} <: \rho}{(op : \sigma') \cdot \hat{\rho} <: (op : \sigma) \cdot \rho}$$

Swap
$$\frac{op \neq op'}{(op : \sigma) \cdot (op' : \sigma') \cdot \rho <: (op' : \sigma') \cdot (op : \sigma) \cdot \rho}$$

Unfold
$$\mu\theta.\, \rho <: \rho[\mu\theta.\, \rho/\theta]$$

Fold
$$\rho[\mu\theta.\, \rho/\theta] <: \mu\theta.\, \rho$$

Fig. 2. Subtyping rules of Affect$_{OS}$.

The call-by-value evaluation strategy of Affect means that variables are substituted with values and so variable expressions are non-effectful (Var). Affine variables in the context must be used at most once, and thus x does not appear in the final context. When τ is unrestricted, we can use Contraction to allow more uses of x.

The typing rule for the function type Abs requires that the body is typed with an empty final context since any final context that is available after evaluation of e would not be available at $\lambda x.\, e$ (we do not evaluate under abstractions).

Rule App shows the typing of applications. The expressions e_1 and e_2 must conform to the same effect row ρ as the function type. This is not a limitation since Sub can be applied to subtype compatible rows into ρ. The initial context Γ_1 is also threaded between sub-derivations according to the (right to left) evaluation order. The Let rule follows a similar pattern.

In IfElse all sub-derivations must adhere to the same effect row just as in App, and the context is passed along according to the evaluation order. Since only one of the two expressions e_2 and e_3 will be evaluated, the context Γ_2 can be shared between them.

Rules RowIntro and RowElim introduce and eliminate effect polymorphic types. RowIntro requires the expression e to be well-typed with type τ for all θ (row variable θ is free in τ). The typing rules for type polymorphism follow a similar principle and are omitted.

Reference allocation has a standard typing rule Alloc with the context and effect row depending on the inner expression. The Read rule is for reading from a reference and requires the inner type τ to be unrestricted, similar to Cell::get in Rust. Rule Replace has as resulting type τ which represents the previous value stored in the reference, similar to Cell::replace in Rust.

The Do rule describes the typing of an effect call. The type of the argument passed to the effect call must be τ. The effect handler can only respond with values of type κ, which will take the place of the do $op\ e$ expression. The signature is polymorphic over type variables $\vec{\alpha}$, and so the argument

$$\boxed{\Gamma_1 \vdash_v e \,:\, \tau} \quad \boxed{\Gamma_1 \vdash e \,:\, \rho \,:\, \tau \dashv \Gamma_2}$$

VALTYPED

$$\frac{\Gamma_1 \vdash_v e \,:\, \tau}{\Gamma_1; \Gamma_2 \vdash e \,:\, \langle\rangle \,:\, \tau \dashv \Gamma_2}$$

SUB

$$\frac{\Gamma_1' \vdash e \,:\, \hat{\rho} \,:\, \kappa \dashv \Gamma_2' \qquad \Gamma_1 \lessdot \Gamma_1' \qquad \Gamma_2' \lessdot \Gamma_2 \qquad \hat{\rho} \lessdot \rho \qquad \kappa \lessdot \tau}{\Gamma_1 \vdash e \,:\, \rho \,:\, \tau \dashv \Gamma_2}$$

BANGINTRO

$$\frac{\text{Multi}\,\Gamma_1 \qquad \Gamma_1 \vdash_v e \,:\, \tau}{\Gamma_1 \vdash_v e \,:\, !\tau}$$

VAR

$$\frac{}{x : \tau, \Gamma \vdash x \,:\, \langle\rangle \,:\, \tau \dashv \Gamma}$$

ABS

$$\frac{x : \tau, \Gamma_1 \vdash e \,:\, \rho \,:\, \kappa \dashv}{\Gamma_1 \vdash_v \lambda x. e \,:\, \tau \xrightarrow{\rho} \kappa}$$

APP

$$\frac{\Gamma_1 \vdash e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_1 \,:\, \rho \,:\, \tau \xrightarrow{\rho} \kappa \dashv \Gamma_3}{\Gamma_1 \vdash e_1 \, e_2 \,:\, \rho \,:\, \kappa \dashv \Gamma_3}$$

LET

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

IFELSE

$$\frac{\Gamma_1 \vdash e_1 \,:\, \rho \,:\, \mathbf{B} \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_3 \qquad \Gamma_2 \vdash e_3 \,:\, \rho \,:\, \tau \dashv \Gamma_3}{\Gamma_1 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \,:\, \rho \,:\, \tau \dashv \Gamma_3}$$

ROWINTRO

$$\frac{\Gamma_1 \vdash_v e \,:\, \tau}{\Gamma_1 \vdash_v e \,:\, \forall \theta. \tau}$$

ROWELIM

$$\frac{\Gamma_1 \vdash e \,:\, \rho \,:\, \forall \theta. \tau \dashv \Gamma_2}{\Gamma_1 \vdash e \,:\, \rho \,:\, \tau[\hat{\rho}/\theta] \dashv \Gamma_2}$$

ALLOC

$$\frac{\Gamma_1 \vdash e \,:\, \rho \,:\, \tau \dashv \Gamma_2}{\Gamma_1 \vdash \text{ref } e \,:\, \rho \,:\, \mathbf{Ref}\,\tau \dashv \Gamma_2}$$

READ

$$\frac{\text{Multi}\,\tau \qquad \Gamma_1 \vdash e \,:\, \rho \,:\, \mathbf{Ref}\,\tau \dashv \Gamma_2}{\Gamma_1 \vdash \,!e \,:\, \rho \,:\, \tau \dashv \Gamma_2}$$

REPLACE

$$\frac{\Gamma_1 \vdash e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_1 \,:\, \rho \,:\, \mathbf{Ref}\,\tau \dashv \Gamma_3}{\Gamma_1 \vdash \text{replace } e_1 \, e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_3}$$

DO

$$\frac{\sigma = \forall \vec{\alpha}. \iota \Rrightarrow \kappa \qquad \rho = (op : \sigma) \cdot \hat{\rho} \qquad \Gamma_1 \vdash e \,:\, \rho \,:\, \iota\,[\vec{\tau}/\vec{\alpha}] \dashv \Gamma_2}{\Gamma_1 \vdash \text{do } op\, e \,:\, \rho \,:\, \kappa\,[\vec{\tau}/\vec{\alpha}] \dashv \Gamma_2}$$

HANDLER

$$\frac{\sigma = \forall \vec{\alpha}. \iota \Rrightarrow \kappa \qquad \hat{\rho} = (op : \sigma) \cdot \rho \qquad \text{Multi}\,\Gamma \qquad \Gamma_1 \vdash e \,:\, \hat{\rho} \,:\, \tau \dashv \Gamma_2 \qquad x : \iota, k : \kappa \xrightarrow{\rho} \tau', \Gamma \vdash h \,:\, \rho \,:\, \tau' \dashv \Gamma \qquad x : \tau, \Gamma_2; \Gamma \vdash r \,:\, \rho \,:\, \tau' \dashv \Gamma}{\Gamma_1; \Gamma \vdash \text{handle } e \text{ by } op\, x\, k. h \mid \text{ret } x. r \,:\, \rho \,:\, \tau' \dashv \Gamma}$$

Fig. 3. Selected typing rules of Affect$_{\text{OS}}$.

and result type can depend on both of these variables. At the effect call site, the type variables $\vec{\alpha}$ are instantiated with concrete types $\vec{\tau}$.

The HANDLER rule specifies that the overall effect row ρ and result type τ' must match with that of the branches. To type the effect branch, the effect call value x is required to have type ι, and the continuation k must have argument type κ due to the effect signature being $\forall \vec{\alpha}. \iota \Rrightarrow \kappa$. Additionally, h needs to be well-typed for any instantiation of type variables $\vec{\alpha}$, which is expressed by letting $\vec{\alpha}$ be free and distinct from all other type variables. The initial context is split into Γ_1 and Γ with Γ_1 forwarded to expression e and Γ towards the branches h and r of the effect handler. The expression e evaluates first, and when a value is reached the return branch r takes over, which explains the presence of Γ_2 in its initial context. The effect branch h on the other hand can only access the Γ context and not Γ_2. At the moment when the effect branch h takes control, the expression e has not finished executing and thus the context Γ_2 is not yet available. The effect branch h can be called multiple times when more than one effect call occurs in e. This forces us to require the context Γ to be unrestricted, which is captured by the Multi Γ constraint that requires every variable in Γ to

satisfy Multi. The handler also reinstates itself in the continuation, and as a result the continuation inherits the return type and effect row of the handler.

4 Combined One-Shot and Multi-Shot Affect Language

Multi-shot effects, *i.e.*, effects that have their continuation resumed more than once, offer additional expressive power over their one-shot counterparts, making it possible to express among others backtracking and probabilistic programming [25, 32, 45]. It is beneficial for a language to support both kinds of effects, but done in a controlled manner where the type system distinguishes them by tracking how continuations are called. This way we can get the additional expressivity of multi-shot effects, and at the same time keep the performance benefits of one-shot effects.

In this section we extend Affect with multi-shot effects. A key requirement that the extended type system must address is the *capturing condition*, which forbids substructural resources such as one-shot continuations from being captured in multi-shot continuations. Failure to do so can break the one-shot discipline of one-shot effects (and thus type soundness) as we will show. Furthermore, inspired by Wadler [57]'s *use types*, we introduce mode polymorphism, and generalise bang types to express the relationship between substructural types and multi-shot effects at the type level.

We describe the extended syntax and operational semantics (§ 4.1), types and subtyping relation (§ 4.2), typing rules (§ 4.3), and finally mode polymorphism (§ 4.4).

4.1 Syntax and Operational Semantics

We generalise the one-shot handlers from § 3.1 by tagging them with a *mode* $m \in$ Mode, which says that the corresponding continuation should be resumed *at most once* (O) or *zero or more times* (M):

$$m \in \text{Mode} ::= v \mid \text{O} \mid \text{M}$$
$$e \in \text{Expr} ::= \ldots \mid (\text{handle}_m\ e\ \text{by}\ op\ x\ k.\ e \mid \text{ret}\ x.\ e)$$
$$K1 \in \text{EvalCtx} ::= \ldots \mid (\text{handle}_m\ []\ \text{by}\ op\ x\ k.\ e \mid \text{ret}\ x.\ e)$$

We anticipate the introduction of mode polymorphism, so modes m also include mode variables v. However handlers can only use O and M because the operational semantics of Affect is defined on untyped expressions.

The reduction rule for one-shot handlers is identical to the rule Red-Handler of Affect$_{OS}$. For multi-shot handlers, we do not track operationally if their continuations are resumed, so cont ℓ N values are not used. Instead we directly reify the context K captured by the effect call as an abstraction that is substituted into the variable k, allowing k to be resumed multiple times:

$$\text{Red-Handler-MS}\ \frac{\text{handle}_\text{M}\ (\text{eff}\ op\ v\ K)\ \text{by}\ op\ x\ k.\ h \mid \text{ret}\ x.\ r\ /\ \sigma}{h[v/x, (\lambda x.\ \text{handle}_\text{M}\ K[x]\ \text{by}\ op\ x\ k.\ h \mid \text{ret}\ x.\ r)/k]\ /\ \sigma} \to_\text{h}$$

4.2 Types and Subtyping

Selected subtyping rules are shown in Fig. 4. We extend types, signatures and rows from § 3.3 as:

$$\tau, \kappa, \iota \in \text{Type} ::= \ldots \mid !_m \tau \mid \forall v.\ \tau$$
$$\sigma, \hat{\sigma} \in \text{Signature} ::= \text{i}_m \sigma \mid \forall \vec{\alpha}.\ \tau \Rightarrow \kappa$$
$$\rho, \hat{\rho} \in \text{Row} ::= \ldots \mid \text{i}_m \rho$$

Similar to Wadler [57], *generalised bang types* $!_m \tau$ are tagged with a mode with the property that $!_\text{M} \tau$ represents unrestricted types just as $!\tau$ does, and $!_\text{O} \tau$ is equivalent to τ (GBangO-Intro). We use $!\tau$ as sugar for $!_\text{M} \tau$. Modes are ordered O $\prec$: M (Mode-O, Mode-M), and the relation extends to bang types (GBang-Comp). Mode polymorphic types $\forall v.\ \tau$ enable abstraction over modes.

$$\text{Mode-O} \qquad\qquad \text{Mode-M} \qquad\qquad \frac{\text{SigM}}{\forall \vec{\alpha}. \, \tau' \Rightarrow \kappa' <: \forall \vec{\alpha}. \, \tau \Rightarrow \kappa}$$

$$\text{O} <: m \qquad\qquad\qquad m <: \text{M} \qquad\qquad\qquad \frac{\tau' <: \tau \qquad \kappa <: \kappa'}{\forall \vec{\alpha}. \, \tau' \Rightarrow \kappa' <: \forall \vec{\alpha}. \, \tau \Rightarrow \kappa}$$

$$\begin{array}{ccccc}
\text{GBang-Intro-Bool} & \text{GBang-Intro-Ref} & \text{GBangO-Intro} & \text{GBang-Idemp} & \text{GBang-Elim} \\
\mathbf{B} <: {!}_m \, \mathbf{B} & \mathbf{Ref} \, \tau <: {!}_m \, (\mathbf{Ref} \, \tau) & \tau <: {!}_{\text{O}} \, \tau & {!}_m \, \tau <:> {!}_m \, {!}_m \, \tau & {!}_m \, \tau <: \tau
\end{array}$$

$$\begin{array}{cccc}
\text{GBang-Comm} & \text{GBang-Comp} & \text{GBang-AllT-Comm} & \text{GBang-AllR-Comm} \\
{!}_m \, {!}_{m'} \, \tau <: {!}_{m'} \, {!}_m \, \tau & \dfrac{m' <: m \qquad \tau <: \kappa}{{!}_m \, \tau <: {!}_{m'} \, \kappa} & {!}_m \, \forall \alpha. \, \tau <:> \forall \alpha. \, {!}_m \, \tau & {!}_m \, \forall \theta. \, \tau <:> \forall \theta. \, {!}_m \, \tau
\end{array}$$

$$\begin{array}{ccc}
\text{FBang-Elim-Nil} & \text{FBang-Dist-Cons} & \text{FBang-Elim-Rec} \\
\mathsf{i}_m \, \langle \rangle <: \langle \rangle & \mathsf{i}_m \, ((op : \sigma) \cdot \rho) <: (op : \mathsf{i}_m \, \sigma) \cdot \mathsf{i}_m \, \rho & \dfrac{\mathsf{i}_m \, \rho <: \rho}{\mathsf{i}_m \, (\mu\theta. \, \rho) <: \mu\theta. \, \rho}
\end{array}$$

$$\begin{array}{ccccc}
\text{FBangM-Elim} & \text{FBang-Intro} & \text{FBang-Idemp} & \text{FBang-Comp} & \text{FBang-Comm} \\
\mathsf{i}_{\text{M}} \, \rho <: \rho & \rho <: \mathsf{i}_m \, \rho & \mathsf{i}_m \, \mathsf{i}_m \, \rho <:> \mathsf{i}_m \, \rho & \dfrac{m' <: m \qquad \hat{\rho} <: \rho}{\mathsf{i}_m \, \hat{\rho} <: \mathsf{i}_{m'} \, \rho} & \mathsf{i}_m \, \mathsf{i}_{m'} \, \rho <: \mathsf{i}_{m'} \, \mathsf{i}_m \, \rho
\end{array}$$

Fig. 4. Selected subtyping rules of Affect.

The new *flip-bang* signature $\mathsf{i}_m \, \sigma$ is used to represent one-shot effects in a similar manner to how bang types represent unrestricted types in ordinary linear type systems. In our combined one-shot and multi-shot language, effect signatures $\forall \vec{\alpha}. \, \tau \Rightarrow \kappa$ are by default multi-shot. Hence, one-shot signatures $\forall \vec{\alpha}. \, \tau \Rrightarrow \kappa$ are sugar for $\mathsf{i}_{\text{O}} \, (\forall \vec{\alpha}. \, \tau \Rightarrow \kappa)$ (we also use $(\forall \vec{\alpha}. \, \tau \Rightarrow_m \kappa)$ to denote $\mathsf{i}_m \, (\forall \vec{\alpha}. \, \tau \Rightarrow \kappa)$). Effect rows also have a flip-bang row $\mathsf{i}_m \, \rho$ to represent multiple one-shot effects.

Flip-bang signatures $\mathsf{i}_m \, \sigma$ and rows $\mathsf{i}_m \, \rho$ are dual to bang types ${!}_m \, \tau$. That is, while inhabitants of type ${!}_{\text{M}} \, \tau$ can be used multiple times, the continuation of an effect $\mathsf{i}_{\text{O}} \, \sigma$ must be used at most once. This duality in conjunction with mode polymorphic types $\forall \nu. \, \tau$ is used to capture the relations between the substructurality of polymorphic types and effects, see § 4.4.

The duality between bang and flip-bang is further illustrated in the subtyping relation. Elimination of flip-bang rows is restricted (FBang-Elim-Nil, FBang-Dist-Cons and FBang-Elim-Rec), unless the mode is M (FBangM-Elim). However, as opposed to bang types, flip-bang can be introduced to any row (FBang-Intro). Note that the flip-bang signature $\mathsf{i}_m \, \sigma$ enjoys identical FBangM-Elim, FBang-Intro, FBang-Idemp, FBang-Comp, and FBang-Comm rules.

Just as with the Multi τ constraint, its dual Once ρ constraint is defined accordingly to identify effect rows that can only be handled by one-shot handlers:

$$\text{Once } \rho := \mathsf{i}\rho <: \rho$$

4.3 Typing Rules

The typing rules from Affect$_{\text{OS}}$ in § 3.5 need to be adapted to soundly track the interplay between one-shot and multi-shot effects. The type system additionally needs to enforce the *capturing condition*, which says that substructural resources are not captured in multi-shot continuations.

The rules Do and Handler from Affect$_{\text{OS}}$ need to be generalised to allow multi-shot effects, and rules App and Replace need to be modified since they are unsound in the combined one-shot and multi-shot language. On the other hand, Let, IfElse, RowIntro, RowElim, Alloc and Read do not need to be changed. We can directly sequence their sub-derivations as specified by the typing

rules in Fig. 3, even in the presence of multi-shot effects due to how two-context judgements are interpreted. The final context contains exactly the variables from the initial that *can* be used after evaluation which means that when an expression performs a multi-shot effect, the final context only has variables of unrestricted types. For instance, in LET if e_1 performs a multi-shot effect, variables in Γ_2 will be of unrestricted types which coincidentally are also the variables that can be captured in the continuation.

To present counterexamples for the rules that need to be changed, we assume that $(k : \mathbf{1} \multimap \mathbf{1})$ is a one-shot continuation. We can bring such a k into the context by wrapping a computation in:

$$\begin{aligned}
&\mathtt{let}\ (k : \mathbf{1} \multimap \mathbf{1}) = \mathtt{let}\ (r : \mathbf{Ref}\ (\mathbf{1} \multimap \mathbf{1})) = \mathtt{ref}\ (\lambda\,().\,()) \ \mathtt{in} \\
&\qquad\qquad\qquad\qquad\ (\mathtt{handle}_\mathrm{O}\ (\mathtt{do}\ op\ ())\ \mathtt{by}\ op\ ()\ k'.\,r \coloneqq k'\ |\ \mathtt{ret}\ x.\,x); \\
&\qquad\qquad\qquad\qquad\ \mathtt{replace}\ r\ (\lambda\,().\,()) \\
&\ \mathtt{in} \dots
\end{aligned}$$

Here, we use $\mathbf{Eff} \coloneqq (op : \mathbf{1} \Rrightarrow \mathbf{1})$. We utilise a reference r to extract the continuation from the effect branch of a one-shot handler. The call to `replace` extracts the continuation from the reference. Note that we use `replace` instead of an ordinary load since we are dealing with a one-shot continuation. Similar to the pattern in *iter2gen* (§ 2.2), we employ a dummy value $(\lambda\,().\,())$.

Performing and handling an effect. Figure 5 shows that DoGEN is now defined on a flip-bang signature $\mathfrak{i}_m\ (\forall\vec{\alpha}.\,\iota \Rightarrow \kappa)$ and introduces a substructurality constraint $m \preceq \Gamma_2$. For the one-shot case, the typing rule is identical to Affect$_\mathrm{OS}$ as the constraint $\mathrm{O} \preceq \Gamma_2$ is trivially satisfied (MODESUB-O). However for the multi-shot case, the final context Γ_2 must consist of only variables with unrestricted types (MODESUB-M). Recall from § 2.4 that this is needed to rule out unsafe expressions such as *handle_choice* $(\lambda\,().\,\mathtt{do}\ choice\ ();\,k\,())$. The relation $m \preceq \Gamma_2$ ensures that k cannot appear in the final context of the $\mathtt{do}\ choice\ ()$ expression, and as a result is ill-typed.

HANDLERGEN rule generalises the function type of continuations k compared to the HANDLER rule in Affect$_\mathrm{OS}$. One-shot handlers (with mode m as O) require one-shot signatures and the continuation k must be used affinely, as $!_\mathrm{O}\ (\kappa \xrightarrow{\rho} \tau')$ is equivalent to $\kappa \xrightarrow{\rho} \tau'$. Multi-shot handlers (with mode m as M) allow the continuation k of the effect to be used more than once. This is reflected in the typing rule by giving k the unrestricted function type when m is M.

Application typing. The substructurality restriction via the DoGEN rule alone is insufficient to ensure that variables captured by multi-shot effects are of unrestricted types. To see why, let us slightly tweak the previous counterexample to *handle_choice* $(\lambda\,().\,(\lambda\,().\,\mathtt{do}\ choice\ ())\,();\,k\,())$ which encloses the effect call in an abstraction and calls it directly afterwards. The semantics is the same as before, so k is called twice. The restriction in DoGEN does not rule out this program because the body of an abstraction is typed with the portion of the context that is accessed (VALTYPED followed by ABS). Hence, the abstraction $\lambda\,().\,\mathtt{do}\ choice\ ()$ is typed with the empty context and the relation $\mathrm{M} \preceq \emptyset$ is trivially satisfied. To address this, the rule APPGEN requires via the substructurality relation $\rho \preceq \Gamma_3$ that when the application produces multi-shot effects all variables used afterwards are of unrestricted types. The condition $\rho \preceq \Gamma_3$ generalises the substructurality relation $m \preceq \Gamma$ to effect rows, and states that Γ_3 is unrestricted when ρ contains at least one multi-shot effect.

We also need to restrict the argument type of the function using $\hat{\rho} \preceq \tau$. Consider the application $(\mathtt{do}\ choice\ ();\,\lambda\,x.\,x\,())\,k$ that calls *choice* and returns an evaluator function that is called with the one-shot continuation k. We can again break the one-shot guarantee of k by installing *handle_choice* to it. The final context Γ_3 in the application of $(\mathtt{do}\ choice\ ();\,\lambda\,x.\,x\,())$ with k is empty, and so the previous substructurality constraint is trivially satisfied. The substructurality constraint $\hat{\rho} \preceq \tau$ is needed to require unrestricted argument types when $\hat{\rho}$ contains a multi-shot effect.

$$\boxed{m \preceq \tau} \quad \boxed{m \preceq \Gamma}$$

MODESUB-O

$\textsc{o} \preceq \tau$

MODESUB-M

$\dfrac{\text{Multi } \tau}{\textsc{m} \preceq \tau}$

MODESUB-NIL

$m \preceq \emptyset$

MODESUB-CONS

$\dfrac{m \preceq \tau \qquad m \preceq \Gamma}{m \preceq x : \tau, \Gamma}$

$$\boxed{\rho \preceq \tau} \quad \boxed{\rho \preceq \Gamma}$$

ROWSUB-ONCE

$\dfrac{\text{Once } \rho}{\rho \preceq \tau}$

ROWSUB-MULTI

$\dfrac{\text{Multi } \tau}{\rho \preceq \tau}$

ROWSUB-NIL

$\rho \preceq \varnothing$

ROWSUB-CONS

$\dfrac{\rho \preceq \tau \qquad \rho \preceq \Gamma}{\rho \preceq x : \tau, \Gamma}$

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

DOGEN

$$\dfrac{\begin{array}{c} m \preceq \Gamma_2 \\ \sigma = \mathsf{i}_m\,(\forall \vec{\alpha}.\,\iota \Rightarrow \kappa) \qquad \rho = (op : \sigma) \cdot \hat{\rho} \\ \Gamma_1 \vdash e \,:\, \rho \,:\, \iota\,[\vec{\tau}/\vec{\alpha}] \dashv \Gamma_2 \end{array}}{\Gamma_1 \vdash \mathsf{do}\ op\ e \,:\, \rho \,:\, \kappa\,[\vec{\tau}/\vec{\alpha}] \dashv \Gamma_2}$$

HANDLERGEN

$$\dfrac{\begin{array}{c} \sigma = \mathsf{i}_m\,(\forall \vec{\alpha}.\,\iota \Rightarrow \kappa) \qquad \hat{\rho} = (op : \sigma) \cdot \rho \qquad \text{Multi } \Gamma \\ \Gamma_1 \vdash e \,:\, \hat{\rho} \,:\, \tau \dashv \Gamma_2 \\ x : \iota, k : \mathsf{!}_m(\kappa \xrightarrow{\rho} \tau'), \Gamma \vdash h \,:\, \rho \,:\, \tau' \dashv \Gamma \\ x : \tau, \Gamma_2; \Gamma \vdash r \,:\, \rho \,:\, \tau' \dashv \Gamma \end{array}}{\Gamma_1; \Gamma \vdash \mathsf{handle}_m\ e\ \mathsf{by}\ op\ x\ k.\ h \mid \mathsf{ret}\ x.\ r \,:\, \rho \,:\, \tau' \dashv \Gamma}$$

APPGEN

$$\dfrac{\begin{array}{c} \hat{\rho} \lessdot \rho \qquad \hat{\rho} \preceq \tau \qquad \rho \preceq \Gamma_3 \\ \Gamma_1 \vdash e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_2 \\ \Gamma_2 \vdash e_1 \,:\, \hat{\rho} \,:\, \tau \xrightarrow{\rho} \kappa \dashv \Gamma_3 \end{array}}{\Gamma_1 \vdash e_1\,e_2 \,:\, \rho \,:\, \kappa \dashv \Gamma_3}$$

REPLACEGEN

$$\dfrac{\begin{array}{c} \hat{\rho} \lessdot \rho \qquad \hat{\rho} \preceq \tau \\ \Gamma_1 \vdash e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_2 \\ \Gamma_2 \vdash e_1 \,:\, \hat{\rho} \,:\, \mathbf{Ref}\ \tau \dashv \Gamma_3 \end{array}}{\Gamma_1 \vdash \mathsf{replace}\ e_1\,e_2 \,:\, \rho \,:\, \tau \dashv \Gamma_3}$$

Fig. 5. Selected typing rules of Affect. (Changes compared to Affect$_{\text{OS}}$ are displayed in gray.)

To make the APPGEN rule less restrictive, instead of specifying the substructurality constraint on the overall row ρ, it is defined on the compatible row $\hat{\rho}$. This way, τ can be substructural when $\hat{\rho}$ does not contain multi-shot effects even if ρ does ($\hat{\rho} \lessdot \rho$ requires that all effects in $\hat{\rho}$ are in ρ but not the other way around).

Replace typing. REPLACE needs to be adapted similarly to APPGEN. When e_1 produces a multi-shot effect, the evaluation of e_2 will be captured in the continuation and so τ must be unrestricted.

4.4 Mode Polymorphism

Polymorphism is more subtle in the combined one-shot and multi-shot language compared to Affect$_{\text{OS}}$ due to the substructurality relations between quantified effect rows and the resources captured in their continuations. Polymorphism over modes as well as generalised bang and flip-bangs allow us to express these relationships at the type level.

To illustrate this, reconsider the higher-order identity function $g := \lambda f\,x.\,f\,()\,;\,x$ from § 2.4. We want to capture two requirements: (1) a partial application $g\,f$ with some function f could be used at most as much as f, and (2) the type of x has to be unrestricted when f produces multi-shot effects. The type signature that expresses both requirements is (recall, $\tau \xrightarrow{\rho}_m \kappa$ is sugar for $!_m(\tau \xrightarrow{\rho} \kappa)$):

$$\forall \nu\,\nu'\,\theta\,\alpha.\ (\mathbf{1} \xrightarrow{\mathsf{i}_\nu\,\theta}_{\nu'} \mathbf{1}) \rightarrow\ !_\nu\,\alpha \xrightarrow{\mathsf{i}_\nu\,\theta}_{\nu'}\ !_\nu\,\alpha$$

Mode polymorphism on v' achieves (1), and the duality between $i_v \theta$ and $!_v \alpha$ achieves (2).

Two new rules are added that allow us to work with generalised bang types and flip-bang rows:

MODESUB-MBANG

$$m \preceq \, !_m \tau$$

ROWSUB-FMBANG

$$i_m \rho \preceq \, !_m \tau$$

The rule MODESUB-MBANG allows us to frame variables of type $!_m \tau$ when performing an m effect via DOGEN. Similarly, ROWSUB-FMBANG allows us to frame variables of type $!_m \tau$ in APPGEN when m-restricted rows are performed.

Note that mode quantification and bang/flip-bangs are not powerful enough to capture all forms of bounded quantification. A stacking of bang types $!_m !_{m'} \tau$ says that τ is unrestricted if *any* one of the two modes is M. Currently, we cannot express the dual requirement that *both* modes need to be M for τ to be unrestricted. While the expressiveness of Affect is sufficient to give generic types to common programming patterns (such as iterators, see § 2.5), it might be useful to equip Affect with join $\sqcup$ and meet $\sqcap$ operations on modes to make it possible to express this dual requirement.

5 Semantic Typing

Recall from § 3.2 that type soundness ensures that well-typed programs are safe, which particularly means they obey the one-shot discipline and leave no effects unhandled:

$$\vdash e : \langle\rangle : \tau \dashv \quad \implies \quad \textit{safe } e$$

In this section we prove that Affect enjoys type soundness using the logical approach to semantic typing in the Iris logic [29–31, 34, 53]. We begin with an introduction to semantic typing (§ 5.1), followed by a brief explanation of the AffectLogic program logic that we use (§ 5.2), and lastly give the semantic definition of the type system into AffectLogic (§ 5.3).

5.1 What is Semantic Typing

Semantic typing is an approach to proving type soundness that interprets types and typing judgements into a semantic domain such as a higher-order logic [1, 4]. It contrasts with the syntactical approach to type soundness, which relies on the Progress and Preservation theorems [21, 41, 60]. The central idea of semantic typing is building a logical relation defined recursively over the objects in the language such as types. We take the *logical approach* to semantic typing [5, 20, 53], where our semantic domain is the Iris-based AffectLogic program logic that allows us to abstractly reason about effectful programs. Typing judgements become propositions in the logic and types become predicates over values. Effect signatures and effect rows are represented as higher-order predicates in the style of de Vilhena and Pottier [17]'s protocols. The main theorems are:

$$\vdash e : \langle\rangle : \tau \dashv \quad \xRightarrow{\text{FUNDAMENTAL}} \quad \vDash e : \langle\rangle : \tau \dashv \quad \xRightarrow{\text{ADEQUACY}} \quad \textit{safe } e$$

A semantic type judgement $\Gamma_1 \vDash e : \rho : \tau \dashv \Gamma_2$ 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 typing judgement in terms of the program logic (ADEQUACY arrow). The fundamental theorem (FUNDAMENTAL arrow) relates the syntactic type judgement to the semantic one by interpreting syntactic objects semantically, and by showing that each syntactic typing rule has a semantic version. In the semantic world, semantic typing rules are lemmas that we prove.

Throughout this section we avoid referring to syntactic type, signature and effect row expressions. Inspired by Appel and Felty [3] and Jung et al. [28], we adopt a purely semantic viewpoint where these notions are really objects in the logic. We do not define the (straightforward) translation from syntactic to semantic expressions, which is also the approach taken in our Coq development.

5.2 Program Logic

To reason about effectful programs we define AffectLogic, a separation logic suited for proving safety and partial program correctness. AffectLogic is derived from the Hazel and Maze program logics of de Vilhena and Pottier [15–17], which support solely one-shot and solely multi-shot (unnamed) effects, respectively. AffectLogic introduces effect rows and combines one- and multi-shot effects successfully with only little extra overhead in complexity. Hazel and Maze, and thus AffectLogic, are defined on top of Iris [29–31, 34, 53]: a higher-order, separation logic framework with Coq support for machine-checked proofs [33, 35]. A snippet of the AffectLogic syntax is:

$$P, Q, R \in \text{iProp} := \text{True} \mid \text{False} \mid P * Q \mid P \ast Q \mid \ell \mapsto v \mid \forall x. P \mid \exists x. P$$
$$\mid \Box P \mid \rhd P \mid \boxed{P}^{\mathcal{N}} \mid \text{ewp}_{\mathcal{E}} \, e \,_{\rho} \{\Phi\} \mid \ldots$$

Propositions iProp in the logic are resource aware (they can assert ownership of locations ℓ) and are step-indexed over the computation steps to enable recursive reasoning [1, 4]. Apart from the usual propositional logic connectives (which are omitted) there are a number of features that are essential to our work. First, there are the usual connectives of separation logic [40], which make it possible to model affine types. The *separating conjunction* $(P * Q)$ asserts that resources referenced by P and Q are distinct. The *magic wand* $P \ast Q$ says that Q holds provided additional resources P are provided. The *points-to connective* $\ell \mapsto v$ asserts ownership of location ℓ with value v.

Impredicative higher-order quantification $\forall x. P$ and $\exists x. P$ is used to model polymorphic types. The *persistence modality* $\Box P$, which implies that P is duplicable, allows us to model bang types. The *later modality* $\rhd P$, which expresses that P holds after one step of computation, allows us to support recursive types and effects. The *invariant assertion* $\boxed{P}^{\mathcal{N}}$ allow us to share resources, and is crucial to model mutable references. Finally, to model the typing judgements we use the *extended weakest precondition* $\text{ewp}_{\mathcal{E}} \, e \,_{\rho} \{\Phi\}$, which states that e is safe, performs effects according to semantic row ρ and if it terminates, its evaluation satisfies predicate Φ. The mask $\mathcal{E}$ in weakest preconditions is needed to track open *invariants* and thus ensure soundness of the logic. For the sake of clarity, we will drop the mask from hereafter, as it is mostly an administrative detail. We also use the notation $\text{ewp} \, e \, \{\Phi\}$ for non-effectful expressions instead of $\text{ewp} \, e \,_{\langle \rangle} \{\Phi\}$.

Specifications have the form $P \vdash Q$, where logical entailment is used to denote that Q follows from P. Intuitively, propositions P and Q can be thought as predicates over heaps, and entailment as requiring that for any heap σ, $P \, \sigma$ implies $Q \, \sigma$. Using entailment and extended weakest preconditions we can recover Hoare triples that track the effects as $\{P\} \, e \,_{\rho} \{\Phi\} := P \vdash \text{ewp} \, e \,_{\rho} \{\Phi\}$. We now present some features of AffectLogic, Hazel, Maze and Iris in more detail.

Persistence modality. The persistence modality ($\Box$) asserts non-exclusive ownership of a proposition. Iris, the underlying logic of AffectLogic, is an affine logic and the persistence modality is used to represent unrestricted propositions. Its main proof rules are $\Box P \vdash \Box P * \Box P$, which allows us to duplicate a proposition; $\Box P \vdash P$ for elimination; and $\boxed{P}^{\mathcal{N}} \vdash \Box \boxed{P}^{\mathcal{N}}$, which states that invariants are persistent and thus, when an invariant is established it becomes shareable knowledge.

Invariants. Impredicative invariants [50] $\boxed{P}^{\mathcal{N}}$ allow us to place invariants P on the state of resources. Once an invariant $\boxed{P}^{\mathcal{N}}$ is allocated, the proposition P will hold in all future steps. To ensure the P remains to hold, we can only obtain ownership of P for the duration of individual atomic steps of execution—called *opening* the invariant. The $\mathcal{N}$ superscript denotes the invariant's namespace, which is used to track opened invariants and is necessary for soundness of the logic.

Proof rules. Figure 6 shows selected rules of AffectLogic. The rule EWP-Val expects a non-effectful extended weakest precondition since values do not produce effects. The rule EWP-Pure takes a single pure step in the execution by requiring that after one step (due to later modality) the

$$\Phi\, v \vdash \mathsf{ewp}\ v\ \{\Phi\} \tag{EWP-Val}$$

$$(e_1 \rightarrow_{\mathsf{pure}} e_2) * \triangleright (\mathsf{ewp}\ e_2\ _\rho\{\Phi\}) \vdash \mathsf{ewp}\ e_1\ _\rho\{\Phi\} \tag{EWP-Pure}$$

$$\ell \mapsto w \vdash \mathsf{ewp}\ !\ell\ \{v.\, v = w * \ell \mapsto w\} \tag{EWP-Load}$$

$$\ell \mapsto u \vdash \mathsf{ewp}\ (\mathsf{replace}\ \ell\ w)\ \{v.\, v = u * \ell \mapsto w\} \tag{EWP-Replace}$$

$$\mathsf{ewp}\ e\ _\rho\{v.\, \mathsf{ewp}\ N[v]\ _\rho\{\Phi\}\} \vdash \mathsf{ewp}\ N[e]\ _\rho\{\Phi\} \tag{EWP-Bind}$$

$$\rho \text{ mono in } R * R * \mathsf{ewp}\ e\ _\rho\{\Phi\} \vdash \mathsf{ewp}\ e\ _\rho\{v.\, \Phi\, v * R\} \tag{EWP-Frame}$$

$$\rho\, op\, v\, \Phi \vdash \mathsf{ewp}\ (\mathsf{do}\ op\ v)\ _\rho\{\Phi\} \tag{EWP-Do}$$

Fig. 6. Selected proof rules of AffectLogic.

resulting weakest precondition for e_2 holds. The rules EWP-Load and EWP-Replace are standard proof rules for reading and replacing from a reference respectively. The rule EWP-Bind enables modular reasoning, and it is notably restricted to neutral contexts N instead of arbitrary contexts K. Neutral contexts ensure that no intermediary handlers are captured in N, and thus e must indeed perform effects according to row ρ [17]. Framing through the weakest precondition requires that row ρ is monotonic in the proposition R to be framed.

Before we explain the monotonicity property, let us first look into how semantic signatures and rows are represented in the logic. Semantic signatures σ capture the communication behaviour of a handlee and handler such as the values they can exchange and the transfer of resource ownership. They are represented as morphisms from effect values and continuation predicates to propositions, and correspond to *persistently monotonic protocols* in the Maze logic. Semantic rows ρ group signatures together and are morphisms from operations to signatures:

$$\sigma \in \mathsf{Sig} := \mathsf{Val} \rightarrow (\mathsf{Val} \rightarrow \mathsf{iProp}) \rightarrow \mathsf{iProp}$$
$$\rho \in \mathsf{Row} := \mathsf{Operation} \rightarrow \mathsf{Sig}$$

To perform an effect, EWP-Do requires a proof of $\rho\, op\, v\, \Phi$, where op is the effect to be performed, v is the effect value, and Φ is the postcondition of the weakest precondition. Crucially, the postcondition Φ can be seen as a predicate which ensures that plugging the handler's response to the current context is safe. For instance, to prove $\mathsf{ewp}\ \mathsf{ref}\ (\mathsf{do}\ op\ ())\ _\rho\{\Phi\}$ for some row ρ and postcondition Φ, we first apply EWP-Bind which gives $\mathsf{ewp}\ \mathsf{do}\ op\ ()\ _\rho\{v.\, \mathsf{ewp}\ \mathsf{ref}\ v\ _\rho\{\Phi\}\}$ followed by EWP-Do that requires $\rho\, op\, ()\, (\lambda v.\, \mathsf{ewp}\ \mathsf{ref}\ v\ _\rho\{\Phi\})$. Thus the continuation predicate that is passed to row ρ is a predicate that proves safety for the rest of the computation.

As a result, EWP-Frame is only sound in the presence of effects ρ if we can extend the continuation predicate that is passed to the effect row with proposition R. This exact requirement is captured by the predicate ρ mono in R:

$$\rho \text{ mono in } R := \square\, (\forall\, op\, v\, \Phi.\, \rho\, op\, v\, \Phi \twoheadrightarrow R \twoheadrightarrow \rho\, op\, v\, (\lambda\, w.\, \Phi\, w * R))$$

EWP-Handle is used to verify handlers for effectful expressions e that perform effects op with signature $\forall \vec{\alpha}.\, \tau \Rightarrow_m \kappa$ (a more general version of EWP-Handle similar to de Vilhena [15] can be

found in our Coq development [56]):

EWP-Handle

$$\frac{\begin{array}{c} \text{ewp } e \,_{(op:\forall \vec{\alpha}.\ \tau \Rightarrow_m \kappa)\cdot\rho} \{\Phi\} \quad * \\ \Box\,(\forall v.\ \Phi\, v \relbar\joinrel\ast \text{ewp } r[v/x] \,_\rho \{\Phi'\}) \quad * \\ \Box\,(\forall v_x\, v_k.\ (\exists\, \vec{\alpha}.\ \tau\, v_x * \Box^m(\forall w.\ \kappa\, w \relbar\joinrel\ast \text{ewp } v_k\, w \,_\rho \{\Phi'\})) \relbar\joinrel\ast \text{ewp } h[v_x/x, v_k/k] \,_\rho \{\Phi'\}) }{\text{ewp } (\text{handle}_m\ e\ \text{by } op\ x\ k.\ h \mid \text{ret } x.\ r) \,_\rho \{\Phi'\}}$$

The weakest preconditions for the handler's branches are persistent since we are using deep handler semantics which leads to the branches being used more than once. Verification of the return branch amounts to showing that $r[v/x]$ is safe and satisfies Φ' for all values v that satisfy Φ since by assumption, if e evaluates to a value it will satisfy Φ. The effect branch requires $h[v_x/x, v_k/k]$ to be safe and satisfy Φ' for any values v_x and v_k that satisfy the signature. Value v_x represents the effect call value which is assumed to satisfy τ for some type variables $\vec{\alpha}$ and the continuation value v_k is assumed to be callable with any values w that satisfy κ. Multi-shot signatures allow the weakest precondition for v_k to be persistent and thus allow k to be called more than once in the effect branch. We refer the reader to de Vilhena [15] for a thorough discussion on the handler proof rule.

Adequacy theorem. Similar to Iris, AffectLogic enjoys the following adequacy theorem: *For any postcondition Φ and closed expression e, if* True $\vdash$ ewp e $\{\Phi\}$ *is derivable, then e is safe.*

The adequacy theorem proves soundness of the program logic. It states that extended weakest preconditions proved under an arbitrary heap imply expression safety. Recall that safety guarantees that one-shot continuations are called at most once, no effects are left unhandled and the expression either diverges or reaches a value. The theorem follows from the definition of extended weakest preconditions which for reasons of brevity we choose not to present. We refer the interested reader to de Vilhena [15] for a thorough explanation.

5.3   Semantic Definitions

The semantic interpretation of the Affect type system is shown in Figure 7.

Judgements. The judgement $\Gamma_1 \vDash e : \rho : \tau \dashv \Gamma_2$ is a proposition defined using the extended weakest precondition of AffectLogic. It requires that for any parallel substitution γ (*i.e.,* finite mapping of variables to values) that satisfies the initial context Γ_1, expression $e[\gamma]$ is safe to execute, performs only effects of row ρ and the resulting value is of type τ. Additionally, the final context Γ_2 must also be satisfied for the same variable map γ after evaluation of e. The fact that proposition $\Gamma_2 \vDash \gamma$ appears in the postcondition is what allows us to use the variables in Γ_2 after evaluation of e. The interpretation of the two-context judgement is similar to that of Jung et al. [28] for Rust, and that of Hinrichsen et al. [27] for session types.

Inspired by de Vilhena and Pottier [18], the value typing judgement $\Gamma \vDash_v e : \tau$ is defined using pwp, a version of ewp restricted to pure (no mutable state) and non-effectful expressions. Compared to ewp, pwp enjoys $\Box\,(\text{pwp } e\{\Phi\}) \vdash \text{pwp } e\{v.\ \Box\,(\Phi\, v)\}$ and $(\forall x.\ \text{pwp } e\{\Phi\}) \vdash \text{pwp } e\{v.\ \forall x.\ \Phi\, v\}$, which allow us to prove BangIntro and RowIntro, respectively.

Types. Types are predicates in the Iris logic. The definitions are mostly standard and are adapted to incorporate effects. We refer the reader to Timany et al. [53] for a more thorough discussion on this logical approach to typing, but we note that contrary to Timany et al. [53] we take a direct semantic approach and avoid interpreting syntactic expressions as they do (in the spirit of Appel and Felty [3] and Jung et al. [28]).

The boolean type is satisfied for boolean values. The bang type $!_m \tau$ is directly defined using the persistence modality. The $\Box^m P$ simplifies to P when m is O and $\Box P$ for m being M. Polymorphic

<div align="center">

Semantic judgements

$$\Gamma \models \gamma := \bigast_{(x,\tau)\in\Gamma}.\tau\,(\gamma(x))$$

$$\Gamma_1 \vDash e \,:\, \rho \,:\, \tau \dashv \Gamma_2 := \square\,(\forall\gamma.\,\Gamma_1 \models \gamma \mathrel{-\!\!*} \mathsf{ewp}\,e[\gamma]\,_\rho\{v.\,\tau\,v * \Gamma_2 \models \gamma\})$$

$$\Gamma \vDash_v e \,:\, \tau := \square\,(\forall\gamma.\,\Gamma \models \gamma \mathrel{-\!\!*} \mathsf{pwp}\,e[\gamma]\{\tau\})$$

Semantic types, signatures and rows

</div>

$$\tau, \kappa \in \mathsf{Ty} := \mathsf{Val} \to \mathsf{iProp} \qquad\qquad\qquad \sigma \in \mathsf{Sig} := \mathsf{Val} \to (\mathsf{Val} \to \mathsf{iProp}) \to \mathsf{iProp}$$

$$\mathbf{B} := \lambda v.\, v \in \mathbb{B} \qquad\qquad\qquad \forall\vec{\alpha}.\,\tau \Rightarrow \kappa := \lambda v\,\Phi.\,\exists\vec{\alpha}.\,\tau\,v * \square\,(\forall w.\,\kappa\,w \mathrel{-\!\!*} \Phi\,w)$$

$$!_m\,\tau := \lambda v.\,\square^m\,(\tau\,v) \qquad\qquad\qquad \mathsf{i}_m\,\sigma := \lambda v\,\Phi.\,\exists\Phi'.\,\sigma\,v\,\Phi' *$$
$$\square^m\,(\forall w.\,\Phi'\,w \mathrel{-\!\!*} \Phi\,w)$$

$$\forall\alpha.\,\tau := \lambda v.\,\forall\,\alpha : \mathsf{Ty}.\,\tau\,v \qquad\qquad\qquad \rho \in \mathsf{Row} := \mathsf{Operation} \to \mathsf{Sig}$$

$$\forall\theta.\,\tau := \lambda v.\,\forall\,\theta : \mathsf{Row}.\,\tau\,v \qquad\qquad\qquad \langle\rangle := \lambda\,op\,v\,\Phi.\,\mathsf{False}$$

$$\tau \xrightarrow{\rho} \kappa := \lambda v.\,\forall w.\,\tau\,w \mathrel{-\!\!*} \mathsf{ewp}\,v\,w\,_\rho\{\kappa\} \qquad (op : \sigma) \cdot \rho := \lambda\,op'\,v\,\Phi.\,\begin{cases} \triangleright(\sigma\,v\,\Phi) & op = op' \\ \rho\,op'\,v\,\Phi & op \neq op' \end{cases}$$

$$\mathbf{Ref}\,\tau := \lambda v.\,(v \in \mathsf{Loc}) * \boxed{\exists w.\,v \mapsto w * \tau\,w}^{\mathcal{N}_\ell} \qquad \mu\theta.\,\rho := \textit{fix}\,(\theta : \mathsf{Row}).\,\rho$$
$$\mathsf{i}_m\,\rho := \lambda\,op\,v\,\Phi.\,\exists\Phi'.\,\rho\,op\,v\,\Phi' *$$
$$\square^m\,(\forall w.\,\Phi'\,w \mathrel{-\!\!*} \Phi\,w)$$

<div align="center">

Semantic relations

</div>

$$\tau <: \kappa := \square\,(\forall v.\,\tau\,v \mathrel{-\!\!*} \kappa\,v) \qquad\qquad m \preceq \tau := \square\,(\forall v.\,\tau\,v \mathrel{-\!\!*} \square^m(\tau\,v))$$

$$\sigma <: \hat{\sigma} := \square\,(\forall v\,\Phi.\,\sigma\,v\,\Phi \mathrel{-\!\!*} \hat{\sigma}\,v\,\Phi) \qquad\qquad m \preceq \Gamma := \square\,(\forall\gamma.\,\Gamma \models v \mathrel{-\!\!*} \square^m(\Gamma \models \gamma))$$

$$\rho <: \hat{\rho} := \forall\,op.\,\rho\,op <: \hat{\rho}\,op \qquad\qquad \rho \preceq \tau := \forall v.\,\rho\,\text{mono in}\,\tau\,v$$

$$\Gamma <: \Gamma' := \square\,(\forall\gamma.\,\Gamma \models \gamma \mathrel{-\!\!*} \Gamma' \models \gamma) \qquad\qquad \rho \preceq \Gamma := \forall\gamma.\,\rho\,\text{mono in}\,(\Gamma \models \gamma)$$

Fig. 7. Definition of semantic judgements, types, signatures, rows and relations of Affect.

types $\forall\alpha.\,\tau$ are defined using higher-order quantification in the logic. Function types $\tau \xrightarrow{\rho} \kappa$ require applications where the argument satisfies τ to be safe, perform effects ρ and finally the resulting value to satisfy κ. Reference types utilise the points-to connective to state that the location is defined and points to a value that satisfies the inner type τ. The points-to connective and proposition $\tau\,w$ are stored behind an invariant $\mathcal{N}_\ell$. This ensures that location ℓ will always point to a value of type τ in all future computation steps under the restrictions on opening and closing of invariants. Most importantly, invariants allow us to treat reference types as unrestricted.

Signatures. When calling an effect, the handlee needs to show $(\forall\vec{\alpha}.\,\tau \Rightarrow \kappa)\,v\,\Phi$ for some effect value v and continuation predicate Φ. Since the instantiation of type variables $\vec{\alpha}$ happens by the handlee at the effect call site, the universal quantification on the signature level is transformed to an existential quantification on the logic level. The effect value must also satisfy argument type τ and the continuation predicate must be satisfiable for all values that satisfy κ. Signatures are assumed multi-shot by default and thus the continuation predicate applied to appropriate values must be persistent. (Recall, one-shot signatures are defined using flip-bang.)

Flip-bang signatures $\mathsf{i}_O\,\sigma$ are monotonic in all propositions (satisfy $\forall R.\,(\lambda\_.\,\sigma)\,\text{mono in}\,R$)., allowing us to frame arbitrary resources to the continuation predicate and as a result the effect in σ can capture substructural resources from its context. Note that $\mathsf{i}_M\,\sigma$ is logically equivalent to σ.

Rows. Semantic rows are morphisms from operations to signatures. Calling an effect op with value v translates to showing $\mathsf{ewp}\,\mathsf{do}\,op\,v\,_\rho\{\Phi\}$ for some postcondition Φ, which in turn requires $\rho\,op\,v\,\Phi$. The $\langle\rangle$ row, which represents the absence of effects, is thus defined as False. To call the

effect op with value v in the presence of row $op : \sigma \cdot \rho$ resorts to showing $\triangleright(\sigma\,v\,\Phi)$. The later modality $\triangleright$ guards the recursive occurrence of the row in signature σ, which is what allows us to take its fixpoint in $\mu\theta.\,\rho$. When the effect operation to be called does not match (case $op \neq op'$), the tail of the row is used. Flip-bang rows follow the same semantics as flip-bang signatures.

Relations. The subtyping relations between types, signatures and rows are defined pointwise and context subtyping firstly interprets the context into a proposition $\Gamma \models \gamma$.

The substructurality relation $m \preceq \tau$ that ensures τ is unrestricted when m is M is lifted to the logic using persistent propositions. The generalisation to contexts $m \preceq \Gamma$ follows a similar approach by firstly interpreting the context into a proposition using $\Gamma \models \gamma$.

The substructurality relations $\rho \preceq \tau$ and $\rho \preceq \Gamma$ used in typing rules AppGen and ReplaceGen ensure that the continuation of an effect can be extended with resources of type τ or variables specified by Γ. The monotonicity property captures this exact requirement.

6 Related Work

Substructural Type Systems. Many substructural type systems have been devised that restrict the contraction or weakening rules from logic. A key challenge of substructural type systems is to enable smooth integration between unrestricted and substructural types.

Traditionally, linear logics are equipped with the bang $!A$ proposition, and in this spirit, many substructural type systems adopt this approach [26, 57]. Unrestricted values are represented by bang types and through elimination rules we can freely cast them as substructural. Wadler [57] equips bang types with a use attribute $!^u\tau$, an idea we borrow and generalise by introducing its dual flip-bang for signatures and rows. Alternatively, types are qualified with a usage (or linearity) attribute at the kind level [2, 39, 51, 54, 55]. To express structural restrictions on these types, polymorphism over qualifiers and inequalities between qualifiers are used. Other approaches exist that attach linearities solely on function arrows [6, 11].

We have opted to use generalised bang/flip-bangs, but could have equally taken a constraint-based approach. For instance, we can imagine a typing of g (from § 2.5) that lifts the substructurality relation $\rho \preceq \tau$ to the type level:

$$\forall \nu.\,\forall\theta.\,\forall\alpha.\,\theta \preceq \alpha \Rightarrow (\mathbf{1} \xrightarrow{\theta}_\nu \mathbf{1}) \to \alpha \xrightarrow{\theta}_\nu \alpha$$

This typing captures both requirements regarding the partial application of g and the substructurality relation between θ and α. Indeed, Tang et al. [51] take a similar constraint-based approach and the type they give for g is almost identical to the type above. Tov and Pucella [54] take also a kind-and-qualifier based approach, but instead of introducing inequalities at the type level, they use meet and join operations on the qualifiers. As mentioned in § 4.4, introducing join and meet operation on modes would address the current limitations of bounded quantification in Affect.

The choice for bounded polymorphism is orthogonal to the other purposes of the paper. On the one hand, a constraint-based approach may be more intuitive than the one that uses bang/flip-bangs (! and ¡). On the other hand, bang/flip-bangs enjoy nice dualities and the encodings of one-shot signatures and are interesting from a theoretical perspective. We conjecture that the two approaches have the same expressiveness power if we introduce meet and join operations to Affect, but a detailed comparison between the two approaches is left for future work.

Substructural type and effect systems. Tov and Pucella [55] investigate the interaction of substructural types with control effects. They develop $\lambda^{\mathrm{URAL}}(\mathscr{C})$, which extends the λ^{URAL} calculus [2] with control effects in a generic way. To ensure the usage guarantees of substructural types are not broken, their type system treats affinely continuations that capture affine values. To achieve this, they utilise substructurality relations in the typing rules similarly to Affect by tracking

what is captured in a continuation. However, due to the use of standard single-context judgements, substructurality relations are more pervasive in their system compared to Affect as they appear in most typing rules. Furthermore, instantiating $\lambda^{\mathrm{URAL}}(\mathscr{C})$ with algebraic effects and handlers is a non-trivial task due to the sophisticated effect systems they require.

Hillerström et al. [25] devise an affine type system for a language with effects handlers that only allows one-shot effects. 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 and multi-shot continuations, and investigate the interplay between the two.

Tang et al. [51] combine linear types with multi-shot effects in a way closer to this paper. To ensure linearity, they devise a linear type and effect system that distinguishes between one-shot and multi-shot effects. Their approach relies on two dual notions of linearity: value linearity which governs how variables from the context are used, and control-flow linearity which tracks how many times control can enter a code block. Thus value linearity corresponds to restricting the Contraction rule and Weakening rule, whereas control-flow linearity corresponds to the *capturing condition* of Affect. Similar to Affect, their system supports effect polymorphism. A crucial difference is that their system enjoys principal types (by incorporating qualified types that allow specification of linear dependencies between types and effects), which can be completely inferred algorithmically. The motivation for our work differs: we devise an (affine) type and effect system for an OCaml-like language that has higher-order references in the style of ML. To the best of our knowledge, extending Tang et al.'s system with references in the style of Affect is challenging due to their syntactic type soundness approach. Our semantic approach to type soundness is well-suited to modelling unrestricted references (that can store one-shot continuations) through the support of impredicative invariants in Iris, as well as for modelling equi-recursive rows through step-indexing, and for mechanising all results in Coq via Iris's infrastructure.

Effect rows and effect polymorphism. There are multiple approaches to support effect polymorphism in row-based effect systems. The Links programming language adopts a Rémy [46]-style row polymorphism approach where duplicate operations cannot appear in the same row [22, 25, 51]. To ensure collisions do not occur during instantiation of effect polymorphic types, they take a kind-based approach which associates rows with sets of forbidden operations.

Affect borrows its effect rows from the Koka language [38], which in turn, are in the style of scoped labels [36]. Contrary to the Links language, Koka and thus Affect, allow duplicate operations. This simplifies the effect system since it does not need to ensure that effect rows contain unique effects albeit with some loss of expressiveness of effect polymorphic functions. Biernacki et al. [12] additionally employ a lifting construct that addresses Koka's loss of expressivity.

Tes [18], an unrestricted type and effect system built for an OCaml-like language takes an alternative approach where effect rows carry a disjointness hypothesis, which ensures that functions annotated with rows that have duplicate effects are not callable. Furthermore, they use dynamically allocated effect labels to guard against accidental handling.

Effect subtyping. Multiple algebraic effect systems support effect subtyping, such as core Eff [8], and $\lambda^{\mathrm{H/L}}$ of Biernacki et al. [12], Tes of de Vilhena and Pottier [18], and the Q°_{eff} calculus of Tang et al. [51]. Our effect subtyping relation is similar to that of Biernacki et al. [12] with the addition that we incorporate subtyping between the substructurality of types and effects.

Recursive effects. Recursive effects have been incorporated to calculi such as core Eff of Bauer and Pretnar [8], $\lambda^{\mathrm{H/L}}$ of Biernacki et al. [12], and $\lambda_{\circlearrowleft\circlearrowright}$ of Zhang and Myers [62]. All three approaches differ with the recursive effects of Affect in that their effect rows only contain operations and

their accompanied signatures live in a separate effect context. In Affect, effect signatures are incorporated into rows and via the use of an explicit recursive row $\mu\theta.\,\rho$. This makes it possible to specify recursive effects and their type specification in a unified way.

Semantic typing for effect handlers. The logical approach to semantic typing in Iris has been employed to prove soundness of many type systems [53]. Our work is influenced by that of Jung et al. [28] for Rust, and that of Hinrichsen et al. [27] for session types—particularly the semantics of the two-context judgements. Up to our knowledge, the semantic approach has not been applied to the combination of effect handlers and substructural types.

de Vilhena and Pottier [18] develop Tes, an unrestricted type system that introduces an alternative approach to address effect collisions (see discussion "Effect rows and effect polymorphism" above). To prove type soundness, they semantically interpret their type and effect system by building a unary logical relation using a variation of the Iris-based Maze program logic [15]. We also build a (unary) step-indexed logical relation to prove type soundness based on a program logic derived from Maze, but our semantic definitions differ since we consider a substructural type system and use effect rows inspired by Koka [38].

Biernacki et al. [12] present a calculus with effect handlers and one of their contributions is a binary, step-indexed logical relation, that allows reasoning about contextual equivalence of effectful programs. Their work is also mechanised in the Coq proof assistant, but does not use Iris. By virtue of taking a semantic approach and building a binary logical relation they also prove type soundness as a corollary. Zhang and Myers [62] present the $\lambda_{\diamondsuit\diamondsuit}$ calculus that supports effect handlers in an abstraction-safe manner. They too build a logical relation inspired by Biernacki et al. [12] to prove via context equivalence that abstraction-safe guarantees are satisfied.

Our contribution differs from the former two approaches in that we consider a language with mutable state. As a result, in our case propositions are *resource-aware* step-indexed propositions that can account for the heap as well as setting invariants on its state. Iris's support of impredicative invariants [50] is essential to treat reference types in an unrestricted way.

7 Conclusion and Future Work

We presented Affect—an affine type and effect system for an OCaml-inspired language with effect handlers and mutable state that distinguishes between effects that are one-shot or multi-shot. The one-shot language Affect$_{OS}$ (§ 3) could serve as an initial more realistic extension to OCaml 5 with a proper effect system that statically ensures one-shot usage of continuations (guaranteed by the type soundness theorem) and avoids OCaml's dynamic multi-shot usage errors. The full Affect language with multi-shot effects (§ 4) might enable OCaml to safely and efficiently support multi-shot effects. Future work remains to investigate the interaction with other OCaml features (such as modules, GADTs), investigate a linear (instead of affine) version of Affect with a 'discontinue' operation for unused continuations, and investigate how to make mode polymorphism and bang/flip-bang types practical through principal typing and (partial) type inference.

One could implement a (prototype) compiler 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, *i.e.,* that the compiled executable behaves exactly as described by the semantics of the source Affect program. To achieve this, our unary logical relation could be extended to a binary one that can prove soundness of optimisations for expressions that make use of one-shot and multi-shot effects.

Another direction is to investigate destructive reads for references that allow storing affine content while themselves being unrestricted (and avoid using the cumbersome `replace` pattern), a comparison of expressivity of constraint-based versus bang/flip-bang approaches, and to investigate deep handler typing rules that allow the return branch to capture affine resources.

Acknowledgments

References

[1] Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP (LNCS)*, Vol. 3924. 69–83. https://doi.org/10.1007/11693024_6

[2] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. 2005. A step-indexed model of substructural state. In *ICFP*. 78–91. https://doi.org/10.1145/1086365.1086376

[3] Andrew W. Appel and Amy P. Felty. 2000. A Semantic Model of Types and Machine Instructions for Proof-Carrying Code. In *POPL*. 243–253. https://doi.org/10.1145/325694.325727

[4] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

[5] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122. https://doi.org/10.1145/1190216.1190235

[6] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *LICS*. 56–65. https://doi.org/10.1145/3209108.3209189

[7] Henk P. Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

[8] Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *LMCS* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

[9] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *JLAMP* 84, 1 (2015), 108–123. https://doi.org/10.1016/J.JLAMP.2014.02.001

[10] Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax Journal of Functional Programming. *JFP* 11, 4 (2001), 395–410. https://doi.org/10.1017/S0956796801004099

[11] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical linearity in a higher-order polymorphic language. *PACMPL* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

[12] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

[13] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *PACMPL* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

[14] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *JFP* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

[15] Paulo Emílio de Vilhena. 2022. *Proof of Programs with Effect Handlers*. Ph.D. Dissertation. Université Paris Cité. https://inria.hal.science/tel-03891381

[16] Paulo Emílio de Vilhena and François Pottier. 2020. Hazel. https://gitlab.inria.fr/cambium/hazel/.

[17] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *PACMPL* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434314

[18] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *ESOP (LNCS)*, Vol. 13990. 225–252. https://doi.org/10.1007/978-3-031-30044-8_9

[19] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *TFP (LNCS)*, Vol. 10788. 98–117. https://doi.org/10.1007/978-3-319-89719-6_6

[20] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

[21] Robert Harper. 2016. *Practical foundations for programming languages*. Cambridge University Press.

[22] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. 15–27. https://doi.org/10.1145/2976022.2976033

[23] Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (LNCS)*, Vol. 11275. 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

[24] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs)*, Vol. 84. 18:1–18:19. https://doi.org/10.4230/LIPICS.FSCD.2017.18

[25] Daniel Hillerström, Sam Lindley, and John Longley. 2020. Effects for efficiency: asymptotic speedup with first-class control. *PACMPL* 4, ICFP (2020), 100:1–100:29. https://doi.org/10.1145/3408982

[26] Daniel Hillerström, Sam Lindley, and John Longley. 2024. Asymptotic speedup via effect handlers. *JFP* 34 (2024). https://doi.org/10.1017/S0956796824000030

[27] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP*. 178–198. https://doi.org/10.1145/3437992.3439914

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[29] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

[30] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[31] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

[32] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP*. 145–158. https://doi.org/10.1145/2500365.2500590

[33] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[34] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS)*, Vol. 10201. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[35] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

[36] Daan Leijen. 2005. Extensible records with scoped labels. In *TFP*, Vol. 6. 179–194.

[37] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP@ETAPS (EPTCS)*, Vol. 153. 100–126. https://doi.org/10.4204/EPTCS.153.8

[38] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. 486–499. https://doi.org/10.1145/3009837.3009872

[39] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in system F°. In *TLDI*. 77–88. https://doi.org/10.1145/1708016.1708027

[40] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS)*, Vol. 2142. 1–19. https://doi.org/10.1007/3-540-44802-0_1

[41] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT press.

[42] Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *FoSSaCS (LNCS)*, Vol. 2030. 1–24. https://doi.org/10.1007/3-540-45315-6_1

[43] Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *FoSSaCS (LNCS)*, Vol. 2303. 342–356. https://doi.org/10.1007/3-540-45931-6_24

[44] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP (LNCS)*, Vol. 5502. 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[45] Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. In *MFPS (ENTCS)*, Vol. 319. 19–35. https://doi.org/10.1016/J.ENTCS.2015.12.003

[46] Didier Rémy. 1994. Type inference for records in natural extension of ML. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*. MIT Press, 67–95.

[47] Rust Team. 2024. Module std::cell of the Rust standard library. https://doc.rust-lang.org/std/cell/.

[48] K.C. Sivaramakrishnan. 2015. Pearls of Algebraic Effects and Handlers. https://kcsrk.info/ocaml/multicore/effects/2015/05/27/more-effects.

[49] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. 206–221. https://doi.org/10.1145/3453483.3454039

[50] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS)*, Vol. 8410. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[51] Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *PACMPL* 8, POPL (2024), 1600–1628. https://doi.org/10.1145/3632896

[52] Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *PACMPL* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

[53] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *JACM* 71, 6 (2024). https://doi.org/10.1145/3676954

[54] Jesse A. Tov and Riccardo Pucella. 2011. Practical affine types. In *POPL*. 447–458. https://doi.org/10.1145/1926385.1926436

[55] Jesse A. Tov and Riccardo Pucella. 2011. A theory of substructural types and control. In *OOPSLA*. 625–642. https://doi.org/10.1145/2048066.2048115

[56] Orpheas van Rooij and Robbert Krebbers. 2024. Coq mechanisation of "Affect: An Affine Type and Effect System". https://doi.org/10.5281/zenodo.13907548

[57] Philip Wadler. 1991. Is There a Use for Linear Logic?. In *PEPM*. 255–273. https://doi.org/10.1145/115865.115894

[58] David Walker. 2004. Substructural Type Systems. In *Advanced topics in types and programming languages*, Benjamin C. Pierce (Ed.). MIT press, Chapter 1, 3–43.

[59] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355.

[60] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *I&C* 115, 1 (1994), 38–94. https://doi.org/10.1006/INCO.1994.1093

[61] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *PACMPL* 4, ICFP (2020), 99:1–99:29. https://doi.org/10.1145/3408981

[62] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *PACMPL* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318